Vrije Universiteit Brussel

FACULTEIT VAN DE WETENSCHAPPEN
Vakgroep Informatica
Laboratorium voor Programmeerkunde

Modularizing Advanced Transaction Management

Tackling Tangled Aspect Code

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Wetenschappen

Johan Fabry

Academiejaar 2004 - 2005

Promotors: Prof. Dr. Theo D'Hondt, Prof. Dr. Tom Tourwé

To my parents

Nederlandstalige Samenvatting

Grootschalige gedistribueerde systemen, bijvoorbeeld internet-applicaties zoals webwinkels en online banktoepassingen, moeten verzekeren dat gelijktijdig gebruik door meerdere personen de consistentie van de gegevens van het systeem bewaart. De defacto standaard hiervoor is het gebruik van transacties. Transacties zijn echter ontwikkeld voor één specifiek geval van het benaderen van gegevens: kortstondige benaderingen die weinig gegevens omvatten en geen interne structuur hebben. Bovenvermelde systemen willen echter vaak de gegevens op een gestructureerde wijze of voor een lange tijd benaderen en worden dan ook geconfronteerd met de beperkingen van transacties.

Structurele oplossingen voor de beperkingen van transacties zijn reeds gekend onder de naam van 'geavanceerde transacties', maar we zien dat geen enkele van deze oplossingen momenteel in gebruik is. Een belangrijke reden hiervoor is het feit dat deze geavanceerde mechanismen moeilijk te gebruiken zijn, m.a.w. dat het moeilijk is voor een programmeur om de code voor het gebruik van deze systemen te schrijven en te onderhouden.

In deze thesis bieden wij een oplossing aan voor de programmeur door softwareengineering-technieken toe te passen op het gebruik van geavanceerde transactiemechanismen. Meer bepaald passen we de technieken van aspectgeoriënteerd programmeren en domein-specifieke talen toe om de programmeur toe te staan op een hoog abstractieniveau met geavanceerde transactiemechanismen om te gaan.

Aspectgeoriënteerd programmeren (AOP) laat toe om de code voor bekommernissen die verspreid zijn over het systeem, zoals het gebruik van transacties, te verzamelen in één module. Deze module wordt aspect genoemd en de code in deze module wordt meestal geschreven in een specifieke aspect-programmeertaal. Het belangrijkste voordeel van een aspect is dat de aspect-code gemakkelijker te schrijven en te onderhouden is. Dit is omdat de programmeur zich volledig kan concentreren op die ene bekommernis en niet meer gestoord wordt door andere bekommernissen die dezelfde code

doorweven. AOP is reeds gebruikt om het gebruik van transacties te vereenvoudigen maar werd nog niet toegepast op geavanceerde transacties.

Als eerste deel in dit onderzoek hebben we dan ook een aspect gemaakt voor geavanceerde transacties. Hier hebben we geobserveerd dat de code voor een aspect zelf ook kan bestaan uit verschillende door elkaar geweven bekommernissen. Specifiek voor het aspect van geavanceerde transacties hebben we hiervoor een oplossing ontwikkeld: de aspect-programmeertaal KALA. KALA is gebaseerd op een bestaand formeel model voor geavanceerde transacties, genaamd ACTA, en laat toe om het aspect op te splitsen in de verschillende bekommernissen en deze volledig apart neer te schrijven. Dit geeft de programmeur niet alleen de mogelijkheid om een grote verscheidenheid van geavanceerde modellen te gebruiken als een aspect maar ook om nieuwe mechanismen te creëren als dit nodig is.

De programmeertaal KALA is dus een krachtige aspect-taal voor het beschrijven van geavanceerde transactiemechanismen die ook toelaat om nieuwe mechanismen te creëren als het nodig is. Maar deze kracht heeft ook een kostprijs en die is dat programma's in KALA complex en langdradig zijn omdat ze werken met elementen van de onderliggende implementatie van een model. Programmeurs die echter enkel bestaande mechanismen willen gebruiken hebben geen boodschap aan deze kracht en wensen de complexiteit en langdradigheid te vermijden. Hiervoor hebben we een tweede abstractielaag uitgewerkt, met behulp van domeinspecifieke talen.

Het gebruik van domeinspecifieke talen laat toe programma's te schrijven die werken met de concepten fundamenteel aan het domein, die dus abstractie maken van de implementatie van het domein. Toegepast op geavanceerde transactiemechanismen betekent dit dat we een aantal domeinspecifieke talen hebben ontwikkeld voor een aantal geavanceerde mechanismen. Er is een taal per mechanisme, en programma's in elke taal worden vertaald naar equivalente KALA-code. Code in elke domeinspecifieke taal behandelt de concepten inherent aan het model in kwestie. Dit zorgt voor een hoger abstractieniveau van de code en maakt de code ook summier. Daarenboven vormen de verschillende talen een familie: concepten die gedeeld worden door verschillende modellen, en dus ook door verschillende talen, worden telkens op dezelfde wijze neergeschreven.

Als resultaat van onze combinatie van aspectgeoriënteerd programmeren en domeinspecifieke talen is het nu ook mogelijk voor een programmeur het gebruik van een geavanceerd transactiemechanisme op een summiere wijze neer te schrijven, en dit volledig apart van de code van het systeem zelf. Naast deze bijdrage in het onderzoeksgebied van geavanceerde transactiemechanismen hebben we ook bijgedragen aan het onderzoeksgebied van AOP. Dit is eerst en vooral de observatie dat een aspect zelf kan lijden aan het probleem van verschillende door elkaar geweven bekommernissen. Onze tweede bijdrage is de verwezenlijking van het volledig apart specificeren van de code van een aspect.

Acknowledgments

A mere 300 pages can not contain all the things I have looked at and all the work which I have performed in the last five and three quarter years. However, I do have the opportunity here to thank all the people who have assisted me throughout this journey.

First of all, I thank my promotor Theo D'Hondt for accepting me as a Ph.D. student in his lab, giving me the freedom to explore the field and the support I needed to finish the Ph.D. Also, I am grateful to Tom Tourwé for accepting to co-promote this work.

I thank the members of my jury; Olga De Troyer, Viviane Jonckers, Awais Rashid and Peter Van Roy for taking valuable time out of their schedule to read through this large volume and for giving me feedback. My apologies for being so verbose.

Thanks to Johan Brichau for his invaluable help with Soul. Johan did come first in our little race but I have more pages, so I propose we call it a draw. Thomas Cleenewerck was an essential catalyst in the last phase of the thesis and a great support in the last few months and I thank him for that. Wolfgang De Meuter and Pascal Constanza have given me valuable insights in the final stages of the Ph.D. process and have helped me 'to get it right'. For this I am much obliged.

I distinctly recall a phrase told to me by Werner Van Belle, last year in Tromsoe, which really forced me to focus: "Look, you have six possible thesises here. Which one are you going to pick?". Without that sentence I would probably still be at it. So, Werner, for this I am forever in your debt.

Many thanks to the proof-readers, in addition to large contributions by Thomas, Wolf, Pascal and Werner, the following three have also aided in making this text: Jessie Dedecker, Andy Kellens and Kim Mens.

Thanks to my office-mate Dirk van Deun for switching on `pedantic` mode for me when I asked him to (and not forgetting to switch it off).

Stijn Mostinckx, Tom Van Cutsem, and Ellen Van Paesschen covered for me on

the CoDAMoS project in these last few months. Not needing to worry about it made my life much simpler and is greatly appreciated.

I thank all the current and former members of the PROG lab which I have had the pleasure to count as my colleagues during these years. You all helped me by providing interesting discussions, food for thought and occasional entertainment. I wish you all well.

Last but not least, thanks to Sara for coping with me, especially these last few months, and thanks to my parents for keeping the faith. It took a while, but I made it!

Contents

Chapter 1

Introduction

The story so far:
In the beginning the Universe was created.
This has made a lot of people very angry and has been widely regarded as a bad move.
— **Douglas Adams, "The Restaurant at the End of the Universe"**

It is telling that the earliest known instance of writing is not a story, nor a poem or a religious text. The oldest clay tablet that has yet been discovered, records a commercial transaction. Transactions have always been and still are the cornerstone of trade and their record-keeping is of vital importance. Therefore, it should come as no surprise that the concept of *transaction management* is also present in computer science, and takes a prominent role in many business-oriented computer systems.

The fundamental concept of transaction management in computer science seems, relatively speaking, as ancient as the clay tablet mentioned above, as it has been developed in the early age of computer science. This, however, does not mean that transactions are obsolete or irrelevant in current systems. On the contrary, transaction management permeates a large domain of applications, among which a significant group is the multi-tier distributed systems.

Multi-tier is a particular architecture for distributed systems: systems which do not consist of one application running on a single computer. Instead these consist of different, interacting applications distributed over different computers, which are connected by a network. Within the realm of distributed systems, multi-tier distributed systems are a class of client-server systems conceived specifically to handle a large amount of clients which work concurrently on shared data, usually kept in a database,

and communicating over a wide area network. Typical examples of such systems are web banking applications and e-commerce applications such as internet shops. The multi-tier architecture is successful in building such enterprise applications because it allows the distributed system to cope with a large number of clients connected over possibly slow network links, can integrate well with legacy applications and allows for reuse of application logic between different systems. Standards for multi-tier distributed systems are available, and the de facto standard is Enterprise JavaBeans [MH01]. EJB was created because of the demand for such a standard, and has also provided an extra impetus to use the three-tiered architecture where possible. As a result, in general, a large part of the applications accessible through a web browser, such as the web applications described above, but also web-based airplane reservation systems, intranet applications and web services, are built as multi-tier systems.

Multi-tier systems evolved out of large-scale mainframe client-server applications such as banking systems and monolithic airline ticket reservation systems. This evolution was spawned by the need to scale up client-server applications to handle a very large number of clients, which was caused, amongst others, by the advent of the internet. This evolution, combined with the growth in numbers of such systems have led to a change in the profile of the developers of these systems. Where previously the development of mainframe applications was the province of an elite cadre of expert programmers, this is no longer the case. Because of the sheer number and diversity of multi-tier systems currently in use, and being built, a *much larger number of developers are working on these systems,* and it follows that developing such systems now is performed by programmers of an average skill level.

Technologies and standards for multi-tier distributed systems, such as EJB, cater to this varying skill-set of programmers by aiding development through an offering of standardized services, such as remote communications, persistency of data and transaction management. The presence, in itself, of these services has caused distributed applications to be built using just these technologies, because they provide relief from the issues [TCLL03] of remote communications, persistency, transactions, and so on. Lesser skilled programmers can simply use the provided services, as is, in their applications, while higher-skilled programmers can tweak these services, or even roll their own, to obtain better performance.

One of these provided services is transaction management, which is a core theme of this thesis. As we said above, transaction management as a concept in computer science also has a significant history. It has been developed in conjunction with database technology and it has become the de facto standard for concurrency management when using databases. A transaction in computer science consists of a sequence of program instructions that are to be considered as an indivisible block. Such blocks, when executed concurrently, may not interfere with each other's data, and will therefore maintain global data consistency, i.e. keep the database in a consistent state.

More formally, a transaction is a protected form of program execution which has the following properties (Also known as the *ACID* properties) [GR93]:

Atomic: A transaction is an atomic sequence of actions, i.e. either all actions are executed, or none of them is executed.

Consistent: A transaction takes the database from one consistent state to another consistent state.

Isolated: The intermediate effects of a running transaction are isolated from the other transactions.

Durable: The results of a transaction are kept in persistent storage.

At run-time, a transaction monitor, also known as *TP Monitor* [GR93], ensures that concurrent transactions do not inadvertently work on each others intermediate results, and generally prevents the underlying data of a system to become inconsistent. This effectively prevents race conditions to occur, which is why transaction management has become the mainstay for concurrency management when using databases. This in turn has led to transaction management to be incorporated as a standard service in multi-tier technologies, which therefore include their own TP Monitor.

To summarize, in multi-tier distributed systems, a de facto standard for concurrency management is transaction management, and it is offered as a standard service in technologies that support multi-tiered systems. Consequently, all programmers who are implementing multi-tiered distributed systems, including lesser skilled programmers, are exposed to transaction management. Because multi-tiered distributed systems are so large in number, this means that right now, an exceedingly large number of applications and averagely skilled application programmers need to deal with transaction management.

1.1 The Need for Advanced Transaction Management

Transactions are not a silver bullet, and while they are widely successful, they have an important downside, which is rollbacks. As we see next, such rollbacks must be handled by the application, which pollutes the application code. Furthermore, there are limitations to applicability of transaction management, with regard to the way data is accessed. This is important because transactions are currently being used in multi-tier distributed systems *beyond the original scope* for which transaction management was conceived. When operating beyond the original scope, workarounds are often used to try and address the mismatch between properties of concurrency management using transactions and the concurrency management properties required by the application.

It is better to have a fundamental approach to address this mismatch, and we find this in Advanced Transaction Mechanisms.

In the remainder of this section we discuss the above topics, starting with rollbacks, next discussing the limitations of transaction management, followed by the need to match transaction management to the design of the application, and we end with Advanced Transaction Mechanisms.

Handling Transaction Rollbacks

A far reaching consequence and important downside of the use of transactions which is frequently underestimated is the possibility of *rollback of a running transaction.* When a transaction is rolled back, all the modifications performed by that transaction to the database are undone and the transaction is terminated, the end result being, seen from the vantage point of the database, that the transaction never occurred. A possible cause for such a rollback is a deadlock between two or more transactions, which occurs when these different transactions are all waiting for each other to complete. A transaction waits for another transaction when the first transaction wants to access data which is currently being used by the second transaction. Due to the isolation property, the first transaction can not see these intermediate results until the second transaction ends. When that happens, the first transaction can continue. However, if the second transaction is also waiting for the first to finish, because it wants to access its data, both will wait for each other indefinitely. Such a wait-for cycle is called a deadlock, and to break this cycle, one of the transactions will need to be rolled back by the TP Monitor.

A transaction rollback entails that the work performed during this transaction, with regard to the database, is undone. Side-effects or operations outside of the database that occurred during the transaction, however, cannot be canceled. This is because these are not within the scope of control of the TP Monitor. Consequently, an important restriction for transactional code is that it may not produce such side-effects.

Furthermore, rollbacks introduce an extra non-functional concern: handling the rollback. The application issuing this transaction will need to handle this exceptional situation, for example by re-starting the transaction. Fundamental to this exception is the fact that it can always occur, even if the application logic does not suggest a possibility, and this due to the possibility of deadlocks, which we discussed above. Consequently, as a result of using transactions, the application must always be prepared to handle transactional exceptions, such as rollbacks. This is a prominent consequence of using transactions, and is a downside because it pollutes the code with application logic to handle rollbacks, which is not required if no transactions are used.

A less immediately obvious, but still fundamental property of transaction man-

agement is due to the class of applications for which transaction management was originally conceived. This leads to a number of limitations for transaction management, and we discuss these next.

Limitations of Transaction Management

Transactions have been designed to treat small units of work, which only access a few data items, and take a short time to complete. Therefore, in applications that process large units of work, needing a long time to complete, or if these units of work have a complex hierarchical structure, i.e. outside of this domain, the transaction concept is *ill-suited to perform concurrency management.* For example, consider the lack of fine-grained rollbacks in the case of a complex hierarchically structured unit of work. Aborting just part of the hierarchy can not be mapped to the flat structure of a transaction and therefore entails aborting all of the work. As a second example we consider cooperative activities, such as the cooperative creation of a document. Here we want the ability to share some data between different cooperating transactions, which is impossible due to isolation between transactions. The third example we consider is a well-known drawback of transaction management when using long-lived transactions, and we outline this in more detail next.

If we extend transaction time to a few seconds or even further, to a few days, using what is known as *long-lived transactions,* the speed at which transactions can be processed by the TP Monitor drops dramatically. Other transactions which need to access data in use by the long-lived transaction will have to wait for the long-lived transaction to complete before they can proceed, which reduces processing speed. Furthermore, long-lived transactions are more likely to be involved in a deadlock [WR92], possibly leading to their rollback and the loss of all their work. This implies that the other transactions have been needlessly kept waiting, i.e. needlessly reducing transaction processing speed, and furthermore the lost work needs to be restarted, further reducing transaction processing speed as all the work needs to be redone.

Transaction processing speed is an important design element of transaction management, a testament to which is that standard benchmarks for TP Monitors, the TPC benchmarks [tpc04], are available. These benchmarks are maintained by a non-profit corporation, the Transaction Processing Performance Council [tpc01], created solely for that purpose, and test transaction throughput. The top performer on one of these tests, the TPC-C test, on 24 november 2004, is an IBM eServer p5, with an IBM DB2 database and a Microsoft COM+ TP Monitor, achieving 3,210,540 transactions per minute. That amounts to 53,509 transactions per second, or roughly five transactions per 10,000th of a second. Granted, the hardware is impressive (64 CPUs at 1.9Ghz, with a system cost of over sixteen and a half million US dollars), but this should not detract from the goal of the test, which is processing as many transactions per second

as possible. Therefore, it is clear that reductions in transaction processing speed due to the usage of long-lived transactions in the application should be avoided.

Considering transaction execution speed in multi-tier distributed systems, we see however that the time-frame in which a transaction is executed is of a different order of magnitude. For example, because in multi-tier systems the application logic can be spread out over multiple servers connected by a network, the execution of one transaction can involve multiple servers. As a result execution speed of transactions running in parallel drops significantly, and communication between the different servers over the network introduces a significant overall slowdown. Also, cases have been reported where a multi-tier application programmer uses the transaction service but is not fully aware of the impact of long-lived transactions. This allows the following scenario to take place: a unit of work is created by the programmer where in the middle of execution some information is asked of the user. Asking the user for this information takes a large amount of time, due to the communication with the client, and the time required for the user to fill in this information. Furthermore, there is no guarantee that the user will immediately fill in this information, so this can take an arbitrary amount of time. As a result, due to the need for user interaction, the programmer has created a potentially long-lived transaction, which will significantly slow down the other transactions within the application.

Matching Transaction Management to the Design of the Application

The underlying problem of when an application encounters the limitations of transaction management is that the fundamental properties of the transaction model regarding long-lived transactions, granularity of rollback, sharing of intermediate results, et cetera, only align well with a limited kind of concurrent behavior of the application: accessing a small number of data items in a short and isolated unit of work. If the concurrent behavior of the application lies beyond the design scope of classical transactions, there is a *mismatch between the properties* of the transaction model and the concurrency management properties requested by the application, i.e. the design of the application does not match with the design of the concurrency management scheme of transaction management.

Because of the *growing amount and large diversity* of multi-tiered distributed systems we can be certain that a significant subset of these applications are faced with these transactional issues. A testament to this is that workarounds for the mismatch in design between transaction management and the application are known to be in use [GM83]. For example, take the case of a long-lived transaction resulting from user input required in the middle of the unit of work, seen above. In this case, the workaround is ensuring that all user interaction is performed before the transaction starts. This can be difficult, however, for example, if the input required from the user depends on some

computation performed in the beginning of the transaction. Other workarounds range from letting the user resubmit all data if a transaction is rolled back, to skipping the use of transactions and processing everything in batch. Such workarounds, however, *do not address the crux of the problem,* while the demand for multi-tier applications which overcome this design mismatch increases. Take for example the rise of internet banking: whereas previously banking operations were performed overnight in batch, with internet banking such operations are expected to be performed directly, and the results should be immediately visible to the user. Similarly, a user of a web shop will not be inclined to resubmit all the information previously entered in case of the roll-back of her transaction. Instead, she probably will be discouraged, lose faith in the system or be annoyed with the procedure, and may decide to move to a competitor's site.

It is clear that what is needed to address the mismatch in design of the application with the design of transaction management is, instead of workarounds, *a fundamental approach* to extend such classical transaction management with more advanced forms.

Advanced Transaction Mechanisms

To address the above issues of performance, cooperation and fine-grained rollback handling, among others, a number of *advanced transaction mechanisms* (ATMS) have been developed, mostly between 1981 and 1997. An impressive number of alternate ATMS can be found in the literature, and two books have been published about the subject [Elm92, JK97]. However, each ATMS usually focuses on one single issue, and no overall system has been developed which treats a large number of the above issues of classical transactions. We now briefly introduce the two best known ATMS, as examples.

The most well-known ATMS is *nested transactions* [Mos81], which enables a tree of hierarchically structured transactions to be built. In such a transaction tree, a child transaction also has access to the data used by its parent transaction, and this recursively up to the root transaction. Furthermore, when a child transaction commits its data, this is not committed to the database, but instead to its parent, which is now responsible for committing this data. Aborting a child does not entail that its parents are aborted, which gives us a more fine-grained mechanism for handling rollbacks.

A second example of ATMS is *Sagas* [GMS87]. Sagas relaxes the atomicity requirement of long-term transactions by splitting them into a sequence of atomic sub-transactions. The sequence of sub-transactions should either be executed completely or not at all. Splitting the long-term transactions releases intermediate results earlier, which increases concurrency as other transactions can execute concurrently, and decreases the probability of deadlocks, since after each sub-transaction all data is released, and each sub-transaction will probably require less resources than the complete

Saga. However, to rollback the Saga extra work needs to be done: *compensating actions* must be executed to undo the effects of already committed sub-transactions. Hence, the application programmer should define a compensating transaction for each sub-transaction, which performs a semantical compensation action. To rollback a Saga, the TP Monitor aborts the currently running sub-transaction, and subsequently runs all required compensating transactions in reverse order.

In general, available ATMS differ greatly due to the fact that *each ATMS was designed to address a specific subset* of the limitations of classical transaction management. The above two ATMS, nested transactions and sagas, illustrate this: nested transactions addresses the granularity and scope of rollbacks, while sagas addresses the issues with long-lived transactions. While both models are hierarchical, sagas has only one level, while nested transactions has an unlimited nesting depth. In nested transactions results are passed to the parent, while in sagas they are written to the database. Lastly, performing a rollback is quite intricate for the TP Monitor in sagas, and straightforward in nested transactions.

As we said above, in multi-tier distributed systems transaction management is used for purposes beyond the original scope for which it was designed. As a result in some applications, for example, long-lived transactions pop up, drastically reducing application performance, where it would be better to use sagas, or an inherently tree-structured computation will be coded as one classical transaction, where it would be better to use nested transactions. Furthermore, due to the large amount and large diversity of multi-tiered distributed systems we can be certain that a significant subset of these applications are faced with these transactional issues, *which have already been solved by an ATMS.* Therefore, to address these issues in these systems, we at least need to be able to use an ATMS the design of which better fits the design of the application.

Summary

In brief, classical transaction management is designed to perform concurrency management for small units of work which only access a few data items, and takes a short time to complete. Not taking this into account results in reduced performance or is tackled by the use of workarounds which do not address the crux of the issue. Fundamental approaches to address these issues, in the form of advanced transaction mechanisms have been developed, each mechanism addressing a specific subset of these issues. Because of the large amount and diversity of multi-tiered distributed systems, which all use transaction management, we can be sure that these systems will be faced with these issues, and therefore they should be able to use such advanced transaction mechanisms where advisable.

1.2 Outline of the Contributions

In this dissertation, we address the problem of using advanced transaction management in multi-tiered distributed systems. Before giving an overview of the problem and our solution, we provide a brief outline of the work performed. The different topics we briefly touch here are summarized in the remainder of this chapter, and we also refer to parts of the text that discuss these different topics in more detail.

To modularize the use of an ATMS in a given application, we have used aspect-oriented programming (discussed in chapter 3) to describe ATMS as an aspect. We developed our proper aspect language, named KALA, and aspect weaver (introduced in chapter 8). KALA allows a wide variety of ATMS to be used in the application, because it is based on the ACTA formal model for ATMS (discussed in chapter 6). Furthermore all code for the ATMS concern is written in KALA, making the core application logic syntactically oblivious of the ATMS concern.

When studying advanced transaction management (outlined in chapter 2), and more particularly the structure of the code required to use such models, we have observed the problem of tangled aspect code (introduced in chapter 4). Tangled aspect code occurs when an aspect itself consists out of multiple cross-cutting sub-concerns. Current aspect tools provide inadequate support for modularizing these sub-concerns, forcing the developer to write aspect code which itself is tangled, hence the name tangled aspect code. KALA was designed to tackle the problem of tangled aspect code for the ATMS aspect, by providing a modularization mechanism that allows the different sub-concerns to be expressed in different modules (introduced in chapter 9).

KALA programs describe the inner workings of an ATMS and, as a consequence, are quite large and complex. To increase ease of use of ATMS for application programmers, we have added an extra level of abstraction. We have built a number of domain-specific languages for ATMS (introduced in chapter 10), each language reflecting the concepts of a specific ATMS, instead of its inner workings. If a DSL for a given model is available, the application programmer can write code in the DSL for that ATMS, working with the concepts of the model (illustrated in chapter 11), instead of KALA code, which treats the inner workings of the model. The different DSLs are built on top of KALA: internally the DSL compilers translate DSL code to the equivalent KALA code before invoking the KALA weaver on that code. This provides for extensibility as it aids development of a new DSL. For a new ATMS, the DSL compiler solely needs to translate the DSL code to the equivalent KALA code.

In order to have run-time support for ATMS we built a custom TP Monitor, called ATPMos (introduced in chapter 7). ATPMos is also based on the ACTA formal model and provides run-time enforcement of the specifications declared in KALA code.

Figure 1.1 gives an overview of how these different topics are related. The remainder of this chapter provides a more extensive summary.

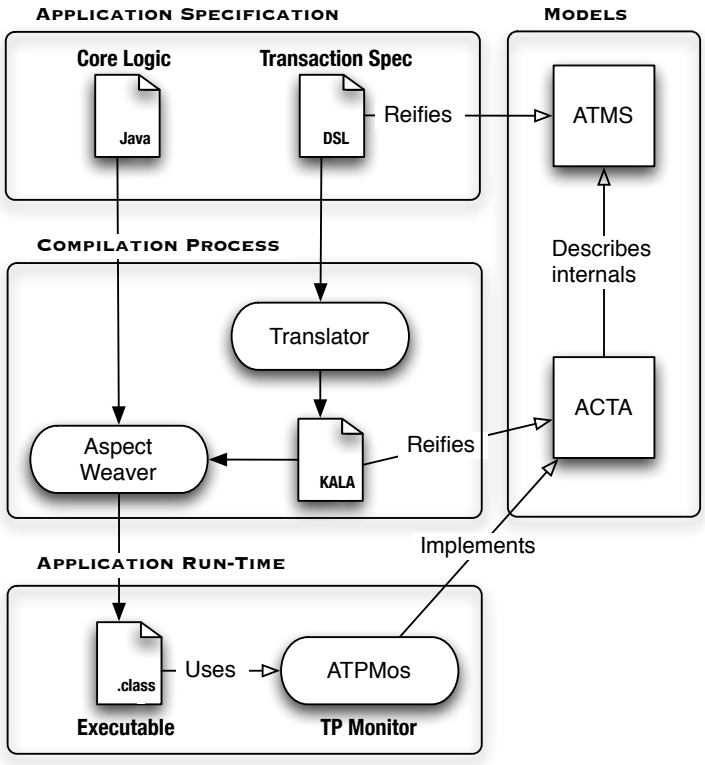

Figure 1.1: Overview of how the different topics in the thesis are related.

1.3 Using Advanced Transaction Management

In general, when writing an application which uses transactions the application programmer has to mark the start and end of each transaction in the application code, and what to do in case of a rollback. In this *transaction demarcation code,* the programmer will also associate each data access with a running transaction. At run-time TP Monitors are steered by this demarcation code, and will mediate concurrent accesses to the data by the different transactions within the application. Writing such demarcation code has been recognized as being difficult work [KG02, SLB02, RC03], and the resulting demarcation code is often found to be incomplete, or even erroneous. A telling example of this is found in [TCLL03] where in an application for a financial institution, some transaction rollbacks were simply ignored, leading to a security hole.

Modularizing Transaction Demarcation Code

A fundamental reason why transaction demarcation code is difficult to write, is that it can not be encapsulated into one module using traditional software engineering techniques. This observation is not new, it has already been made by several members of the research community on *Aspect-Oriented Programming* (AOP) [KLM+97]. In general, the AOP observation is that in programs that rely on the classical, hierarchical modularization mechanisms, such as modules, procedures and classes, the code of certain concerns is always spread out over different modules, instead of grouped into one module.

An example of such a *cross-cutting concern* is transaction management. Indeed, if we let transaction boundaries coincide with method boundaries, as is the norm in multi-tier technologies, we see that each method which is transactional is tainted with demarcation code. The method not only addresses the core concern of the application, but tangled with this code is transaction demarcation code. As a result, transaction demarcation code is scattered throughout the entire system, crosscutting different modules, where it is intermingled with the other code.

The problem with such tangling of code addressing different concerns is that, although, conceptually, these concerns can be reasoned about independently, at code level they become tightly coupled. This runs contrary to the software engineering principle of *separation of concerns* [Par72, HVL95] , which states that each concern of the application should be addressed in one and only one module of the program. Because of the lack of separation of concerns, integration of a new concern is difficult, as the programmer must manually add code in the correct locations. Furthermore, when debugging one concern, the programmer is needlessly faced with the code treating other concerns, requiring manual extraction of the concern being tackled out of the multiple concerns present in the code. Maintenance is also affected, as for main-

tenance the same concern extraction is required to understand and modify the code, and when changes are made to one concern this also affects the code for the other concerns, as they have become tightly coupled. In summary, and applied to the concern of transaction management, the above signifies that adding the concern of transaction management makes the application code more complicated, making it more difficult to implement and maintain such code.

AOP is a prominent technique which does allow for *modularization of cross-cutting concerns,* by defining them as *aspects*. Aspects can themselves specify how they cross-cut other modules and exactly where or at what points they do so, and are described separately from the main program, in a special-purpose aspect language. At compile-time, a tool, called the aspect weaver, combines the code from the main program and the aspect code into one executable. The resulting separation of concerns, not only at design level but now also at the implementation level, is known to greatly enhance implementation and maintenance [SLB02, RC03].

AOP has been applied to separate out transaction demarcation code for classical transactions [KG02, SLB02, RC03] into a module. This work has shown that AOP succeeds in modularizing transaction demarcation code, *providing effective relief* from the difficult task of manually inserting and maintaining transaction demarcation code all over the application. For *advanced* transaction management, however, no work has yet been published which modularizes the concern of ATMS.

Using an ATMS

To use an ATMS in a multi-tier application, the application programmer will also have to write demarcation code for that ATMS, as in classical transactions. However, using an ATMS compounds the above problem of complexity of the code, because demarcation code for ATMS will be even *more complicated* than for classical transactions. Take, for example, the Sagas model, where the programmer does not specify just one transaction. For Sagas, he needs to specify a sequence of different subtransactions, and furthermore needs to declare compensating transactions for each sub-transaction. In other words, the demarcation code now needs to perform much more tasks then previously. In addition to the start and end of transactions, and marking data accesses, demarcation code must also include the extra information mentioned above. Furthermore, compensating transactions must be fired in the correct order, namely the inverse order of the running of the sub-transactions. Determining this sequence must therefore also be performed in the demarcation code: logging the sequence of sub-transactions when they run, and ensuring that at compensation time the compensating transactions are fired in the right order. If we consider another example; nested transactions, the developer also needs to define extra information in the demarcation code. A child transaction, when running, must identify which currently running trans-

action is its parent. The child then reuses data accessed by the parent and, at commit time, delegates all its work to the parent. Again this implies that the demarcation code will contain these additional tasks, making it more complex.

Put briefly, transaction demarcation code for ATMS will be more complex than transaction demarcation code for classical transactions, as it must perform more work. However, the complexity of demarcation code for classical transactions is already considered to be a problem when considering the impact this code has on ease of implementation and maintenance of the application. As the complexity of demarcation code for classical transaction is already a known issue, adding extra work only compounds the problem. In other words, *writing demarcation code for ATMS is even more problematic,* as this code is more complex. Therefore the relief of a good modularization for demarcation code of ATMS will be even greater than the relief of a good modularization of classical transactions.

Using Multiple ATMS

As a consequence of the large number of multi-tier distributed systems, there will be a significant amount of systems which can benefit from using ATMS, so many people should be able to work with an ATMS. However, combine this with the wide variety of multi-tier distributed systems, which allows many shortcomings of the classical model to be encountered, and the limited focus of each ATMS, which yields that knowledge of just one ATMS will only be sufficient to aid in development of a small sub-set of such distributed systems. Therefore, programmers for multi-tier distributed systems must be able to *use a wide variety of existing ATMS.* This allows their knowledge to be applied in many systems, and for each case the design of the ATMS to fit the design of the application, i.e. the specific transactional properties of the system being built are supported by the ATMS.

While a wide variety of ATMS exists, there is, however, no certainty that the transactional properties required by the system being built are provided by an existing ATMS. In other words, we cannot be sure that an ATMS exists of which the design fits the design of the application. Therefore, in such cases, we should allow a programmer versed in ATMS to *design a new ATMS* that better fits the design of the application. This could be performed either by tweaking an existing ATMS, by modifying certain properties of this ATMS, or by designing a completely new ATMS from scratch. This new ATMS can then be used not only for that specific application, but also for other applications which require the same transactional properties.

Summary

To recapitulate, transaction demarcation code is difficult to write because it can not be modularized. This is a known problem for classical transactions, but it is even more pertinent with ATMS, because the demarcation code for these models is far more complex. What is needed is a solution to modularize demarcation code for ATMS that is general enough to cover a wide variety of existing ATMS and allows for the tweaking of an existing ATMS or even the creation of a new ATMS. Also, this modularization must enable the average multi-tier distributed systems programmer to write demarcation code for many ATMS. This will allow an application programmer to choose the ATMS of which the design best fits the design of the application, yielding application-specific advanced transaction management.

1.4 Application-specific Advanced Transaction Management

The goal of this thesis is to allow an average application programmer (who is not necessarily an expert on ATMS) of a multi-tiered distributed system to use an advanced transaction model of which the design is best suited to the design of the application, i.e. to have *application-specific advanced transaction management.* The application programmer will be able to chose from a wide variety of ATMS, or even be able to roll his own ATMS when needed. Furthermore, we increase ease of use of these ATMS by modularizing transaction demarcation code in such a way that coding skills for one ATMS are easily transferable to others, where applicable. This will allow an application programmer to use many different models without requiring detailed expertise about ATMS internals. The demarcation code will be gathered in one module, and abstracted to the level of concepts present in the ATMS. This shields the application programmer from the technical details involving the inner workings of an ATMS and the intricacies of manually writing transaction demarcation code.

To attain the goal of application-specific advanced transaction management, we *apply existing techniques from software engineering, marrying Aspect-Oriented Programming [KLM+97] and Domain-Specific Languages [vDKV00].* Using these two techniques allows us to raise ease of implementation and maintenance, but without losing general applicability. We achieve this through a two step approach, the first step using Aspect-Oriented Programming and the second step using Domain-Specific Languages. We now explain each step in more detail.

Using AOP to Modularize ATMS Demarcation Code

As we said above, the goal of Aspect-Oriented Programming (AOP) [KLM+97] is to modularize cross-cutting concerns as aspects. Work has already been done to modu-

larize classical transactions as an aspect [KG02, SLB02, RC03], but this does not take ATMS into account.

To modularize *advanced* transaction management as an aspect, we first conceptually analyze different existing ATMS, and determine their properties. From this analysis, we will see that we should not consider such an ATMS as one monolithic block, but as *a composition* of a number of sub-concerns. Examples of such concerns are the management of rollbacks and allowing intermediate results to be visible to other transactions. This view on ATMS allows us to change the design and implementation of one of the constituent concerns, which amounts to tweaking the ATMS to exhibit a slightly different set of properties. In other words, by changing one of the concerns constituting an ATMS, we can change the design of the ATMS better to fit the application. We can also go beyond tweaking, and compose a new ATMS, by combining existing concerns of different ATMS into one new ATMS. This allows us to effectively tailor an ATMS to the design of the application at hand.

To facilitate tailoring an ATMS to the design of the application, we should *modularize the different concerns of an ATMS into different modules.* This allows easy modification of the implementation of such a concern, and in general brings all the benefits of separation of concerns to the design and implementation of an ATMS. We therefore look at existing solutions for modularization of transaction management using AOP, to determine their capabilities with respect to composing the ATMS aspect out of different modules. Up until today, *only one* aspect language and weaver have yet been used to modularize transaction management as an aspect: AspectJ [Asp05]. In general, AspectJ is the best known and most used technology to implement aspects, and many tools with features similar to AspectJ have been created. We find, however, the composition features of AspectJ to be too weak for our needs. Conceptually, AspectJ and similar tools do not allow us to define the different concerns of an ATMS in separate modules, and combine these into one aspect when weaving. If we use AspectJ or similar tools, we are forced to manually compose the different concerns in the aspect code. We show that this leads to aspect code (which treats different concerns of the ATMS) that itself consists of multiple tangled concerns, a phenomenon which we call *tangled aspect code.*

To our knowledge, we have uncovered *the first case in which an aspect, ATMS, is found to be composed of different subconcerns, and the aspect code itself to be tangled code.*

KALA: An Aspect Language for ATMS

To address the problem of tangled aspect code for ATMS, we use domain information for this aspect, advanced transaction management, to create *an aspect language and corresponding weaver specifically for ATMS.* This language is called *KALA,* (**K**ernel

Aspect **L**anguage for **A**TMS) and allows for separate specifications of the different concerns within an ATMS, and for these modules to be merged into one aspect definition before weaving.

As a basis for the design of KALA, we *use an existing formal model for ATMS, ACTA* [CR91]. ACTA was created as a common framework in which it is possible to specify different ATMS. The purpose of ACTA is to be able to reason about a wide variety of ATMS, such that commonalities and variabilities between the different models can be deduced, and new models can be synthesized by combining features of different existing ATMS. In ACTA, an ATMS is formally defined by using three low-level building blocks (called axioms) to describe the properties of an ATMS. A wide variety of ATMS have been formally described in ACTA, and a number of these descriptions have been published. These descriptions are quite complicated and lengthy: for example, nested transactions is defined using nineteen axioms, which are not that straightforward, as can be seen in [CR91]. The underlying cause for this is that the ACTA axioms work in terms of the fundamental underlying principles of ATMS, in order to be able to support such a wide variety of ATMS. These axioms are very fine-grained in comparison with the concepts presented by the ATMS to the transaction programmer. For example, a concept we have already mentioned when discussing the Sagas ATMS is the compensating transaction, which defines a semantic undo for a given action. In ACTA, this single concept requires three different axioms to be specified. In general, the ACTA axioms need to be combined extensively to form the resulting ATMS concepts. However, the key advantage of ACTA is that it covers a wide variety of ATMS. Therefore, because we use ACTA as a base for KALA, KALA programs are able to express many ATMS and allow the application programmer to use an ATMS the design of which is best fit to the design of the application.

The KALA language uses the ACTA axioms as statements, allowing ACTA specifications to be programmed as KALA code. In other words, KALA allows us to specify the transactional behavior of the application *in terms of the ACTA primitives,* which enables us to use a wide variety of ATMS in multi-tiered distributed systems. If a formal specification for an ATMS can be given in ACTA, we can also express this specification as a KALA program. This means that an application programmer can either use an existing ATMS, if its properties fit the transactional properties required by the application, or create and use a new ATMS if no existing ATMS fits. Furthermore, KALA allows for *modularizing such specifications.* As we have said above, an ATMS is not one monolithic entity, but rather a composition of different concerns. In KALA, we can write each concern as a separate module, and compose these to form a full ATMS specification. This brings the known benefits of separation of concerns, discussed above, to the process of writing and maintaining KALA code, i.e. creation and usage of an ATMS.

As we said above, the ACTA axioms are, conceptually, at a lower level than the

concepts present in the different ATMS, because different axioms need to be combined to form one ATMS concept. As a result, the programs in KALA describing ATMS specify the inner-workings of these ATMS and are quite large. So while using AOP, we have separated out transaction management from the core concern of the application, when we consider the KALA code we see that it still needs to deal with many interacting sub-concerns. Clearly this level of complexity is not suitable for the application programmer as he only needs to reason in terms of the concepts relevant at that time, i.e. the abstractions offered by an ATMS. Consequently, while we have enabled application-specific advanced transaction management, it is beneficial to go one step further, through the use of a Domain-Specific Language for each ATMS.

DSLs: From ATMS Implementation to ATMS Concepts

Domain-Specific Languages (DSL) [vDKV00] are little languages that are specifically designed to express applications in a particular domain. This entails that the language constructs of a DSL directly reflect the concepts of the domain and hide the DSL programmer from non-domain-specific technical issues. As a result, using a DSL for each ATMS shields the application programmer from the technical details involving the inner-workings of the ATMS he is currently using. Also, the specifications written in the DSL are more declarative and are closer to the intended behavior [vDKV00].

We *raise the level of abstraction* of KALA code to the level of concepts present in an ATMS, using a number of Domain Specific Languages we have built. We create *specific DSLs for multiple ATMS,* each DSL reflecting the abstractions used in that model, and we illustrate how a DSL can be built for a newly created ATMS. Using an ATMS DSL to specify how an application uses an ATMS requires much less code than the equivalent KALA specification. The DSL specification is much more concise due to the fact that the decomposition of ATMS concepts into ACTA axioms is not required, as it is automated by the preprocessor. This makes *specifying the code in the DSL less error-prone* as less operations need to be considered and combined. Declaring transactional properties at the level of the concepts of the ATMS therefore makes it much easier to work with the code, because these concepts are immediately apparent, instead of the KALA specification, where the programmer must mentally construct these out of the primitive operations.

We have chosen to create different DSLs over adding abstraction and modularization mechanisms to KALA because of two reasons. First, DSLs ensure that we can completely shield the application programmer from the internals of the ATMS. Second, while ensuring correct use of abstraction and modularization mechanisms can be a tricky task, a DSL can embed such constraints in the language, which eliminates possible causes for errors.

Creating a DSL for an ATMS is eased by the availability of KALA. KALA can

be used as an intermediate representation for transactional specifications. At compile time, under the hood, a DSL compiler can simply translate the DSL specification into the equivalent KALA program, instead of having to produce executable code. The DSL compiler then calls the KALA aspect weaver which combines this code with the application code to create the final executable. This process in two steps is illustrated in figure 1.2.

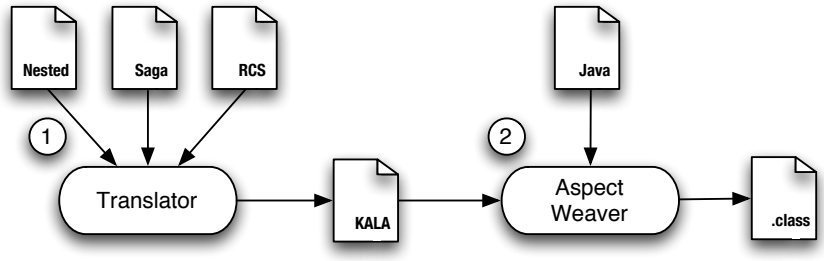

Figure 1.2: At compile time first code in the DSL is translated to the equivalent KALA code, which second is woven into the Java code.

The use of a DSL for each ATMS would seem to imply that programming skills in one ATMS are non-transferable to other ATMS, but this is not the case. This is because we do not have a disjoint set of DSLs, but a family of DSLs, where ATMS concepts are reused as much as possible. For example, the concept of a compensating transaction can be found in many ATMS, and in each of the DSLs we created for these models, a compensating transaction is declared in the same way, using the same syntax. Reusing concepts as much as possible ensures that knowledge of one ATMS is partially transferable to other ATMS, because common concepts of the ATMS are simply reused in the DSL. For example, a ATMS we have not yet discussed is Relatively Consistent Schedules, which can be seen as an extension to Sagas. In the DSL for this ATMS, all concepts of Sagas are reused, and one new type of declaration is added, which corresponds to the extension made to Sagas.

Summary

The use of a family of DSLs to raise the abstraction level of KALA code concludes the description of our approach to enable application-specific transaction management. We achieved this by first designing an aspect language, KALA, which allows for modularization of the ATMS concern as an aspect. In KALA, the transactional properties of an application are declared using fine-grained constructs taken from the

ACTA formal model for ATMS. KALA code itself can also be modularized, so that each module reflects one concern of the ATMS being used. KALA code, however is quite verbose and complex as it treats transactional properties at a very fine-grained level. To tackle this, we further raise the level of abstraction to the level of concepts present in the ATMS. We defined a family of DSL's, one DSL per ATMS. Using these DSLs completely shields the application programmer from the implementation issues of an ATMS, and allows the programmer to work at the level of the concepts present in the ATMS.

Having described our technique to enable application-specific transaction management, through the use of AOP and DSLs, from a birds eye perspective, we can now detail the contribution of and benefits yielded by our approach, as a result of the union of the advantages both offered by AOP and DSLs.

1.5 Contributions

To the best of our knowledge, this thesis is the first research that has been performed on the topic of using advanced transaction models that addresses the *software engineering perspective* of using an ATMS by application programmers. More specifically, it is the first time the software engineering techniques of AOP and the use of a DSL have been applied to alleviate the problem of demarcation code for ATMS. This enables these models to be used by a large number of application programmers. We state that this is the first research on this topic as, to the best of our knowledge, the most advanced research in this area has been performed by Kienzle and Gerraoui [KG02] and Procházaka [Pro01]. The former limits itself to considering how optimizations of classical transaction management can be performed using AOP, while the latter does not achieve a full modularization, and only supports a select few advanced models. We go beyond classical transaction management, and systematically treat a wide variety of advanced forms of transaction management.

The major contribution of this work is the *successful modularization of ATMS,* which can be broken down in three different parts. First, we built a general aspect language, KALA, in which we can express a wide variety of ATMS, separate from the application logic, and which allows for modularization of the different concerns contained within such an ATMS. Second, on top of this aspect language a *family of Domain-Specific languages for ATMS* has been created, where each language reifies the concepts exposed by the ATMS for which it has been created. Third, we have shown the *extensibility of this solution* by creating a new ATMS and writing KALA code for it, along with a DSL for that language.

Alongside the above research contributions, two technical contributions have also been made. First a general TP Monitor supporting a wide variety of ATMS, based on

the ACTA formal model has been created. Second, an aspect weaver for KALA has been implemented, which generates code using this interface.

We shall now discuss all of the above contributions, outlining the benefits that result from each contribution.

The Kernel Aspect Language for ATMS

KALA is an aspect language, which first and foremost means that the concern of advanced transaction management is *separated out* from the core concern of the application. This yields separation of concerns not only at the conceptual level, but also at the level of the code. In our examples we see, as a result, that *ease of implementation* of applications when using KALA is much higher than when writing demarcation code by hand. In these examples the use of KALA not only decreases the amount of demarcation code which has to be written, but the complexity of KALA code compared to manually written demarcation code is also lower. This is because in the corresponding KALA code a number of transaction-related sub-concerns no longer need to be addressed, which also eases implementation and maintenance.

KALA is not model-specific, but *a general aspect language for the domain of ATMS.* As it is based on the ACTA formal model for ATMS, it therefore allows for a large variety of ATMS to be programmed: KALA programs define transactional properties of the application at the level of the ACTA formal model. This ensures that the system is widely applicable because it allows the KALA programmer to write code using many ATMS, including models for which no model-specific language has yet been written.

Tangled Aspect Code

In building KALA, we have *identified a shortcoming* of the aspect language and weaver which has currently been used to modularize transaction management as an aspect: AspectJ. AspectJ does not allow us to cleanly modularize the different concerns within one ATMS. As a result, AspectJ code for an ATMS aspect will handle the different concerns within the aspect in a tangled way, disallowing the advantages of using separation of concerns when designing and implementing an ATMS. We name the phenomenon of aspect code itself being tangled with different concerns *tangled aspect code.*

It would be beneficial if AspectJ, specifically, but also other general-purpose aspect weavers, were able to tackle the issue of tangled aspect code. This would allow an aspect which is composed of different concerns to be expressed in different modules, even though the corresponding aspect code is tangled aspect code. This brings *the benefits of separation of concerns at the aspect level,* yielding the advantages in

implementation and maintenance for aspect code.

Model-specific languages

Layered on top of KALA are a number of model-specific languages, each treating one specific ATMS. These languages *reify the concepts specific to the model they treat,* rendering the code in these languages more abstract. Writing code at such a high level of abstraction has as a first advantage that it has become understandable for the average application programmer. Using our approach, an application programmer with a basic understanding of the concepts in the model will be able to use this model in the application. The second advantage of the high level of abstraction is that *this code is very concise,* making it easier to focus on the essentials when developing or maintaining that code. Unburdened by needless verbosity, the transactional code is now easily written and adapted, leading to quicker development.

As we provide a *family of Domain-Specific Languages,* one language per ATMS, there is as much reuse of language concepts as possible which minimizes the learning curve when going from one ATMS to another. In our approach, this is possible because common concepts between different ATMS, such as, for example, the use of compensating transactions, are specified the same way in the different languages. As a result, the programmer only needs to learn the different concepts of the new ATMS to be able to use it.

Extensibility

Our approach is extensible as it has the ability to *treat a new ATMS,* i.e. an ATMS for which no model-specific language has yet been created. We have created such an extensible system thanks to the matrimony of AOP and DSLs and the general applicability of KALA.

Because KALA is a general aspect language for ATMS, a model not previously covered but expressible in the ACTA formal model for ATMS should be expressible in KALA. In other words, to use a new ATMS, a programmer versed in KALA and the specifics of the new model, can write KALA code to use the new model in the application. If this model is used frequently, however, it would be advantageous to also raise the abstraction level for application programmers using this ATMS. To ease the use of this new ATMS, only a new DSL has to be created which translates the high-level specification to the equivalent KALA program. Furthermore, in writing this DSL, the creator of the DSL translator can reuse the concepts already supported by the existing DSL family, and therefore only needs to focus on the new concepts used by that ATMS.

General TP Monitor for ATMS

All code woven by the KALA weaver uses ATPMos: a general TP Monitor for ATMS based on the ACTA formal model, which we created. This yields a number of technical benefits, with regard to run-time behavior. First of all, as the different ATMS use the same underlying run-time infrastructure, this allows the application, or applications, that employ ATPMos to use *different ATMS at the same time.* This enables these applications to mix and match a virtually unlimited number of ATMS, where needed, whereas in current ATMS TP Monitors this is simply impossible as these are model-specific. Furthermore, having a single TP Monitor, instead of many, *leverages the improvements* of this TP Monitor to all ATMS in one fell swoop. Advances in speed, implementation of recovery, adding explicit multi-database support, et cetera, will be immediately available to all ATMS, instead of to just one model, as is the case with model-specific TP Monitors.

KALA weaver

Having a source-code weaver for KALA shows, first of all, that *KALA specifications are translatable to the equivalent transaction demarcation code.* Also, as we give an outline of how these specifications are translated, we allow for other weavers to be built for KALA, which can work through other means, such as e.g. byte-code manipulations. In the extreme, given a powerful enough general-purpose aspect weaver, it is even possible to write a library that implements KALA support for that weaver.

1.6 Overview of the Dissertation

In the next chapter we treat transaction management and advanced transaction management. We start with an introduction to the concepts of transaction management before talking about advanced transaction management. We detail four representative ATMS in this chapter: Nested Transactions, Sagas, Relatively Consistent Schedules and Split Transactions.

After detailing different models for advanced transactions, chapter three considers the complexity of demarcation code for these models. In this chapter we provide a brief introduction to separation of concerns and Aspect-Oriented Programming. As AOP has already been used to modularize demarcation code for classical transactions, we discuss this work here. We end chapter three by considering the need for a good separation of concerns for ATMS demarcation code.

Chapter four analyzes the concern of advanced transaction management, and finds that this concern itself is composed of multiple sub-concerns. We then consider the location of the code, within the demarcation code, treating these sub-concerns. We see

that, in such demarcation code, the code for the different sub-concerns is itself tangled code. Considering existing, popular aspect technology, we observe that these tools do not allow these sub-concerns themselves to be modularized, and require aspect code to be written that itself is tangled.

In chapter five, we change the perspective from the application using an ATMS to the middleware and TP Monitor which provide the support for running the application. We first give a brief introduction to multi-tier distributed systems and the role of TP Monitors in these systems. We then consider one standard architecture for such systems in more detail: Enterprise JavaBeans (EJB). Of EJB we give a general overview before discussing how transaction management is performed in this architecture. We end this chapter by giving a number of critiques on transaction management in EJB, with regard to separation of concerns and support for a wide variety of ATMS.

To be able to support a wide variety of ATMS, we use an existing formal model as a basis of our TP Monitor and aspect language. Chapter six details this formal model, named ACTA. We first describe this model in detail before giving a number of formal specifications of transaction models using ACTA. We end this chapter by considering how we can go from this formal model to an implementation.

Chapter seven starts with giving an overview of our TP Monitor based on the ACTA formal model, called ATPMos. We then proceed with showing the difficulty of writing demarcation code both for classical transaction management and for ATMS. This is performed by means of an example application. This example confirms the observations of chapter three and four with regard to the complexity of demarcation code and tangling of different sub-concerns in this code.

In chapter eight we introduce KALA: the aspect language we created that provides for a good modularization of the ATMS concern. We first detail the language, giving some example KALA specifications. Second we outline how our aspect weaver weaves KALA specifications into Java code.

After having detailed KALA and our weaver, chapter nine discusses the modularization features of KALA. We show how KALA allows the different sub-concerns of an ATMS to be specified in different modules, and how these modules are combined to form a complete specification. We show this by defining KALA code for Sagas in different modules, and giving the equivalent composed code. Furthermore, we give modular descriptions of the Nested Transactions and Relatively Consistent Schedules ATMS, and show how we can use KALA to build a new ATMS by changing the implementation of some modules.

Chapter ten raises the level of abstraction of KALA code by defining a family of domain-specific languages, one DSL for each ATMS treated in chapter nine. For each of these ATMS we show the DSL we have built, detailing why this achieves concise code. Furthermore, we show how the family of DSLs achieves reuse of the language concepts common to multiple models.

Having discussed our approach, chapter eleven illustrates how we ease the usage of an ATMS of which the design best fits the application being built. We give the implementation of three example applications, both the Java code for the core concern and the code in the DSL which specifies the ATMS concern.

The final chapter presents our conclusions and suggests some topics for further research.

Chapter 2

Advanced Transaction Management

'Alright,' said Deep Thought. 'The Answer to the Great Question...'
'Yes...!'
'Of Life, the Universe and Everything...' said Deep Thought.
'Yes...!'
'Is ...' said Deep Thought, and paused.'
'Yes...!'
'Is ...'
'Yes...!!! ...?"
'Forty-two,' said Deep Thought, with infinite majesty and calm.
— **Douglas Adams, unexpected results of a long transaction in "The Hitch Hiker's Guide to the Galaxy"**

Transactions have been around for a long time. In fact, the oldest known example of human writing is a record of a transaction: it is a Sumerian tablet from 3300BC, that records payment of barley, wheat, sheep, and of beer. This tablet is used as an example by [GR93] : When a farmer came to pay his duties, a clerk would chisel this fact onto a clay tablet. Going forward to the present, we see transactions primarily being used in book-keeping operations: A physical operation has to be accounted for in the books, usually by amending the books at different places, e.g. a move of stock between inventories entails decreasing the amount on one page, and increasing it in another page. Within computer science, the books have been replaced by a database, and so we see the concept of transactions firstly in the domain of databases. The accountant working with the books has been replaced by an application, and this application will request the database to perform some work, i.e. reads of and writes to the data, on its behalf.

In this thesis we address how the application performs such requests to the database for transaction management and also for advanced forms of transaction management. Therefore, in this chapter, we provide an introduction to both forms of transaction management. Given the context of this work, multi-tier distributed systems, we will only focus on one kind of advanced transaction management, that is when using a single database, as this is common for these applications. Advanced forms of transaction management for other application domains have been developed, but we will only briefly touch on them here.

We start this chapter with an introduction to concepts of transaction management, in which we go into detail of how transactions are scheduled using locks, the problem of deadlocks and transaction rollbacks, and the use of savepoints. We then proceed with advanced transaction management for single-database systems, detailing three such models, each with its own specific purpose: Sagas, Relatively Consistent Schedules and Split Transactions.

2.1 Concepts of Transaction Management

As said above, in computer science, books and accountants have been replaced by a database and an application. The interaction between application and database needs to be formalized further, because until now we have assumed a number of properties of transactions that do not necessarily hold in this case: When noting down a transaction, an accountant will not interrupt this work until all of the required pages have been amended, because until the work is finished, the books will not correctly reflect reality. Also, because of this inconsistency, other accountants are not allowed to use these intermediate figures for their own work.

Formalizing this, we come to the definition of a transaction as a protected form of program execution that has the following properties (Also known as the *ACID* properties)[GR93]:

Atomic: A transaction is an atomic sequence of actions.

Consistent: A transaction takes the database from one consistent state to another consistent state.

Isolated: The intermediate effects of a running transaction are isolated from the other transactions.

Durable: The results of a transaction are kept in persistent storage.

When writing an application that uses transactions the transaction programmer writes *demarcation code* that marks the start and end of each transaction in the

application code and that associates each database access with a running transaction [GR93]. At run-time the *transaction scheduler* software will mediate concurrent accesses to the database by the different transactions within the application. The sequences of interleaved database accesses and transaction starts and ends are known as schedules. To maintain the ACID properties, the scheduler will ensure the schedules are *serially equivalent*: the interleaved operations of different transactions running in parallel are guaranteed to produce the same results as if the transactions would run in sequence. In other words, this allows multiple concurrent threads of the same application, or multiple concurrent applications, to work on shared data while ensuring that this data remains consistent.

Having introduced transactions, the remainder of this section discusses some basic concepts regarding scheduling of transactions and how this is achieved through locking. Subsequently, we show the problem of deadlocks and lastly we talk about aborting transactions.

2.1.1 Schedules and Serializability

It is well-known that while a single executing transaction will preserve the consistency of the database, multiple transactions running in parallel need to be subjected to some concurrency control to ensure that the database is kept consistent [CDK94, GMDW00]. This section introduces the basic concepts of schedules and serializability, which is used later on in the text.

The concurrency control mechanism will consider the relevant operations of the transactions and the sequence in which these are executed. This time-ordered sequence of important transaction actions is known as a *schedule*. Consider the following example schedules $S1$ and $S2$ for two transactions $T1$ and $T2$:

$$S1 = R1(x)W1(x)R1(y)W1(y)E1R2(x)W2(x)R2(y)W2(y)E2$$

$$S2 = R2(x)W2(x)R2(y)W2(y)E2R1(x)W1(x)R1(y)W1(y)E1$$

Where $Ri(a)$ indicates reading object a by Ti, $Wi(a)$ writing object a by Ti, and Ei the end of Ti by a commit or an abort. These two example schedules are called *serial schedules* [GMDW00]: they execute both transactions in series. It is clear that a serial schedule preserves database consistency, since it does not allow any concurrently executing transactions.

As stated above, the goal of the concurrency mechanism of the transaction manager is to ensure consistency of the database. This is ensured by controlling the schedule of the transactions: the schedule which results from all transactions being executed concurrently must leave the database in the same state as if the schedule was serial. Consider, for example the following schedules:

$$S3 = R1(x)W1(x)R2(z)W2(z)R1(y)W1(y)E1R2(y)W2(y)E2$$

$$S4 = R1(x)W1(x)R1(y)W1(y)E1R2(z)W2(z)R2(y)W2(y)E2$$

$S3$ is not a serial schedule, since $T2$ starts before $T1$ has ended. However, $S3$ is *equivalent* to $S4$, as both leave the database in the same state. Such non-serial schedules that are equivalent to a serial schedule are called *serializable*[GMDW00]. Creating serializable schedules is the key task of the concurrency manager: by enforcing that all schedules are serializable, database consistency will be ensured.

Note that in the schedules we do not consider the values read and written from the database. Consider the following schedule, where we know that $x = 10$ and $y = 30$, $T1$ simply increases values by 100 before writing them back, and $T2$ doubles the values before writing them back.

$$S5 = R1(x)W1(x)R2(x)W2(x)R2(y)W2(y)E2R1(y)W1(y)E1$$

$S5$ is not serial, and is not serializable as it does not preserve database consistency: at the end of $S5$ $x = 220$ and $y = 160$, while for the serial schedules $S1$ and $S2$, we respectively have $x = 220$, $y = 260$ and $x = 120$, $y = 160$. However, consider what happens if the semantics of $T2$ were different. Let $T2$ write back the exact values it has read, without any modification. Then the results of $S1$, $S2$ and $S5$ would be $x = 110$ and $y = 130$. In other words $S5$ would indeed be serializable.

The above shows us that the semantics of the transactions being scheduled play a role in whether a schedule is serializable or not. Unfortunately, the concurrency manager is not made aware of the semantics of the transactions and therefore must always assume the worst possible scenario: If a write to the database, within a transaction, may temporarily render the database inconsistent (until another write from the same transaction fixes this inconsistency), we must assume that it will, indeed, introduce such an inconsistency [GMDW00].

Note that this implies that the values of data being read or written are of no relevance to the concurrency manager, which is why our schedule notation does not include these values.

We have now introduced the concepts of schedules and serializability. However, commercial systems do not guarantee serializability as described above, instead a stronger condition is ensured, which is discussed in the following section.

2.1.2 Conflict-Serializability and Locking

Ensuring serializability is a non-trivial task, which is why most commercial systems ensure a more conservative condition, called *conflict-serializability* [GMDW00] through the use of locks. This section gives an introduction to these mechanisms.

As can be inferred from the name, conflict-serializability uses the concept of a *conflict*: a pair of consecutive actions in a schedule is in conflict if, when their order is changed, the behavior of at least one of the transactions can change i.e, the database state can change.

There are three pairs of actions in a schedule that conflict [GMDW00]:

1. Two actions of the same transaction conflict because the sequence of actions within a transaction may not be modified.

2. Two writes of the same database element by two different transactions conflict. In other words, $Wi(x)$ and $Wj(x)$ conflict (for $i \neq j$). This is because the contents of the database will contain the last of the two writes (in this case the value of Tj). So if we change the order, the database will contain another value (in this case the value of Tj), which may be different.

3. A read and a write of the same database element by different transactions conflict. Put differently, $Ri(x)$ and $Wj(x)$ conflict, as do $Wi(x)$ and $Rj(x)$ (for $i \neq j$). Swapping $Ri(x)$ and $Wj(x)$ changes the value of x read by Ti, which might result in a difference in the actions of Ti somewhere down the line. Similarly, swapping $Wi(x)$ and $Rj(x)$ affects the value of x as read by Tj, which might change the database state.

We may now take any schedule Si and try to make it serial by swapping operations that do not conflict. If we succeed, Si is equivalent to the serial schedule because we have not modified the database state through the use of non-conflicting swaps. In that case, Si is said to be *conflict-serializable* [GMDW00].

Such a-posteriori determination of the (conflict-) serializability of a schedule, however, is not an optimal way to perform concurrency control. The concurrency control mechanism is much more efficient if, while the transactions are running, the schedule is constructed such that it will be serializable when all transactions are finished. Enforcing serializability in this manner is usually performed through the use of a *locking scheduler* [GMDW00].

Such a scheduler requires that transactions correctly use locks: A transaction can only read or write an element if it has first obtained the corresponding lock, and the transaction must at some point unlock the elements it has locked. The locking scheduler will then ensure that no two transactions Ti and Tj have concurrently locked the same element x. This is achieved by suspending a transaction Tj that asks for a lock on x if x is already locked by Ti. Tj will be resumed, with the lock obtained, when Ti releases the lock on x.

Most, if not all, commercial locking systems place an extra condition on locking: all lock requests must precede all unlock requests, i.e. once an element is unlocked, no

further locks may be requested. This condition is called *two-phase locking*; in the first phase locks are obtained, and in the second phase locks are released [GMDW00].

It has been proven that schedules of transactions that obey the two-phase locking condition are conflict-serializable. We do not give this proof here, as it is not relevant for this thesis, instead we refer to [GMDW00]. Intuitively, it is said that each transaction is said to execute the instant it issues its first unlock request, and the equivalent serial schedule would be the one where the transactions are ordered in the same order as their first unlocks.

To summarize: in most commercial systems serializability is enforced through the use of a locking scheduler and the requirement of two-phase locking. If two-phase locking is satisfied, the schedule constructed by the scheduler will be conflict-serializable, which is a more conservative condition than serializability, and therefore the schedule will be serializable.

2.1.3 Deadlocks

There is one major drawback to locking, which is deadlocks. Consider the following interleaving of actions of two transactions Ti and Tj:

$$Wi(x)Wj(y)Wj(x)Wi(y)$$

$Wj(x)$ will cause Tj to wait for Ti to release its lock on x, and $Wi(y)$ will cause Ti to wait for Tj to release its lock on y. Both transactions are *deadlocked* and will wait for each other forever, if no action is taken by the concurrency manager.

It is generally impossible to recover from deadlock situations without aborting at least one transaction [GMDW00], which we discuss in detail below. By aborting a transaction, its locks will be released, which will allow other waiting transactions to acquire the locks for which they have been waiting, breaking the deadlock.

The simplest way to detect a deadlock is by using a timeout: assume that transactions complete within a given timeframe t. If a transaction Ti has been waiting on another transaction Tj for more that t, this means that Tj is involved in a deadlock, and Tj should be aborted.

It is clear that this approach has an important drawback in case of a flexible transaction duration. When some transactions complete quickly, while other transactions may take a very long time, it is not efficient to set t to the longest possible timeframe. If a quick transaction is deadlocked, it will take a very long time to break that deadlock, and the performance of the overall system will be degraded significantly.

In such cases it is better to perform active deadlock detection using a *wait-for graph* [GMDW00]. In such a system, a graph is constructed while transactions are executing. Every running transaction is a node, and a directed arc is placed between

two nodes if the first node waits for the second. Every time an arc is placed, the graph is checked for cycles. If a cycle occurs, this means that every transaction in the cycle is waiting for another transaction within that cycle, i.e. we have a deadlock.

Now the question remains what to do with the transaction that has been aborted. In most cases, an exception will be passed to the application performing the transaction. The burden is placed on the application to perform the needed actions to recover from this error condition. Usually the application will either restart the transaction, or report an error to the user.

2.1.4 Transaction Rollbacks and Savepoints

We have just seen that there is an important reason why we must be able to abort a transaction: to break deadlocks. However, this is not the only reason. Consider the following, typical, scenario: the system under consideration consists of a database, running on a database server, and a user application running on a different server. In this scenario, we must consider what to do with the running transaction when one of the servers fail when a transaction is in progress.

If the user application fails, the database cannot keep the transaction open, waiting indefinitely for the user application to pick up where it left off. Indeed, there is no guarantee that this will ever happen. Therefore, the transaction has to be terminated, and the database state will be reverted to the state before the beginning of the transaction. This operation is known as a *transaction abort* or a *transaction rollback* [GR93].

Database software, however is expected to be able to recover from failures and usually keeps a log of all activities on the database. If the database server fails, upon restart the database software will enter a recovery phase, where it will use this log to recover the database to a known good state [GR93]. Note that this does not necessarily include modifications to the database by all running transactions, in effect, transactions will be rolled back, and the user application will be informed of this fact when it attempts to proceed with these transactions.

This, again, shows us that we need to be able to cleanly abort the work of a running transaction, effectively pretending that the transaction never happened. However, to allow this in our locking scheduler, some extra requirements are needed, as we show below:

Consider the following schedule, where we have included unlock actions $Ui(x)$ for x by transaction Ti, and transaction $T1$ aborts at $A1$.

$$S6 = R1(x)W1(x)U1(x)R2(x)U2(x)E2A1$$

In this schedule, $T2$ has read a value for x, as written by $T1$. But, at the end of the schedule, $T1$ aborted, and therefore $T2$ should not have seen this value, as aborting

$T1$ implies that the value never existed. Such situations are known as a *dirty read* [CDK94], and must be avoided by placing extra restrictions on the possible schedule. These restrictions are required because, in $S6$ we can not simply abort $T2$ to avoid the dirty read when $T1$ aborts, since $T2$ has already committed.

A possible solution would be to delay commitment in case of a possible dirty read: $T2$ has read x, which may become a dirty read if $T1$ aborts. Therefore, we delay the possible commitment of $T2$, until $T1$ has finished executing. If $T1$ commits, $T2$ may commit, if $T1$ aborts, $T2$ has to abort to avoid the dirty write.

This introduces a new issue that has to be dealt with: *cascading aborts* [CDK94]. Aborting a transaction Ti may imply that a number of other transactions $Tj, Tk \ldots$, which have been waiting for Ti to commit have to be aborted. This, in turn, may imply that yet other waiting transactions have to be aborted, and so on ... It is clear that this is not a desirable property, as it may significantly impact the running transactions by forcing a large number to abort. Therefore it is best to avoid cascading aborts by delaying reads: we only allow a read of x in Tj when Ti has already committed (or aborted, as may be the case).

Delaying reads is one half of the requirement of *strict execution of transactions* [CDK94]: transactions should delay both reads and writes of a data item, until other transactions that wrote that item have ended[1].

We can achieve strict execution of transactions in the locking scheduler by defining a simple unlock rule: Transactions may not unlock their data items. Instead, the scheduler will perform unlocking of these resources immediately after a transaction has ended. This ensures that other transactions will not suffer from dirty reads, and avoids cascading aborts. This form of locking is known as *strict two-phase locking* [CDK94].

Having the ability to cleanly undo the work of a running transaction is, of course, also useful from an applications' point of view. At any given time the application can decide that the currently running transaction has become invalid in some way, and simply abort it. The work performed in this transaction will be cleanly undone, without the application needing to intervene.

Considering this undo behavior, it would also be handy if we could not undo the entire transaction, but undo the work up to a certain point. This is possible when using *savepoints*: at any time within a transaction a savepoint can be placed, which is usually identified by a monotonically rising number [GR93]. When the application now decides to abort, it can rollback to a given savepoint. This results in the transaction being returned to the state it was in at that savepoint, and ready to continue.

[1] We do not discuss the need for delayed writing here, as this may be database-specific, as noted in [CDK94]. We include delayed writing here for completeness of the definition.

Conclusion

In this section we have given a general introduction on the concepts of transactions and transaction management. We have shown how serial execution of concurrent transactions can be achieved using the concept of serializability, and seen the most frequent implementation: conflict-serializability through the use of locks. We have also shown the major drawback of using locks: deadlocks. To break deadlocks, we need to be able to abort transactions, so we have discussed how transactions can be safely aborted, while avoiding cascading aborts. This feature is also explicitly made available to the application, so that it may itself decide to abort a running transaction, based on an internal heuristic.

2.2 Toward Advanced Transaction Management

Classical transactions, as described above, have been developed to treat small units of work that only access a few data items. As a result, as transaction time grows, and the number of data items accesses becomes larger, the performance of the system will drop significantly [GR93]. This is due to a number of factors, for example the increased chance of deadlocks associated with longer transactions, and the large discrepancy between complex applications and their data requirements and the simple functionality of transactions.

To increase application performance, and to address additional requirements such as achieving a more fine-grained rollback structure, and allowing cooperation between transactions, a number of *advanced transaction mechanisms* (ATMS) have been developed. Of these, the most well-known ATMS is *Nested Transactions* [Mos81]:

Briefly put, Nested Transactions allow for hierarchically structured transactions, and include rules for nesting the scope of commitment and recovery. More extensively:

- A nested transaction is a tree of transactions, where nodes are nested transactions, and leaves are conventional, flat, transactions.
- A child transaction of a nested transaction is called a subtransaction, the root transaction is called the top-level transaction.
- The rollback of a transaction in the tree causes all its subtransactions to rollback, recursively down the tree.
- While rollbacks of subtransactions take effect immediately, a commit will only take place when its parent transaction commits. Note that this rule is recursive, therefore subtransactions only commit when the top-level transaction commits.

- All changes to the database performed by a subtransaction become visible to its parent when the subtransaction commits. Concurrently executing siblings of the sub-transaction, however, cannot see these changes.

When the application being built has a tree-like execution structure, Nested Transactions can be used to mimic that structure. In these cases, the scope of a rollback is fine-tuned to the structure of the application, which prevents unnecessary loss of work. Gray and Reuter [GR93] comment that there is a strong relationship between Nested Transactions and the modularization of software, but find few attempts to capitalize on this relationship.

While Nested Transactions is the most well-known ATMS, an impressive number of alternate ATMS can be found in the literature. As example ATMS, we cite the following: Sagas [GMS87], Relatively Consistent Schedules [AFTO89], Split Transactions [PKH88], Altruistic Locking [SGM94], Multilevel Transactions [WS92] and Open Nested Transactions [MRKN92]. Between 1997 and now no further fundamental research has been performed in this area. We note, however, that the existing ATMS recently have been applied in the context of multi-tier distributed systems, more specifically as a TP Monitor for a select number of ATMS [Pro01] and a specific implementation in the domain of web services [KS03].

In general, an ATMS will extend the single classical transaction to either form a combination of transactions, or to establish relationships between different transactions, which is why ATMS are also often called *extended transaction models*. An example of the former are what we call *secondary transactions*: multiple ATMS require at certain points, conceptually separate from the main control flow of the application, secondary transactions to run automatically when some constraints are satisfied.

Each ATMS usually focuses on one single issue of classical transactions, and no overall system has been developed that treats a large number of the identified drawbacks of classical transactions. We discuss a small selection of ATMS next, showing the different drawbacks of classical transactions they address. We restrict the choice to well-known (within the community) and representative ATMS for classical transactional, single-database applications, as this is the most common usage of a database in multi-tier distributed applications.

We note that, ATMS, however, are not restricted to this domain only; a significant amount of research has been performed in the area of multi-database systems and workflow management systems. A well-known example ATMS for multi-database systems is Flex Transactions [ELLR90, KPE92], and also well-known, for workflow management is the ConTracts model [WR92]. As implied above, we do not discuss these models, as they lie outside the scope of this dissertation.

In the remainder of this chapter, we detail three ATMS for single-database applications: Sagas, Relatively Consistent Schedules, and Split Transactions.

2.3 Sagas

The first ATMS we detail, after Nested Transactions, is one of the oldest, best-known and widely referenced ATMS within the community: *Sagas* [GMS87]. The Sagas ATMS was introduced as an alternative to Long Lived Transactions (LLT's). LLT's are transactions that last significantly longer than usual, such as transactions that process a large amount of data or that wait for user input at some point.

These transactions can cause a significant drop in performance due to their long life cycle and the large number of database objects they access. It has been claimed that the probability of deadlocks increases with the fourth power of transaction size [WR92], which makes LLT's extremely likely to incur deadlocks. Also, concurrently executing transactions are not able to access these objects until the LLT terminates.

In some cases this problem can be addressed by relaxing the atomicity requirement of LLT's. A Saga SG is a LLT that has been split into a sequence of sub-transactions $T1, T2, \ldots, Tn$ by the application programmer. Each sub-transaction Ti is a fully ACID transaction, and the sequence of transactions should either be executed completely or not at all. Splitting the LLT releases locks earlier, which increases concurrency as other transactions can execute concurrently with a Ti, and decreases the probability of deadlocks, since after each Ti all locks are released and each Ti will probably require less resources than SG.

Conceptually, Sagas can be used if a LLT consists of a sequence of independent operations where each step does not necessarily need to see the database in the same state. As an example, consider a banking application: at the end of the year, we have to calculate and add interest to all accounts. Instead of doing this all in one LLT, which would eventually try to obtain locks on all accounts of the bank, we can use a Saga. One sub-transaction in the saga would simply calculate the interest for one account, and add it to that account. The Saga then sequentially performs this subtransaction for all accounts, only locking one account at a time.

Rollback of a Saga

We need to be able to handle roll-backs of the Saga, which implies that the sequence of sub-transactions is not executed fully. In these cases, compensating actions must be executed to undo the effects of already committed sub-transactions. To allow this, the programmer defines a *compensating transaction* Ci for each sub-transaction Ti. This transaction performs a semantical compensation action, i.e. it undoes the work of Ti. Note that the compensating action is not equal to reverting the affected data to the state before Ti because in the mean time other, concurrent and unrelated, operations may have occurred on the data. For example, consider a bank application in which a transaction Tt transfers 100 EUR from account A to B. The compensating transaction

Ct would be to transfer 100 EUR back from B to A, as opposed to reverting the balance of both accounts to the original amount. We do not revert to the previous balances because in the mean time other deposit and withdraw operations may have occurred on both accounts.

To roll back a Saga, the concurrency manager aborts the currently running sub-transaction Ti, and subsequently runs the compensating transactions in reverse order $(Ci-1, Ci-2, \ldots, C1)$.

Note that compensating transactions are therefore an instance of secondary transactions. They are not contained within the control flow of the application, but are run by the concurrency manager. Each compensating transaction is constrained to run at rollback only, and may only run at the correct point in the sequence.

Because aborting a Saga must always succeed, execution of compensating transactions may never fail. To ensure this, the programmer may need to define preconditions PRi for each compensating transaction Ci. In our bank example, a transfer could have the requirement that the source account contains at least the amount being transferred, so the precondition for Ct would be that account B contains at least 100 EUR.

The transaction manager must now ensure that preconditions PRi are held whenever the sub-transaction Ti completes, until either the Saga commits, or the compensating transaction Ci commits. This could be done, for example, by delaying writes to data that would cause the precondition to be violated. Ensuring these preconditions guarantees that all compensating transactions will succeed, i.e. that the saga can be successfully rolled back.

Note that in case of a Saga rollback, concurrently running transactions and Sagas may have seen and used the partial results of this Saga. We can not ensure neither that these concurrent transactions are notified of this change nor that they are aborted, because they may have already committed.

To summarize, with Sagas a long-lived transaction is split up by the programmer into a sequence of sub-transactions that are executed in sequence. To allow rollback of the saga for each sub-transaction, except for the last, a compensating transaction, i.e. a semantic undo action is defined by the programmer. When the saga is rolled back, the currently executing sub-transaction is rollbacked, and the compensating transactions for the already committed sub-transactions are executed, in the reverse order of the sub-transactions.

This ends our discussion on Sagas, and we now continue with detailing an example ATMS which makes extensive use of semantic information to allow for more concurrency between different transactions.

2.4 Relatively Consistent Schedules

In traditional transaction managers, database consistency is ensured by conservatively restricting operations on the data. We can however relax these restrictions, taking the semantics of the transactions into account, and still semantically preserve consistency of the database, as we have illustrated in 2.1.1. This allows a larger number of possible semantically consistent schedules of transactions at a given time, which increases concurrency.

A number of ATMS have been published which use a form of semantically consistent schedules, and we detail one example ATMS as an illustration. We have chosen the *Relatively Consistent Schedules* (RCS) ATMS [AFTO89] as an example because it is one of the most flexible ATMS using semantic consistency and as it shows the complexity of some of the published ATMS.

Using RCS, the programmer instructs the scheduler how a transaction may be interleaved with other transactions, selectively decreasing isolation between the different transactions. This has a first advantage that schedules can easily be verified a priori. The second advantage is the generally conservative nature of interleavings which eases management: an interleaving will only take place if it is specifically permitted. This allows the programmer to selectively focus on groups of transactions to be optimized. [AFTO89]

We note that in [GM83] the question is addressed if programmers should be burdened with the extra work of providing this semantical information. The authors consider that this extra work should be avoided if possible, but that "in some applications the cost of serializable schedules is unacceptably high, so there really is no choice" [GM83]. Furthermore, the authors claim that in current applications such semantic knowledge is already implicitly embedded anyway by the programmer. In other words, some semantic information is used implicitly, and that it is better to make this information explicit through the use of an ATMS, such as RCS.

Before we detail RCS, however, we need to define the concept of sensitive transactions and semantically consistent schedules. *Sensitive transactions* are transactions that need to observe a consistent state of the database. Consider, for example, a transaction that displays output to the users. While certain database inconsistencies might be allowed using semantically consistent schedules, the output data of sensitive transactions must appear as if it is based on a consistent database. A *semantically consistent schedule* is then defined as a schedule that transforms the database from a consistent state to a consistent state, while all sensitive transactions within the schedule obtain a consistent view of the database.

To obtain such a semantically consistent schedule, in RCS a transaction is subdivided in different steps, and each step may be interleaved with steps of other transactions. To use this ATMS, the programmer performs three extra operations when

defining the transactions: first splitting the transaction into steps, second and third defining the semantic types and compatibility sets, and fourth associating possible interleavings with each step. We now discuss these steps in more detail.

Splitting a Transaction Into Steps

The programmer first declares the steps of each transaction. To allow this, we represent a transaction Ti as a sequence of steps Si_j and a termination command Ei. Each step consists of a sequence of atomic actions (either a read $Ri(x)$ or a write $Wi(x)$) and and a breakpoint Bi_j. The breakpoints signal the end of each step, the termination command signals the end of the transaction.

Consider the following example transaction:

$$T1 = R1(x)W1(y)B1_1R1(z)B1_2W1(x)W1(z)B1_3E1$$

$T1$ contains three steps: $R1(x)W1(y)B1_1$, $R1(z)B1_2$ and $W1(x)W1(z)B1_3$.

Defining Semantic Types

Second, *semantic types* [GM83, AFTO89] are defined by the programmer. He uses his semantic knowledge to divide the transactions into groups of the same semantic type, based on the actions they perform and the data they use. For all semantic types he considers the atomic steps of the transactions (reads or writes on the database), and determines if the steps of a transaction of one type, TX, may be executed simultaneously with the steps of transactions of another type TY. For example, take a customer report transaction Tb that will report the balance of the checking account and the savings account for that customer. If we assume that a withdraw Tw or deposit Td operation will never access both the checking and the saving accounts of the same customer, we can simultaneously execute Tb and Tw or Tb and Td. To allow this, let Tb be of type TB and Tw and Td be of type TT (where the second T stands for Transfer). Now we must allow simultaneous execution of TB and TT.

Simultaneous execution differs from normal concurrent execution in that while concurrently executing transactions are isolated, simultaneously executing transactions are not. When two transactions are executing simultaneously, they share the data on which they operate and may therefore see intermediate results from the other transaction.

Declaring Compatibility Sets

Not all transactions can be executed simultaneously, simultaneous execution is only possible for transactions which are declared to be compatible. To declare this the

programmer will define *compatibility sets* CS for each type [GM83, AFTO89]. Each element of a compatibility set is a set of types called an *interleaving descriptor set*. Interleaving descriptor sets contain transaction types. For the compatibility set CS_{TX} of the type TX, each interleaving descriptor set defines the types of transactions with which transactions of type TX may be interleaved.

For our bank transfer example, where transactions of type TB can be executed simultaneously with transactions of type TT, we would write $CS_{TB} = \{\{TT\}\}$. Conversely, in this example, transactions of type TT can be executed simultaneously with transactions of type TB, so $CS_{TT} = \{\{TB\}\}$.

A more elaborate example, $CS_{TX} = \{\{TY\}, \{TY, TZ\}\}$ consists of two descriptor sets. The first descriptor set indicates that transactions of type TX may be executed simultaneously with transactions of type TY. The second descriptor set indicates that that transactions of type TX may be simultaneously executed with transactions of type TY and TZ. Since CS_{TX} does not include $\{TX\}$, transactions of type TX may not be executed simultaneously with transactions of the same type. Also note that a transaction Ti of type TX may not be simultaneously executed with a transaction Tj of type TZ, unless Tj is simultaneously being executed with a transaction of type TY.

Declaring Interleaving

After defining compatibility sets the programmer associates such sets with each breakpoint of a transaction. This allows transactions of the given types to interleave at this breakpoint in the transaction, yielding a simultaneous execution of these different transactions. Note that an interleaving transaction Tj of Ti may itself be interleaved by other transactions at any breakpoint of Tj where this is allowed by the corresponding interleaving descriptor set.

For example, considering the bank transfer example above, with a report transaction Tb of type TB and a withdraw transaction Tw of type TT. If at a certain breakpoint of Tb the compatibility set $CS_{TB} = \{\{TT\}\}$ is defined, Tw may interleave, i.e. perform (a part of) the withdrawal operation while having access to the data which Tb is using to calculate a balance. Note that we can nest interleavings: if Tw itself also contains breakpoints, with their associated compatibility sets, other transactions (the types of which are included in these sets) can interleave at these points.

The act of splitting each transaction into different atomic steps, and allowing interleavings only at some points, depending on types, significantly modifies the atomicity of the transaction. Transactions of a given type TX will see a transaction Ti as either one atomic whole (if no interleavings are allowed with TX), or as a number of atomic steps (if interleavings are allowed with TX). Transactions of a different type

TY will have a different view on the atomicity of Ti. Having different possible views on the atomicity of a transaction is known as relative atomicity and schedules of these transactions are called relatively consistent schedules [AFTO89].

We must however still consider what should happen when a transaction is aborted. The problem here is that all transactions that are simultaneously executing with the aborted transaction will now work on inconsistent data. This requires us to abort all these simultaneously running transactions, which can lead to cascading aborts and a large performance hit. The paper [AFTO89] does not explicitly address this issue beyond recommending the use of counter-steps. In other words, we could use compensating transactions, to semantically undo the work of the aborted transaction without having to abort the other transactions.

To summarize, the relatively consistent schedules ATMS preserves semantic consistency of the database, while allowing specific transaction types to work simultaneously on the data, i.e. sharing this data as if they were one and the same transaction. Transactions are assigned a type and subdivided in different steps, where each step may run simultaneously with steps of other transactions, as specified by the programmer. Rollback of such transactions is not explicitly addressed in the paper, save for the suggestion of using compensating transactions, as in Sagas.

This concludes our discussion on an example ATMS specifically relying on explicit semantic information about the transactions it is managing. The last ATMS we discuss illustrates a different feature, which is delegation of resources.

2.5 Split Transactions

The last extension to the standard transaction model we discuss is *split transactions* [PKH88]. This model features, amongst others, the ability for a transaction to gather a subset of its resources and pass responsibility for them to another transaction. We call this feature *delegation of resources*. A number of ATMS have been published which use delegation of resources, We chose to use Split transactions as an illustration of delegation of resources as it also shows how a specific class of applications can influence the design of a transaction model.

Split transactions has been designed to deal with *open-ended activities* such as CAD/CAM projects and CASE tools. As described in [PKH88], open-ended activities are characterized by an uncertain duration (from hours to months), uncertain developments that can not be foreseen at the beginning, and interaction with other concurrent activities.

Typically, data access by these open-ended activities is driven by users. This is in contrast to traditional database accesses through transactions, which are known as the application is written. Here we have dynamic transaction construction: a transaction

is made at run-time, as a result of actions of the user.

These characteristics have a significant impact on how transactions can behave. First, due to the uncertain duration the transactions may become LLT's, with all the performance problems this implies, as we discussed in 2.3. Second, uncertain developments require that activities that can be made independently need to be kept in one transaction. For example: an initial activity may modify data early in the transaction, and this data may not be needed further on, and will remain untouched until the transaction is committed. Conversely, an activity at the end of the transaction may be independent of previous activities, which are not committed until the entire transaction has ended.

Splitting Transactions

Split transactions were created to address differing behavior due to open-endedness: a split transaction divides an ongoing transaction Ti into two fully independent, serializable, transactions Tj and Tk. The database objects accessed by Ti are divided amongst Tj and Tk into two or more sets. Split-transactions are mainly used to commit a number of objects accessed by Ti, by assigning them all to one transaction Tj, and committing that transaction.

Note that we also have the inverse operation of split-transaction: *join-transaction*. A join-transaction combines two transactions Ti and Tj together by finishing the transaction Ti and making its results part of the joined transaction Tj. However, we restrict our discussion only to split-transactions, as they are said to be more widely applicable [PKH88].

Consider the split of a transaction Ti into Tj and Tk; we can conceptually divide the objects accessed by Ti into four sets:

- $JReadSet$: the objects, read by Ti that will belong to Tj
- $JWriteSet$: the objects, written by Ti that will belong to Tj
- $KReadSet$: the objects, read by Ti that will belong to Tk
- $KWriteSet$: the objects written by Ti that will belong to Tk

To allow Tj to be serialized before Tk, we must consider the results of the following three conceptual intersections of the above sets:

1. $JWriteSet \cap KWriteSet := KWriteLast$
2. $JReadSet \cap KWriteSet = \phi$
3. $JWriteSet \cap KReadSet := ShareSet$

The first intersection defines a set $KWriteLast$, which are the objects that are updated last by Tk: updates of Tj to elements in this set will be overwritten by Tk. The second intersection being empty guarantees that Tj does not see any values written by Tk. The third intersection defines a set $ShareSet$ that contains updated objects of Tj that are seen by Tk, and that will not be updated by Tj after the split-transaction. If these properties hold we can serialize Tj before Tk, and therefore, because the original transaction Ti was serializable, we can serialize Tj and Tk with respect to all the other running transactions.

Two classes of split transactions are defined through the contents of $KWriteLast$ and $ShareSet$: if both sets are empty, the split transaction is called independent, if not the split transaction is called serial. In the independent case there are no object access conflicts between Tj and Tk, i.e. Tj and Tk are independent, therefore they can be serialized in any order. In the serial case, Tk needs to follow Tj because of data access dependencies. To avoid Tk using uncommitted data of Tj, Tj may not write to objects in the $ShareSet$. Unlike in the independent case, if Tj aborts, Tk must also abort, because it relies on data that was written by Tj.

Note that in [PKH88] it is not addressed what should happen if at some point the original transaction Ti, which conceptually still lives on in Tj and Tk, needs to be rolled back. If either Tj or Tk have already been committed, this rollback will, in a strict sense, not be possible. This problem might be alleviated by using a compensating transaction, as described earlier. However, this will not undo the possibility that other running (and committed) transactions have used the results by the transaction that is being compensated, as we already have mentioned in 2.3. Therefore rollbacks should be treated with caution.

Furthermore, due to the nature of the transactions, it is not straightforward to define compensating transactions. We can not simply define a compensating transaction for each split transaction, because there are a significant amount of possible actions for each split transaction, which can not be determined at priori. We can say that Split transactions virtually require the ability to dynamically form a compensating transaction for each running transaction, which is non-trivial.

Using Split Transactions

We can use Split transactions to increase transaction processing performance when dealing with LLT's, by splitting the LLT at opportune moments, determined by the user, the application. or the concurrency manager. For example, this would be the case when it has been determined that a subset of the database objects accessed by a LLT Ti are no longer needed, and that a concurrent (short) transaction is waiting for these objects to be released. We can split Ti into a transaction Tj containing the requested objects, and a transaction Tk containing the remaining objects. Tj immedi-

ately commits, which allows the short transaction to execute, increasing concurrency.

To determine the split point, the LLT can either continue operating until a good opportunity for a split arises, or it can undo its work up to a previously determined split point. The first case assumes such a point will arise in the future, and could keep the short transaction waiting for a long time, but places no extra requirements on the LLT. The second case allows the short transaction to commence immediately, but requires the LLT to be restarted from the split point.

In [PKH88] it is argued that the second case is more common, and the authors have defined a framework to facilitate such splits. In this case, it is clear that it would be desirable to be able to automatically redo the actions that were rolled back, to effectively undo the rollback. To allow this, the framework considers accesses to database objects not in isolation, but as a group of operations that are performed by a single process. Such a group of operations corresponds to the actions of a high-level tool such as, for example, a compiler, which reads a number of source files and produces a number of output files.

For each transaction a history is maintained, and for each group of operations on the data a flag is kept that indicates if the operations can be automatically repeated or not. In our compiler example, we could repeat the operations automatically. However, we can consider editing of a source file, for example, as not being able to be repeated automatically. When a split is required, the history is examined to determine when a split may occur: we can split the LLT at a given point if all groups of operations that occurred after this point can be repeated.

In summary, Split transactions was created to address the needs of open-ended activities that have uncertain duration and developments. Split transactions allows a transaction to be split in two separate transactions, and in certain cases for one of these to immediately commit, freeing resources to be used by other transactions. Rollback of such split transactions is not straightforward however, as some of the data accesses may already have been committed and therefore compensating actions will have to be performed.

2.6 Conclusion

In this chapter we treated transaction management, and existing extensions to the traditional transaction model in the form of Advanced Transaction Models (ATMS).

The chapter started with an introduction of general concepts of transactions and requirements for correct concurrent execution of transactions. We introduced the well-known ACID properties for transactions, and have shown how these can be met using a locking scheduler.

Next, we talked about Advanced Transaction Models. There are well-known lim-

itations to the classical transaction model, especially concerning the length of transactions and cooperation between transactions. To address these issues, a number of Advanced Transaction Models have been defined in the literature, each focusing on a specific problem, and providing a possible solution. We have discussed four Advanced Transaction Models in detail: Nested Transactions, Sagas, Relatively Consistent Schedules and Split Transactions.

Common to many ATMS is the use of secondary transactions: a transaction conceptually separate from the main control flow of the application that automatically runs when some constraints are met. In the models we discussed we encountered these in case of a rollback, where compensating transactions perform a semantic 'undo' of already committed work.

A second general remark considering the use of these ATMS is in place here; we have noted that for Nested Transactions Gray and Reuter [GR93] comment that there is a strong link between modularization of software and the use of Nested Transactions, but find few cases of the use of this ATMS. Also, we have seen that Garcia-Molina [GM83] states that the cost of providing semantic information is high but unavoidable in some cases. Furthermore he finds that in some cases an ATMS is already used implicitly, and that it is better to make this explicit. So, while we find convincing arguments for the use of an ATMS, we also see that application programmers still stick to using classical transactions in such cases.

In the entire chapter, we have taken a somewhat high-level approach when discussing the different models, without detailing how an application programmer implements the usage of an ATMS. However, as we mentioned in the introduction, such transaction demarcation code is required to start and stop transactions and to associate data accesses with a transaction. This demarcation code will be spread out over different parts of the application, which poses some significant problems, as we discuss in the next chapter.

Chapter 3

Aspect-Oriented Programming and Transactions

Oh no, not again.
—**Douglas Adams, petunias can think in "The Hitch Hiker's Guide to the Galaxy"**

In the previous chapter, we discussed transaction management and talked about a variety of advanced models for transaction management (ATMS), explaining the properties of the different models.

We have also briefly stated that to use transaction management, the application programmer will need to place transaction demarcation code to indicate the beginning and end of transactions, and what data accesses are included in which transactions. This entails that the demarcation code will be spread out over the application, because it will be found everywhere in the application where transactions are used. As we shall see next, this clashes with the software engineering principle of separation of concerns [Par72, HVL95].

As a result of this clash, application code which includes transaction demarcation code is known to be hard to develop and maintain. In this chapter, we talk about a means to resolve this issue, called Aspect-Oriented programming, and discuss existing efforts which modularize transaction demarcation code for classical transactions.

The results of the above research in modularization of transaction demarcation code show that such modularization yields important benefits in ease of development and maintenance of the application. Therefore, to aid implementation and maintenance of systems using an ATMS, we also modularize this demarcation code using

Aspect-Oriented programming. At the end of this chapter we outline the technical approach we have taken to modularize demarcation code for ATMS, which has led us to the identification of tangled aspect code.

3.1 Aspect-Oriented Programming

Applications which need to fulfill a large number of requirements are generally difficult to develop. One principle can be used to ease the development of these applications: separation of concerns. However it is not always easy to separate these concerns at the level of the code, as some of them have a system-wide impact. A technique to separate out these system-wide concerns is Aspect-Oriented programming (AOP).

In this section we introduce the principle of separation of concerns, and briefly discuss AOP as a technique to achieve this separation.

3.1.1 Separation of Concerns

In developing an application, the programmer has to ensure that diverse requirements are met. To be able to meet such a wide variety of requirements, various techniques and methodologies have been developed to reduce the complexity of the software, to increase its comprehensibility, and therefore also to ease its maintenance.

Common to many of the above techniques is the concept of "divide and conquer". The software is decomposed into smaller pieces, which individually are easier to comprehend and manage, and these pieces are later combined to comprise the full system.

A good decomposition of the software allows each separate piece to address a specific concern of the application, i.e. a specific subset of related requirements, such as the core algorithm of the application, management of concurrency, logging, persistence, etc. Conceptually, such a subset corresponds to a design decision for the application, and a module then "hides such a decision from the others" [Par72]. In OO programming, we divide into classes, the idea being that each (group of) classes will address one concern. Handling these concerns separately, not only conceptually but also at an implementation level, is known to greatly ease coding and maintenance. This is because the programmer only needs to keep in mind the exact concern currently being addressed, and can ignore the other concerns, which significantly reduces the complexity of the task at hand. The above principle is aptly named *separation of concerns* [Par72, HVL95]. The ideal of this separation is that each concern of the application is addressed in one and only one module, or class, of the program.

Having each module address one and only one concern is, however, hard to achieve, due to an effect known as "the tyranny of the dominant decomposition" [TOHJ99]. This states that, due to the fact that software can only be modularized in one way

at a time, for a sufficiently large problem, there will always be concerns that cannot be encapsulated into one module, and will therefore be scattered throughout the system. Typical examples of these difficult concerns are persistence, distribution, and indeed, transaction management. This is because as these concerns have a system-wide impact, they can not be added by simply adding a new module to the system using traditional modularization techniques. Take for example transaction management: adding a "Transaction manager" object to the system does not suffice. As we have said in the previous chapters, demarcation code needs to be inserted into the application code, indicating the start and end of transactions, and which data accesses need to be associated to what transactions. In other words, the code in a wide number of existing classes will have to be adapted, breaking the separation of the concurrency concern. The special-purpose concerns mentioned above are said to *cross-cut* the class structure [KLM+97].

Note that, for a number of such *crosscutting concerns*, programming languages have provided constructs that handle these concerns. For example, considering thread-based concurrency management, in Java such synchronization is handled with the `synchronized` keyword and the `wait`, `notify` and `notifyAll` methods. However, it still proves to be hard to write and understand code using these constructs [HVL95, Han99], and many patterns have been proposed to tackle this [Lea99]. Also, if the number of special-purpose concerns in the program increases, it will progressively become more difficult to comprehend the code, increasing the difficulty of developing and maintaining the program. This is because, although the different concerns can be specified independently in an abstract fashion, integrating them into final code is hard, as is extracting the original concerns from the produced code or modifying the code of a single concern. This is largely due to the fact that although there is a loose coupling between the concerns at a conceptual level, at code level this coupling becomes strong when the concerns have been integrated.

If we can separate the different concerns not only at a conceptual level, but also at a code level, by e.g. having one class for each concern, we would have a full separation of concerns. Such a full separation would lead to a higher level of abstraction, making the code easier to understand and maintain. A number of techniques have been developed to address the problem of these cross-cutting concerns, achieving a higher separation of concerns. A flagship technique to modularize cross-cutting concerns is Aspect-Oriented Programming, which we discuss next.

3.1.2 Aspect-Oriented Programming

In *Aspect-Oriented Programming* (AOP), the concerns which are known to cross-cut the class structure of an application are called aspects. Aspects are said to "cut across both each other and the final executable code" [KLM+97]. As said above, although

these aspects can be easily separated at a conceptual level, the code which integrates these aspects is "a tangled mess of aspects" [KLM+97]. This tangling of aspects indicates that in the application code the concerns are not separated, which makes the program hard to comprehend.

In other words, in AOP a concern is either implemented in a component, or in an aspect. A concern that cannot be encapsulated cleanly into a module (such as a class or a group of classes), is called an *aspect*. In a program, all concerns which are components are sometimes also called the *base aspect*. Examples of such aspects are transaction management, and thread-based concurrency management, as we discussed above, as none of these concerns can be cleanly encapsulated in a separate module and added to the program.

The goal of AOP is to achieve a higher degree of separation of concerns by allowing the programmer not only to reason separately about the aspects, but also to implement them in separate modules. This is achieved by specifying the code pertaining to an aspect in a separate aspect specification, in a special aspect language. Once all modules and aspects are defined, a special tool, called an Aspect Weaver, combines these specification to achieve the desired application behavior. The weaver is able to do this because it knows not only how each aspect can be transformed into executable code, but it also knows the relationships between the different aspects, and the correct way to combine them. The process of modularizing an aspect in the above fashion is called *aspectisation* [KG02, SLB02, RC03].

Initial aspect languages were designed to address one specific concern, and therefore to allow the aspect to be easily expressed by the programmer. Some of the first aspect languages were, for example, languages for concurrency control [VLK97], numerical accuracy [I+97], and space optimization [MKL97]. Because of this specific nature of the languages, the programmer can express the code in a natural form, leading to a greater ease of use over general-purpose languages. We feel that these initial aspect languages give AOP a special appeal over other solutions, such as, for example, meta-level programming, which do not use such concern-specific languages.

Current AOP research, however, uses more general aspect languages, in which various aspects can be expressed [Asp05]. The most widely known general aspect language is *AspectJ*, out of which some generally accepted terminology with regard to aspect languages has emerged.Therefore, we outline this terminology below:

- *Join Points* are conceptual entities that represent certain points in the execution of a program. Conceptually, it is at these moments at run-time that an aspect will execute.

- To determine the join points in the code of the program, *Pointcuts* are defined by the programmer. A pointcut usually declares the join points by means of

static descriptions of the source code, combined with selected runtime information, such as the value of certain variables and the contents of the call stack. Furthermore, a pointcut may selectively expose some part of this execution context.

- The aspect code, defined by the programmer, that is executed when a join point is reached is called *Advice*. This code contains the behavior of the aspect, and has access to the execution context which was exposed by the pointcut.

- Code for the aspect may also be inserted into different classes statically. Such code is called *Inter-type declarations*. This code usually declares extra members in the class and may even change inheritance relationships between classes.

So, in this terminology, an aspect is considered to be a collection of pointcuts, advice for that pointcut, and additional inter-type declarations.

As transaction management is a clear example of an aspect, it comes as no surprise that work has already been performed defining aspects for transaction demarcation, and we review this work next.

3.2 AOP and Transaction Management

Transaction demarcation code has been found to cut across different modules in the system: every module which makes use of transactions will have to include demarcation code to indicate start and end of transactions, and what data accesses are included in the transaction. Three papers have been published that include proposals of how to aspectise the concern of classical transaction management [KG02, SLB02, RC03], and we discuss these in this section.

Although transaction management is generally accepted by the AOP community as being an aspect, we remark that only few papers have been published about this subject. We were only able to find three papers about modularizing transaction management as an aspect, and in two of these papers transaction management is but a minor topic. Common to the three papers is that they use AspectJ, which implies that the base aspect is implemented in Java. Furthermore, join points of the beginning of transactions always are the beginning of methods, and join points of the end of transactions always are the end of these methods, i.e. method boundaries are equal to transaction boundaries. Therefore, because transactions coincide with methods, we call these methods *transactional methods*. Note however, that nested transactions are not covered, and the papers do not discuss what happens if a transactional method calls another transactional method, which in nested transactions could straightforwardly be mapped to spawning a child transaction.

The first paper we review, by Kienzle and Gerraoui [KG02], uses transaction management, amongst others, to evaluate AOP. Second, the paper by Soares et al. [SLB02] has transaction management as a side-effect of aspectising distribution and persistence in AspectJ. The third paper, by Rashid and Chichyan [RC03], aspectises persistence, and includes transaction management. Lastly, we discuss some of our own work [Fab04b], on treating transactions as an aspect and focussing on handling transactional failures.

AOP: Does it make Sense?

Kienzle and Gerraoui [KG02] describe a transaction management aspect using AspectJ to explore how AOP can help modularise concurrency in distributed systems. They describe an AspectJ aspect that interacts with OPTIMA: an object-oriented framework that supports transaction management. The goal of the paper is to analyze if transaction semantics can be fully aspectized, and by extension, investigate if AOP is useful for all concerns, or in their own words: "If AOP makes sense".

The authors start with the assumption that a first program is written, totally disregarding transaction management, and subsequently transaction management is added to that program by use of AOP. This approach, however, fails because of two reasons: deadlocks and irreversible actions, and we detail these next.

The first problem is the introduction of deadlocks: the use of transactions implies adding the possibility of deadlocks to the application, as we have seen in 2.1.3. In such cases, the TP Monitor will rollback a deadlocked transaction to break the deadlock cycle, and will inform the application of this action. This entails that the application may not be totally blind to the use of transactions, and that it must be prepared to handle these exceptional situations.

The second problem is irreversible actions: an important feature of transactions is the ability to rollback the work which has been performed, for example when aborting a transaction to break a deadlock. It is clear that, to be able to undo all the work of a transaction through a rollback, no side-effects or irreversible actions may have been performed in that transaction. Again, this entails that the application may not be totally blind to the use of transactions, as the work which will be enclosed in a transaction must be reversible.

This leads the authors to conclude that transaction management must be kept in mind throughout the entire application development, that it can not be completely extracted to a separate aspect. We agree, in general we can say that AOP can not make up for total neglect of certain concerns, and that the overall design of the system still needs to take into account these extra concerns. In that respect, AOP does not make sense, but this is because we can not treat AOP as a silver bullet. AOP simply gives us the possibility to further separate concerns out at the level of

the code of the application, where previously this was difficult. As we have just seen, we cannot guarantee a complete separation, but nonetheless an increased separation of concerns at the code level still increases ease of implementation and maintainability of the application. This is shown by Soares et al. [SLB02] and Rashid and Chitchyan [RC03], which we discuss further on

In the case of transaction management, one element is not separated out and remains in the base code. This element is the non-functional concern of handling rollbacks. If we take transaction management into account from the onset of building the application, the design will not include side-effects in transactional operations. Therefore, the transactions can be fully rolled back by the TP Monitor. The application, however, will still have to handle these rollbacks. This is not a major issue, as we can already perform some rollback handling within the aspect, which we discuss later in this section.

Concerning notification of rollbacks, the authors criticize AspectJ for the inability to modify the methods signature, as this entails that an explicit declaration of exceptions thrown by transactional rollbacks is impossible. Instead these exceptions must be a subclass of `RuntimeException`, which has as a result that type checking for these exceptions at compile time is not performed. The authors note that an application programmer, "relying on the fact that important application exceptions are checked, might forget to handle [these exceptions], which results in an incorrect program behavior" [KG02]. We agree with this critique and consider this an important downside of the AspectJ weaver.

Additional criticism by Kienzle and Gerraoui on aspectising transaction management with AspectJ is the lack of possibilities for transaction optimization: they note that it should be possible to define different kinds of interaction with the transaction manager, depending on the access to the code. For example, if only read access is performed within the transactional code, the transaction manager should be informed of this read-only access at the start of the transaction by a specific demarcation instruction. This information then allows the transaction manager to perform optimisations specific to read-only transactions. The authors, however, have not found the required support in AspectJ to define an aspect for such optimizations. We agree that this support is lacking, but note that this is a drawback of the AspectJ language and weaver, and not necessarily of AOP in general. AspectJ could be extended, or another language and weaver could be built which investigates the code of the method to determine if it is read-only, and if so, includes the corresponding demarcation code.

The authors also consider further optimizations of transaction management, on a per method basis, by declaring specific semantics of the method. As a first example they consider conflict relationships between methods, and declaring methods that commute, overriding conflicts generated by operations within the method. A second example is associating an existing *inverse method* with each method, if available,

which is analogous to the concept of compensating transactions. These can then be used by the TP Monitor to optimize logging. However, such information cannot be deduced from the source code, similarly to what we proposed above, as noted in the paper: "Obviously such semantic knowledge about methods can not be guessed automatically. It must be provided by the application programmer." This is relevant for us, as this kind of semantic information is exactly what is required by ATMS, as we have seen in the previous chapter, and therefore this has to be specified by the application programmer. Furthermore, we note that inverse methods are considered to be part of the base aspect, and not the transaction aspect, as the authors note that these are associated to the transactional method, and not implemented in the transaction aspect. We agree, as the transaction aspect should concern itself solely with the transactional behavior of the application. Adding inverse methods consists of adding base aspect code in the transactional aspect, which is contrary to separation of concerns.

To conclude, Kienzle and Gerraoui state that to successfully aspectise transaction management, the use of aspects must be kept in mind throughout the application development cycle. AOP allows us to further separate concerns out at the level of the code, but the design of the application must allow for the use of transactions. Furthermore, the authors critique the lack of possibilities for optimization of read-only transactions, but we state that this is an AspectJ-specific problem, and not a limitation of AOP in general. Lastly, the authors consider declaring extra semantic information for other optimizations, such as associating inverse methods to transactional methods.

Distribution and Persistence with AspectJ

Soares et al.[SLB02] implement a number of persistence and distribution aspects, which include transaction management, also using AspectJ. They implement a 'health watcher system' first without and afterwards with these aspects, to evaluate the usefulness of AOP.

The paper details the implementation of several aspects in AspectJ, including distribution through Java RMI, persistence through JDBC, transaction control on the persistent objects and exception handling. The authors state to have successfully separated distribution and persistence from each other and from the business and presentation logic, and conclude that the use of aspects indeed assists in producing a better implementation. In the distribution aspect, the authors consider the AspectJ implementation to be better than a pure OO solution as the latter would require about 20% more code which is not separated out at all, and as a consequence is not as adaptable and extensible. For the transaction management aspect, the authors report that in the pure OO version "For each [transactional] method there are at least 6 lines of tangled code to call the transaction life-cycle methods [i.e. demarcation code]" and

that factoring this out into an aspect "avoids tedious work and increases productivity". In general, we can state that this is indeed a successful aspectisation of the concern of transaction management, with a significant increase in separation of concerns.

Note that, while transaction management is indeed treated as an aspect in this paper, it does not receive as thorough a treatment as by Kienzle and Gerraoui. The discussion focusses on how they declare methods as transactional, and how optimizations of read-only methods, deemed impossible in the paper above, have been implemented. Thorough treatment of rollbacks of transactions, and the need for irreversible actions within transactional methods is not provided. The authors only indicate that rollback of a transaction is not notified to the caller of the transactional method and remark "any exception that is thrown and not handled by a transactional method aborts the transaction."

The most notable conclusion of Soares et al. is that aspectisation of transaction management code does significantly increase separation of concerns and they report the aspectised version to be more adaptable and extensible.

Persistence as an Aspect

Rashid and Chitchyan [RC03] implement a persistence aspect, which includes a transaction management part, also using AspectJ. The paper explores whether persistence can be aspectized and if this aspect is reusable for other applications.

In modularizing persistence, the authors build a reusable framework of both components and aspects. The authors claim that one of the widely misunderstood promises of AOP is that an aspect equals one single unit of AspectJ-like code, but that this is not so, except for the most simple cases. Instead, the authors state that aspectisation has different classes and aspects collaborate to modularize one cross-cutting concern. This also "ensures that aspectisation is not forced and in fact leads to a natural separation of concerns".

Also, we must remark that, as in the paper above, transaction management is not treated as throughly as in [KG02], as it is only considered part of the general persistence concern. Again, exception handling for transaction management is not fully addressed. The authors assume that all exceptions thrown in transactional code are related to database access, and therefore simply rollback the transaction when these occur, but without further notifying the application, as above.

The authors conclude that separation of concerns at the level of application code is achieved using AOP, as a highly reusable persistence aspect can be developed. Caution must be taken, however, that the application and the persistence aspect are not developed independently. Instead, the concern of persistence must be considered from the onset of designing the application, as is stated in [KG02].

Exception Handling for Transactions

A recurring pattern in the above three approaches is the lack of treatment of the exceptions caused by the transaction aspect. This prompted us to investigate the topic of exception handling for transactions, which we reported in [Fab04b].

The fact that the non-functional concern of exception handling remains in the base application makes the separation less clean. A higher degree of separation would be achieved if this concern could also be tackled in the aspect. The goal of this work therefore was to centralize transaction declaration and handling of rollbacks due to exceptions in one location, namely the aspect definition, and exploring the feasibility of using a concern-specific language for transaction management. We defined a new aspect language, specifically for declaring transactional methods, which includes the ability to declare exception handlers for these methods in plain Java. Furthermore, the aspect language also provides a generic exception handler, which simply restarts the transaction.

When defining the aspect, the programmer now adds exception handling code to the declaration of the transactional method in the transaction aspect language. This improves on the existing work discussed above, as the programmer may now decide what approach to take upon receiving a transactionally related exception, in contrast to the 'do nothing' approach used above. The programmer can now choose, in the transaction aspect, which approach to take specifically for each method. As any Java exception handler may be provided, the possibilities include, but are not limited to, trying to fix the problem and retry the transaction or notifying the application an error has occurred by throwing a custom exception.

The concern of rollback handling remains present in the caller of the method, however, as these exception handlers may not be able to provide a solution for all exceptions. For example, if the database server is down, the exception handler may not be able to solve this problem and will have to inform the application of this error. Less severe exceptions, such as a rollback caused by a deadlock, can however be tackled by these handlers. As a result, the aspect does allows for a larger degree of separation of concerns.

Conclusion

In conclusion, we have seen convincing reports that AOP does indeed lead to a significant increase in separation of concerns at the level of application code, and specifically when considering transaction management as an aspect.

As a result of the greater separation of concerns at the code level, AOP aids in development and maintenance of applications using transactions, as reported in [SLB02, RC03]. However, it is important to realize that these applications may not

be constructed totally blindly from the concern of transaction management, as is discussed in [KG02, RC03].

Also, it is observed in [RC03] that one aspect does not necessarily equal one unit of AspectJ-like code. One aspect may consist of a mix of aspect and module code, which together treat the intended concern. Lastly, within the confines of classical transactions, [KG02] also envisions declaring extra semantic information, similar to the information used in ATMS: declaring compensating methods and overriding conflict relationships for operations. This suggests us to consider an aspect for ATMS, additional to aspects for classical transactions, and leads us to our problem statement, which is next.

3.3 AOP for ATMS Demarcation code

As we discussed above, an important issue with transaction management is the fact that demarcation code cross-cuts across the application code. As a consequence, the application code becomes difficult to write and maintain, due to a low separation of concerns. This issue has been addressed using AOP, modularizing transaction management as an aspect.

While classical transaction management has been modularized as an aspect, we see no such work on ATMS. However, in ATMS the need for a good separation of concerns is at least as important as in classical transactions, if not higher. If we reflect on the nature of demarcation code for ATMS, we see that these models in general require much more information than classical transactions. Consider the following examples: in nested transactions, a child transaction must reference its parent transaction, in Sagas, the saga must be subdivided in different steps and each step of the saga must declare a counter-step, and in Relatively Consistent Schedules each step must declare an interleaving descriptor set. Such information can not be deduced automatically, as we observed in 3.2, and therefore needs to be provided by the programmer. This results in corresponding demarcation code which must contain much more information and will therefore probably be more complex than demarcation code for classical transactions, and we return to this in more detail in chapter 7. As a consequence, we state that for ATMS the need for aspectisation is even greater than for classical transactions.

In brief, we state that an important problem of ATMS is the high complexity and cross-cutting nature of the demarcation code. We claim that, therefore, modularizing ATMS as an aspect will result in code which is easier to write and maintain. Also, as there are many different ATMS in existence, this aspect must be general and powerful enough to capture a variety of these ATMS, and not be specific to just one model, or its usefulness will be too limited.

As different ATMS address different issues with classical transaction management, support for only a few ATMS significantly limits applicability of the solution. To increase applicability, it is important that many ATMS are supported in one general aspect, such that many of the issues of classical transaction management are addressed. We need to uncover fundamental underlying principles of ATMS to be able to support such a wide variety of ATMS. If we choose instead to develop many different aspects, one for each ATMS, it is not certain that these commonalities between different ATMS will be found and can be reused. Therefore in developing each new aspect little or no previous work can be used to speed up development. Furthermore, the list of ATMS is open-ended, as new models may be developed at any time. So if we have a general aspect for ATMS, it is likely that support for these models can be added to the aspect with minimal work.

The generally applicable aspect language for ATMS will not reify the concepts used by the different ATMS, but work in terms of a common foundation for the different ATMS. As a consequence, the different ATMS will be expressed based on this common foundation. This implies that aspect programs, defining the transactional properties of the application, will be written at a low level: in terms of the common foundation. As a result, this yields a large volume of aspect code where multiple sub-concerns are tangled, as we will illustrate in chapter 7, which leads us to the observation of tangled aspect code.

Considering solely the low level of aspect code, we can already state that it would be better to have aspect programs define transactional properties at a high level: in terms of the concepts of the ATMS actually being used in the application. If we achieve this, the programmer can more easily specify and reason about transactional definitions, because he reasons in terms of the ATMS, and not in underlying, more low-level concepts. This is reminiscent of the original aspect languages, where each language was specific to one concern.

To address the technical issue of modularizing demarcation code we propose a two-level solution. First we develop a general aspect language for ATMS, which forms the kernel of our approach, and is therefore called KALA: (**K**ernel **A**spect **L**anguage for **A**TMS). Second, on top of this kernel, a family of ATMS-specific languages are built, each reifying the concepts in that ATMS. Programs in the ATMS-specific languages are translated to the equivalent KALA programs, and woven in the application at compile time, as illustrated in figure 3.1.

By having a general kernel language, we ensure that a wide variety of ATMS can be supported, and by building different languages on top of the kernel, we can tailor each language to the targeted ATMS, so that aspect programs in these languages are as concise and intentional as possible. Also, by defining a family of such languages, they share as much similarity as possible, making it easier to learn a new language for a new ATMS.

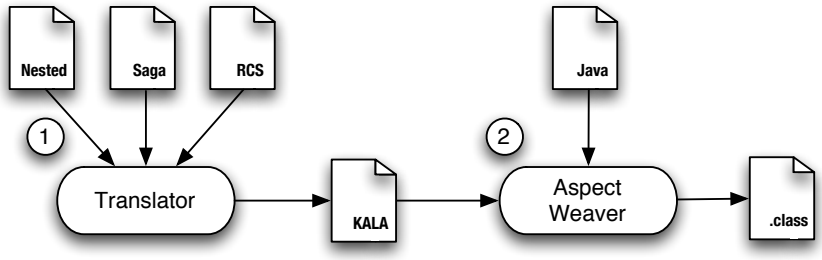

Figure 3.1: At compile time first code in ATMS-specific languages is translated to equivalent KALA code, which second is woven into the Java code.

3.4 Conclusion

In this chapter we have briefly introduced Aspect-Oriented programming as a prominent technique to modularize cross-cutting concerns, which are concerns of which the code can be found throughout different modules in the application. The advantage of modularizing such concerns is that they can now be treated separately, not only conceptually but also at the code level, which is known to greatly enhance implementation and maintenance [HVL95].

We identified transaction management as an example of such a cross-cutting concern, as demarcation code will be spread out throughout the application, everywhere transactional data access occurs. We discussed published work on defining transaction management as an aspect, which has shown that this indeed significantly increases separation of concern. Also, resulting from this work is the observation that while transaction management is separated out from the application, the rest of the application must still take this concern into account.

This led us to the observation that no work has been performed to modularize transaction demarcation code for ATMS, and that such an effort is indeed necessary due to the higher complexity of demarcation code for ATMS, compared to demarcation code for classical transactions.

The above observation gave rise to the technical problem addressed in this thesis, which is that to create an aspect for ATMS, this aspect must be general enough to address many models, while still being straightforward to use by the application programmer. A general aspect for ATMS is required because of the large variety of ATMS: supporting only a few significantly limits applicability. Generality, however may not come at the price of ease of use: application programmers should be able to write aspect code at the level of the concepts provided by the ATMS being used.

We briefly outlined the two-level solution we propose, which is to have a general aspect language for ATMS, and on top of this build different languages, specific to each ATMS.

Before we can construct this general aspect language for ATMS, however, we need to further investigate these ATMS from a conceptual point of view. This reveals that such an ATMS itself should not be considered as one monolithic concern, but in itself is composed of multiple concerns, which has consequences that are discussed in the next chapter.

Chapter 4

Tangled Aspect Code

They both savored the strange warm glow of being much more ignorant
than ordinary people, who were only ignorant of ordinary things.
— **Terry Pratchett, Discworld scientists at work in "Equal Rites"**

Having introduced Aspect-Oriented Programming, in the previous chapter, as the means by which we wish to modularize demarcation code for ATMS, we now further investigate these ATMS, to see how these can be aspectised. Considering different ATMS, we see here that a given ATMS cannot be treated as a monolithic block, but is composed of different concerns. Furthermore, we can tweak the ATMS to the requirements of the application being developed by modifying the implementation of such a concern. Therefore, we should be able to implement each of these concerns in a separate module, and combine these modules into the aspect for ATMS when weaving. However, the problem is that the demarcation code for the different concerns in this ATMS will be tangled, i.e. the demarcation code implements the concerns in a tangled way. Therefore we need to be able to again apply separation of concerns to the aspect code. In other words, we need to be able to untangle the aspect itself into different modules.

Investigating AspectJ, which is the most popular technology for aspectising transaction management as we have seen in 3.2, we find the aspect composition mechanism supplied by AspectJ lacking. AspectJ does not allow us to cleanly write the different concerns in separate modules, and compose these into an aspect for ATMS when weaving. Instead, AspectJ forces us to write code, at the aspect level, in which the multiple concerns present in the ATMS are tangled, a phenomenon we call tangled aspect code. We therefore need a different solution for the aspect language and weaver,

avoiding tangled aspect code for ATMS, and we provide for such a solution, discussed later in the dissertation, from chapter 8 onwards.

We now start with investigating the different concerns present in ATMS, before identifying where the code for these concerns can be found in the demarcation code. We conclude this chapter with a discussion of the aspect composition facilities of AspectJ, which leads us to the observation of tangled aspect code.

4.1 Concerns in Advanced Transaction Management

If we consider different ATMS from a conceptual point of view, we find that these ATMS are not one monolithic block, but incorporate different design decisions. For example, consider how rollback is handled in the Sagas ATMS (which we discussed in 2.3): compensating transactions are executed in the inverse sequence of the steps of the saga. A different design decision could have been taken here, and compensating transactions could run in a different sequence. For example, we could specify that compensating transactions run in the same sequence as the steps of the saga, or even let the compensating transactions run in parallel, to attempt to speed up saga rollback. With different possible implementations for rollback handling, we should encapsulate this functionality into a separate module [Par72], in other words, we can consider rollback handling as a concern in the Sagas ATMS.

Looking at other ATMS, we find that other ATMS, such as Nested Transactions and Relatively Consistent Schedules, contain multiple of these concerns: not only the handling of rollbacks, but also two extra concerns, which we call view management and delegation, as we see in 4.1.2. Each of these concerns can be modified in some way, by providing an alternative implementation with different properties. Changing the ATMS in this way allows us to modify the behavior, yielding a different ATMS with slightly different properties. This then allows an existing ATMS to be tweaked to better fit the design of a particular class of applications, which is one of the goals of this dissertation.

In this section, we show how we identified these concerns in multiple ATMS. Similarly to the previous research on aspectising transaction management, which we discussed in 3.2, we assume that transaction boundaries coincide with method boundaries, i.e. that a transaction starts at the beginning of a method, and ends when that method ends. Hence, for classical transactions a transaction is contained within a single method. Considering ATMS, however, the demarcation code for a single advanced transaction may be spread out over different methods. This is because one advanced transaction may be composed of multiple classical transactions, each contained in a separate method. For example, in Nested Transactions, a tree of such transactions is built and a Saga consists of a sequence of transactions. For clarity, therefore, in the

following discussion we will usually focus solely on one transactional method belonging to the overall advanced transaction.

We start the identification of concerns in ATMS by first discussing the concern of rollback handling within the Sagas ATMS, and second investigate Nested Transactions and Relatively Consistent Schedules, which treat multiple concerns. Note that this discussion is at a conceptual level and does not yet include actual demarcation code for these ATMS. A Code example is given and discussed later in this chapter, in 4.2.

4.1.1 Concerns within the Sagas ATMS

A saga, as we have seen in 2.3, is an advanced transaction which is composed out of a sequence of steps, each step corresponding to a classical transaction. To be able to rollback the execution of a saga, each of these steps has to define a counterstep, i.e. a semantical undo. When rolling back the saga, the currently active transaction is rolled back, and the countersteps for the already committed steps are executed, in reverse order of step sequence. Translated to application code, i.e. methods, each step will correspond to a method, as is each counterstep. A top-level method which calls the different steps in sequence can then play the role of the advanced transaction, and also be transactional. In this discussion, however, we limit ourselves to the demarcation code for just one step in the saga, and see which concerns we encounter.

If we consider conceptually the tasks that need to be performed by demarcation code for such a step, we can infer that some parts of this code will treat managing rollback of the saga. This code will perform the work of defining and starting up compensating transactions, ensuring that these only begin when the saga aborts, and that they run in the right sequence. All of these low-level tasks comprise the code for one concern, which is managing rollback of the saga.

We can indeed consider *management of rollbacks* a true concern in this demarcation code, as it is a design decision of the ATMS that lies conceptually at a higher level than the implementation details of the code, i.e. the different tasks of the code we identified above. This corresponds to the original consideration of a concern by Parnas [Par72] where he states that a module, i.e. a concern, corresponds to the implementation of a design decision.

We claim that management of rollbacks is part of the design of the ATMS, as different implementations of this concern can be easily envisioned. For example, consider the Sagas example we introduced in 2.3 where calculating and adding the interest to all accounts of a bank is implemented as a Saga. Each step in the saga consists of calculating the interest for one account, and adding it to that account. Compensating this step is performed by removing the calculated interest from that account. There is no need, however, for the compensating steps to be executed in the reverse order of step execution. Having the compensating steps execute in the same order as step

execution would also perform correct rollback of the Saga. Furthermore, we can even attempt to speed up rollback by executing the compensating steps in parallel. Making such changes to the order of compensation effectively amounts to tweaking a concern within the sagas ATMS, creating a new ATMS with a different fit to the design of the application.

Demarcation code for Sagas will not be limited to code which treats rollbacks, however, other tasks will also be performed by this code. Each transaction within the saga has to be started and ended, and the operations performed by the application on shared data have to be included in the transaction. These tasks are not directly involved with determining compensating transactions and their sequence. Instead, these tasks build a general frame in which fits the concern of transaction compensation. We could also make modifications here, and we take an example from [CR92]: we could allow each step of the saga to be a saga itself, instead of a transaction.

To summarize, we find demarcation code which does not treat the rollback concern and which does treat a design decision for which another implementation is possible. Therefore this code treats a second concern of the saga, which is the management of the structure of the saga. More generally speaking, we call this concern the *management of the structure* of the overall transaction, as defined in the ATMS.

In other words, if we reflect on the different tasks performed by saga demarcation code, we find that this code treats two different concerns: first the management of the structure of the overall transaction, and second management of how rollback is performed. We can consider these two concerns as true concerns, as they are conceptually at a higher level than the different tasks within the demarcation code, and that a different implementation of such a concern is possible, as we have shown above, leading to a new ATMS.

The conceptual division of the demarcation code for Sagas in two concerns has an important impact on how we must regard an aspect for advanced transaction management. The concern of transaction management using the Sagas ATMS is not one monolithic block, it is itself divided into two concerns. As a consequence, the aspect that is built for this concern will itself consist out of two concerns, and this aspect will, conceptually, not treat solely one concern, but multiple concerns. We could propose to use two aspects to implement the concerns of transaction structure, but conceptually this is a poor choice. If rollback handling is implemented as a separate aspect, there is no meaning of adding this concern to the code if the Sagas structure is absent, as there would be no actual transactions to rollback. Contrariwise, the Sagas structure is not complete without the rollback handling concern. Therefore, conceptually these two must be joined at the level of the aspect to form a whole that has meaning and can be woven into the base code.

We now investigate other ATMS to see if demarcation code for other ATMS also treats multiple concerns or if Sagas is an exceptional case. We perform this next, where

we look at Nested Transactions and at Relatively Consistent Schedules.

4.1.2 Multiple Concerns in other ATMS

After having discovered two concerns treated by the demarcation code for the sagas ATMS, we now consider two more ATMS: Nested Transactions and Relatively Consistent Schedules, to see if here also multiple concerns can be found in the demarcation code. To determine this, we also contemplate the work to be performed by this code at a conceptual level, and we start by considering Nested Transactions.

Nested Transactions

Considering the concerns of structure management and rollback handling, which we identified above, for the Nested Transactions ATMS, we find that these also return here. Nested Transactions build a tree-like structure of multiple transactions, at runtime, so the concern of structure management will certainly be present here. Regarding rollback handling, in Nested Transactions the rollback of a child transaction does not imply rollback of the parent, but if the parent aborts, and some of its children have not ended, these will need to abort. As a result we find that the concern of rollback handling is also present in the demarcation code.

Considering the specification of Nested Transactions, we can identify more work to be performed by the demarcation code. This code will also need to tell the concurrency manager to remove isolation between child and parent transaction, i.e. allow a child to view the intermediate results of the parent. Also, if the child commits, demarcation code will tell the concurrency manager to delegate the used resources to the parent before committing. This work clearly does not pertain to neither the concern of managing the structure of the transaction, nor performing rollback. Therefore this code will have to treat different concerns, as we see next.

Allowing a child transaction to view the parents' intermediate results is an implementation task, but we can also envisage this at a higher conceptual level. Managing such views can be much more extensive work than just setting the view to include one other transaction. We could for example, construct a new ATMS by allowing the siblings of a child transaction to be in the view and this transaction to be in their view. This will require not only views to be set and unset, but also some extra management to be performed to keep track of these siblings. We can therefore regard *view management* as a concern in its own right, the implementation of which in the Nested Transactions ATMS happens to be simple.

This leaves us with transaction demarcation code performing delegation from the child to the parent when committing. Clearly this does not belong in the concern of view management, as the concepts are fundamentally different. Instead, *delegation*

management is also a concern in its own right, which again happens to have a simple implementation in the Nested Transactions ATMS. We could, for example, remove this concern from the ATMS, letting commitment of child transactions write their results directly to the database, which again creates a new ATMS.

Note that we have now created two new ATMS with some peculiar effects regarding consistency of the database due to decreased isolation in the example of the view management concern, and directly writing to the database in the example of the delegation concern. This does not imply that these are not valid ATMS, since, as we have said in chapter 2, certain ATMS may decide to forego classical database consistency criteria in favor of exhibiting a certain behavior. It is the work of the ATMS creator to identify the peculiarities of the ATMS and the effects they have. The application designer needs to evaluate the different ATMS, taking into account their peculiarities, to find the ATMS most suited to the design of the application.

Relatively Consistent Schedules

This ends our discussion on the different concerns present in the Nested Transactions ATMS, and we now proceed with Relatively Consistent Schedules (RCS, see 2.4), investigating the different concerns present in this ATMS. If we consider the structure of such a transaction we see that it consists of a sequence of steps, which is similar to the structure in Sagas. A significant difference, however, is that at the end of each step, work is not committed to the database, but instead is kept within the transaction. We can achieve such behavior in an application by composing the structure concern of Sagas with the delegation concern. In other words, the different steps of the RCS are implemented by different methods, all called by a top-level method, which is the top-level transaction. At commit time of each step, responsibility for operations on shared data is delegated to the top-level transaction. As in Nested Transactions, this has as effect that the work performed in this transaction is not committed to the database. Subsequent steps of the RCS, however, need to have access to this work, and therefore in the beginning of each step we delegate the work from the top-level transaction to this step.

RCS specifies more than just this behavior though, in RCS a given class of compatible transactions may view the intermediate results in between the execution of the different steps. We can implement this by setting the view of the top-level transaction to a collection of transactions which contains the compatible transactions in between each step.

In 2.4, we have remarked that the authors of the RCS ATMS in [AFTO89] do not explicitly address the issue of rollback, beyond recommending the use of counter-steps. In other words, we can state that they have not thoroughly considered the concern of rollback handling in their ATMS. We can take, as they suggest, the existing imple-

mentation of rollback handling for sagas, and add it here, to achieve the recommended form of rollback handling.

The above implies that we can summarize the RCS ATMS as being the Sagas ATMS which two concerns have been added to: view management and delegation. Note that both of these concerns are heavily coupled, adding only one of these concerns to Sagas makes no sense. If we only add delegation we, in effect, make one long transaction and lose the benefits of splitting into steps. If we only add views, the work of the steps is committed to the database, so in between steps no data remains to be viewed by compatible transactions.

This concludes our discussion on RCS, where we identified the same concerns as in Nested Transactions, and have remarked that, considering the concerns, RCS is an extension of Sagas with an implementation of view management and delegation, where both concerns are heavily coupled.

Generalization

A somewhat similar observation considering the different concerns within an ATMS has already been performed by the authors of the ACTA formal model [CR92], which we discuss in 6.1. The authors do not refer to concerns in the ATMS, but instead use terms like 'important notions' and 'properties of the model'. Four such 'notions' and 'properties' are given: visibility, recovery, permanence and consistency. Visibility corresponds to our view management concern, and recovery corresponds to our rollback concern. We cannot provide a straightforward match to permanence and consistency. Permanence refers to "the ability of a transaction to record its results in the database" [CR92]. Consistency is said to pertain "the correctness of the state of the database" after the transaction commits [CR92]. Permanence could be matched to the delegation concern we identified above, because after delegation a transaction is no longer responsible for recording its results to the database. However, we feel that this is not as good a fit as visibility and recovery. Considering consistency, we do find it an important feature of an ATMS, but we consider it rather as an outcome of how the different concerns within the ATMS interact than a concern itself.

In general, we can speculate that the concerns we identified above in Sagas, Nested Transactions and RCS: rollback, structure management, views and delegation, are relevant to other ATMS. This not only because some of these concerns have also been identified in the work on ACTA [CR92], but we can infer this when considering each concern, as we do below.

Rollbacks A vital property associated with transactions is the ability to perform rollbacks, as we have seen in 2.1.4. This is confirmed by the identification of the recovery property in ACTA, which we discussed above. It is therefore almost

required for an ATMS to provide some kind of rollback support. As a result we can safely say that the majority of ATMS will contain the rollback concern.

Structure management This concern is not identified by ACTA, but is most probably present when one advanced transaction is composed of multiple atomic transactions, such as Sagas or Nested transactions. In these cases, managing the relationships between the different atomic transactions forms the structure management concern.

View management As also observed in ACTA, modifying the isolation property (one of the fundamental ACID properties, as we discussed in 2.1) is recurrent in different ATMS. We have also seen this above in Nested Transactions and RCS. Therefore, if an ATMS concerns itself with the isolation property, the view management concern is present in this ATMS.

Delegation management While related to the ACTA property of permanence, delegation management is not as good a match as view management or rollbacks. We do, however, consider it as a concern which does recur in multiple ATMS: we identified it in RCS and Nested Transactions above, and is also present in Split Transactions, which we have introduced in 2.5, and Altruistic Locking [SGM94], which we have not discussed. Therefore, we can say that it will probably also be present in other ATMS.

Consistency This is an ATMS property which is identified in the work on ACTA, but as we said above we consider it rather as a result of the interaction of the different concerns in an ATMS than a concern in itself. We did not identify this concern in the ATMS we investigated above, and do not expect this concern to be explicitly present in any other ATMS.

Note that we consider this list to be open-ended. Although we have identified multiple concerns in many ATMS, it is possible that a new ATMS contains a concern which we have not yet treated.

4.1.3 Conclusion

In the previous sections, we identified four different possible concerns within ATMS: the structure of the transaction, how rollback is handled, management of views and delegation of results. We made an initial observation on Sagas, and confirmed this for Nested Transactions and RCS. Furthermore we have generalized our observations to the presence of these concerns in ATMS we have not treated.

We found multiple instances of ATMS composed out of different concerns, and have argued for the presence of these concerns in other ATMS, which we have not treated.

We can therefore conclude that we may not consider all ATMS as one monolithic concern. Instead, we must be able to consider an ATMS as a composition of multiple concerns.

In other words, if we create an aspect for ATMS, we must take into account that this aspect can itself be composed of different concerns which might be crosscutting concerns. As we have argued above, conceptually, these concerns must be joined at the level of the aspect, and not at the base level. This is because at base level separately they have no meaning, it is only at the level of the aspect that, if different of these concerns are joined, the aspect can have a meaning.

Furthermore. we have also seen, in Nested Transactions as in the Sagas section above, that we can modify the different concerns, i.e. provide a different implementation, which yields a new ATMS with its own peculiarities. The ability to make such changes is important for the goal of this dissertation, which is to have an ATMS of which the design fits the design of the application. As an example we have shown how the rollback concern of the Sagas ATMS can be modified to enable faster rollback of the saga. By allowing a programmer to tweak an ATMS through a modification of such a concern, we can enable a better fit between an ATMS and the program that uses this ATMS. In the example, for the application, the sequence of compensation was unimportant and this allows for faster rollback to be implemented, which can be very useful for the application.

Having identified different concerns within the ATMS, and the benefit of being able to modify the code for such a concern, we now return to the transaction demarcation code, to locate these concerns within this code.

4.2 Location of Concerns Within Demarcation Code

We have concluded above that the aspect of transaction management using an ATMS will itself consist of multiple concerns, i.e. the aspect code will treat multiple concerns. We have furthermore stated that it is important that each of these concerns can be changed, such that an ATMS can be tweaked to a new form that better fits the design of a target application.

If we wish to modify concerns of an ATMS, or add implementations for new concerns to an existing ATMS, the aspect that modularizes the ATMS must take into account this requirement. The aspect code must be structured in such a way that modification of concerns is easy, enabling easy creation of new ATMS through changes in the implementation of these concerns. To realize this structuring of the aspect, it is beneficial to look at the base code that would implement such an ATMS, i.e. the transaction demarcation code for this ATMS. This allows us to evaluate the feasibility of different approaches to creating such an aspect with regard to weaving.

4.2.1 A Concrete Example of Demarcation Code

To support the discussion on location of ATMS concerns within demarcation code, we discuss here an example of a method containing such code. In this method we illustrate how such code can be subdivided in five phases and which tasks can be performed in each phase.

The code we discuss here is part of a larger example, which is discussed in chapter 7 and the full code of which is given in appendix B. The example we discuss in chapter 7 is a banking application, which is part of a multi-tier distributed system, and we focus on the code for a money transfer operation. This money transfer operation uses the Sagas ATMS, and is divided in three steps: first the transfer itself, second printing a receipt to be given to the customer, and last logging the transfer operation. Of these three steps, we include the code for the transfer operation here.

We do not discuss the code we provide here in detail, as it requires background knowledge on the design and implementation of the concurrency manager, which has not yet been provided. Instead, we restrict the discussion below to a more general description of the different parts within the method code, the tasks performed by this code and the purpose of these tasks. This code is discussed in detail in 7.3.1 and appendix B and therefore we refer to these sections for an in depth treatment.

The following code is the implementation of the transfer operation, where transaction demarcation code has been emphasized like this.

```
1  private void transfer
2     (BankAccount from_orig, BankAccount to_orig, int amount)
3     throws TxException
4  {
5     TransactionManager txmgr = TransactionManager.getCurrent();
6     Integer tx_id = txmgr.newID();
7     txmgr.addTransaction(tx_id);
8     Integer saga = txmgr.lookup(Thread.currentThread());
9     txmgr.addToGroup("Saga"+ saga + "Step",tx_id);
10
11    final Integer comp_id = txmgr.newID(); //for compensation
12    txmgr.addTransaction(comp_id);
13    txmgr.addToGroup("Saga"+ comp_id+ "Comp",comp_id);
14    txmgr.bind("Saga"+ comp_id+ "Comp",comp_id);
15
16    final BankAccount compfrom = from_orig; //for inner class
17    final BankAccount compto = to_orig; //for inner class
18    final int compamount = amount; //for inner class
```

```
19
20      Runnable compensator = new Runnable()
21  {
22      public void run(){
23          undoTransfer(compfrom, compto, compamount, comp_id);
24      }
25  };
26
27      txmgr.addDependency(saga, "ad", tx_id);
28      txmgr.addDependency(tx_id, "wd" ,saga);
29      txmgr.addDependency(comp_id, "bcd" ,tx_id);
30
31      new Thread(compensator).run();
32
33      Forcing bf = txmgr.mayBegin(tx_id);
34      if (bf == null)
35          txmgr.begin(tx_id);
36      else {
37          txmgr.rollback(tx_id);
38          return;
39      }
40
41      try {
42          BankAccountWrap from = new BankAccountWrap(from_orig);
43          BankAccountWrap to = new BankAccountWrap(to_orig);
44          int from_amount = from.getAmount(tx_id);
45          int to_amount = to.getAmount(tx_id);
46          to.setAmount(to_amount + amount, tx_id);
47          from.setAmount(from_amount - amount, tx_id);
48
49          Forcing cf = txmgr.mayCommit(tx_id);
50          if (cf != null)
51              throw new TxAbortedException();
52
53          txmgr.addDependency(comp_id, "cmd" ,saga);
54          txmgr.addDependency(comp_id, "bad" ,saga);
55
56          txmgr.commit(tx_id);
57      }
58      catch (TxException ex){
```

```
59          txmgr.mayAbort(tx_id);//will always succeed
60          txmgr.rollback(tx_id);
61          throw ex;
62      }
63  }
```

We can subdivide the above code in five different phases: the preliminaries from line 1 to 25, the begin phase from line 27 to 41, the running phase from line 41 to 47, the commit phase from line 49 to 56, and the abort phase from line 58 to 61. We treat each of these phases in more detail next, in the above sequence.

Preliminaries

In the preliminaries, general setup is performed, obtaining references to external entities and building structures for later use. In lines 5 to 7, this transaction is registered to the transaction scheduler. Lines 8 and 9 link this step of the saga to the actual saga instance which is executing. The compensating transaction for this step is registered to the transaction scheduler in lines 11 thru 14. The `compensator` object, created in lines 16 to 25 enables the compensating transaction to run as a secondary transaction. Note that we do not include the code for the compensating transaction, i.e. the implementation of the `undoTransfer` method here, as it is not relevant to this example.

Begin

The begin phase contains demarcation code for actions that take place at the begin time of the transaction. Lines 27 and 28 put a relationship in place between this step and the saga: if this step aborts, the saga aborts, and if the saga aborts, this step should also abort. A constraint is put on the compensating transaction in line 29: it may not begin before the transfer step has committed. The compensating transaction is then spawned as a secondary transaction, in a separate thread in line 31. Lines 33 thru 39 verify that this transaction may indeed begin, which would not be the case if the saga has aborted, given the relationship placed in lines 27 and 28. In line 35, if the transaction may indeed begin, the transaction scheduler is informed that the transaction begins.

Running

The running phase essentially consists of the base application logic of the method, in which wrappers on shared data are used. These wrappers ensure that the transaction

scheduler mediates accesses to this data. This guarantees that the transactional properties of the model regarding data accesses are adhered to. We do not discuss this code in further detail.

Commit

In the commit phase, demarcation code for actions that take place when a transaction commits is located. Lines 49 thru 50 verify that this transaction may indeed commit. which is the case if the saga has not aborted. Lines 53 and 54 constrain execution of the compensating transaction (which is a secondary transaction) to only begin if the saga aborts. In line 56 the transaction scheduler is informed of commitment of this step, which will commit all accesses to shared data in this step to the database.

Abort

The abort phase treats aborting this step, which entails aborting the transaction. Lines 59 and 60 inform the transaction scheduler of the abortion of this step. This undoes all the modifications to shared data performed by this step, and will also trigger the saga to abort. Furthermore, line 61 ensures that the caller of this method is informed of the abortion of this transaction by throwing an exception

Conclusion

This concludes our discussion of the five different phases within the above demarcation code. For each phase we discussed the tasks performed by the demarcation code and the purpose of these tasks

It is important to note that the general sequence of tasks within this demarcation code is fixed, as a later operation will usually rely on results of an earlier one. For example, in the begin phase, in line 31 the compensating transaction can only be run in a separate thread after, in line 29, this transaction is prohibited to immediately start executing. Therefore, we cannot modify the order of the statements within the above code, which is important in 4.3.

4.2.2 Skeleton Code

From the concrete example code we have given above, we now take a step back to perform a more general analysis. Not all methods that include such demarcation code perform all the tasks we discussed above. Instead, these methods will only perform the tasks required for the ATMS being used. We can, however, expect a minimum skeleton of demarcation code to be present in the method: some preliminaries, the original method code, the decision to commit or abort the transaction, the calls to

the transaction scheduler to begin, commit and abort and an indication of failure in case of such an abort. Consider the different parts of this skeleton code. Some preliminaries will be required to, for example obtain a reference to the transaction scheduler. The original method code always has to be present, or no work is performed. Also, demarcation code must include calls to begin, commit and abort, otherwise the life cycle of the transaction is not complete. Lastly, in case of a transaction abort, code is needed that notifies the caller of abortion of the transaction. In summary, the skeleton code ensures a minimum of functionality that must be provided by the demarcation code, and therefore we expect this code always to be present.

Further considering this skeleton code, we can locate this code in the different phases outlined above: preliminaries, beginning, running, commit and abort. In other words, these phases will be found in each transactional method and we illustrate their flow in figure 4.1. We therefore describe the purpose of each of these phases next, from a general perspective. We also show how a concrete ATMS adds to this skeleton code using as examples the demarcation code we discussed above and tasks to be performed for the Nested Transactions ATMS:

Preliminaries In the preliminaries phase, skeleton code will perform preliminary setup work, such as obtaining a reference to the transaction scheduler. The example code above adds the definition of the compensating transaction to this skeleton. In Nested Transactions, a reference to the parent of this nested transaction can also be obtained here, to be used later, and the parent can be informed of this new child.

Beginning In the beginning phase, skeleton code will issue a call to the transaction scheduler indicating the beginning of the method. In the sagas example above, the step is linked to the saga instance, and the secondary transaction is restricted to run only after the step commits. Demarcation code in Nested Transactions will relax isolation between this transaction and the parent here.

Running In the running phase the original code for the method is contained, with data accesses made transactional. Also in this phase the decision to either commit or abort the method will be taken. We foresee no other demarcation code to be required here, for no ATMS.

Commit In the commit phase, the skeleton code informs the transaction scheduler of transaction commit. In the sagas example above, the compensating transaction is restricted to run only when the saga aborts. For Nested Transactions, demarcation code will first delegate all the work of this transaction to the parent, before the commit in the skeleton code.

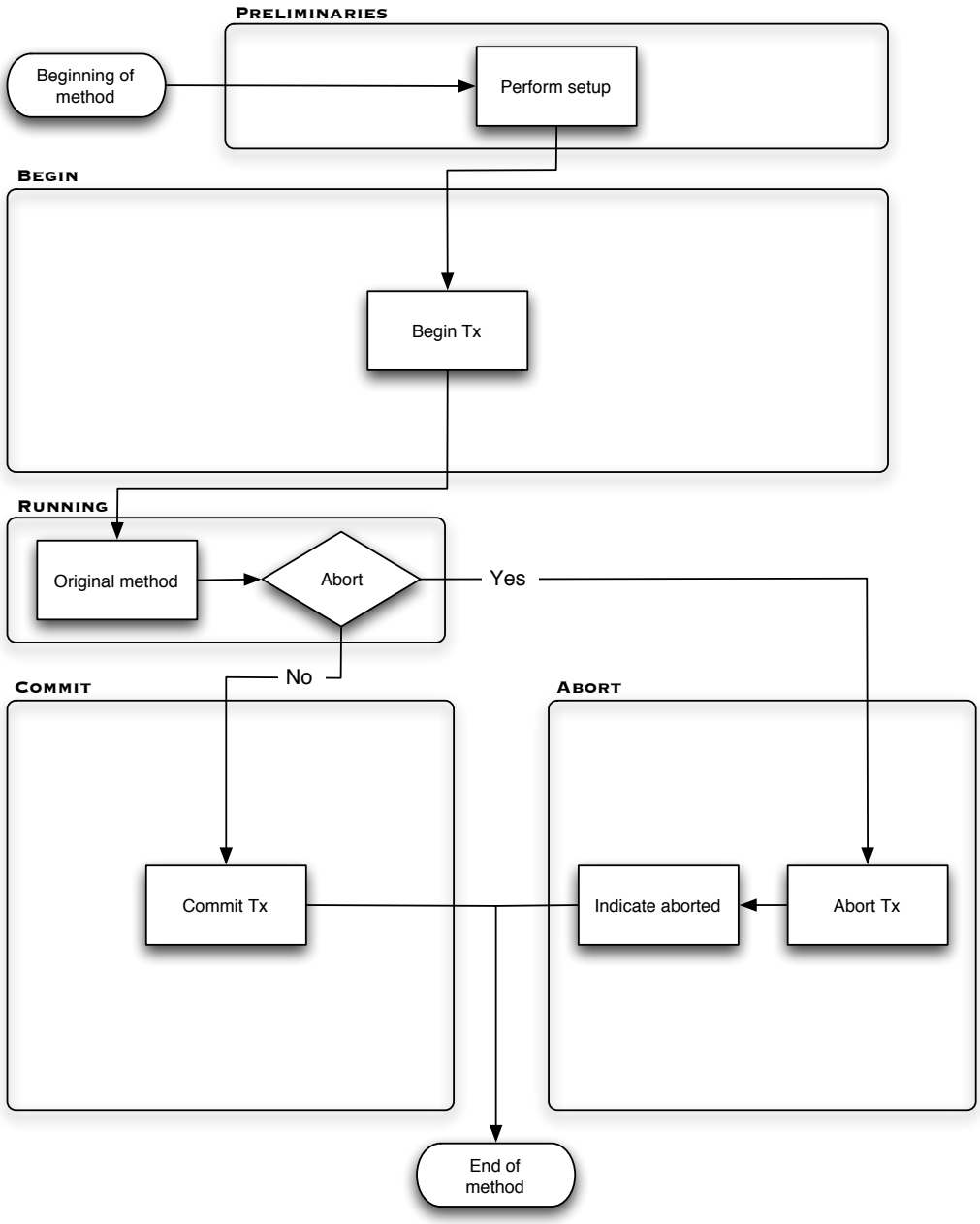

Figure 4.1: Phases in skeleton transaction demarcation code within one method.

Abort In the abort phase, the skeleton code informs the transaction scheduler of transaction abort, and will also inform the caller of this method, for example by throwing an exception. The example code above adds no further starements. For Nested Transactions, demarcation code will have to ensure that all the children of this method also abort.

Given this subdivision of work performed by demarcation code, and tasks performed by the demarcation code in these phases, we can now investigate the different ATMS we discussed previously in this chapter. This allows us to see where the different concerns of these ATMS can be found in the demarcation code.

We start by re-evaluating the sagas code above, and identify the location of code for the rollback concern.

4.2.3 Location of the Rollback Concern in Sagas

Using as a base figure 4.1, outlining the general sequence of phases within one method, we now indicate in which phases demarcation code is contained for the rollback concern in the Sagas example code discussed above. We show this in figure 4.2, where the phases colored red indicate that they will contain demarcation code for this concern, in addition to the skeleton code.

In figure 4.2 we see that, in addition to the skeleton code, code is placed in the preliminaries phase, begin phase and commit phase. In the preliminaries phase, the compensating transaction is specified inline 11 thru 25. In the begin phase, this transaction is restricted to run in line 29 and started as a secondary transaction in line 31. In the commit phase, the compensating transaction is specified to run when the saga aborts in lines 53 and 54.

Considering the preliminaries and the begin phase and the location of concerns within this code, we see that code in the beginning of preliminaries phase is skeleton code and at the end of the phase treats the rollback concern. This is followed by the begin phase where first the rollback concern is treated and then again we have skeleton code. In other words, in this part of the demarcation code first we have code for the skeleton, then for the rollback concern, and then again for the skeleton. This shows that skeleton code is tangled with code for the rollback concern.

4.2.4 Location of View and Delegation in Relatively Consistent Schedules

Demarcation code for sagas only tackles two concerns; the structure of the advanced transaction and how to perform rollback. We have identified other ATMS that contain more concerns above, such as RCS. In RCS, along with the above two, the concerns of view management and delegation are present. To illustrate the layout of demarcation code that tackles multiple concerns, in figure 4.3 we provide an indication for the

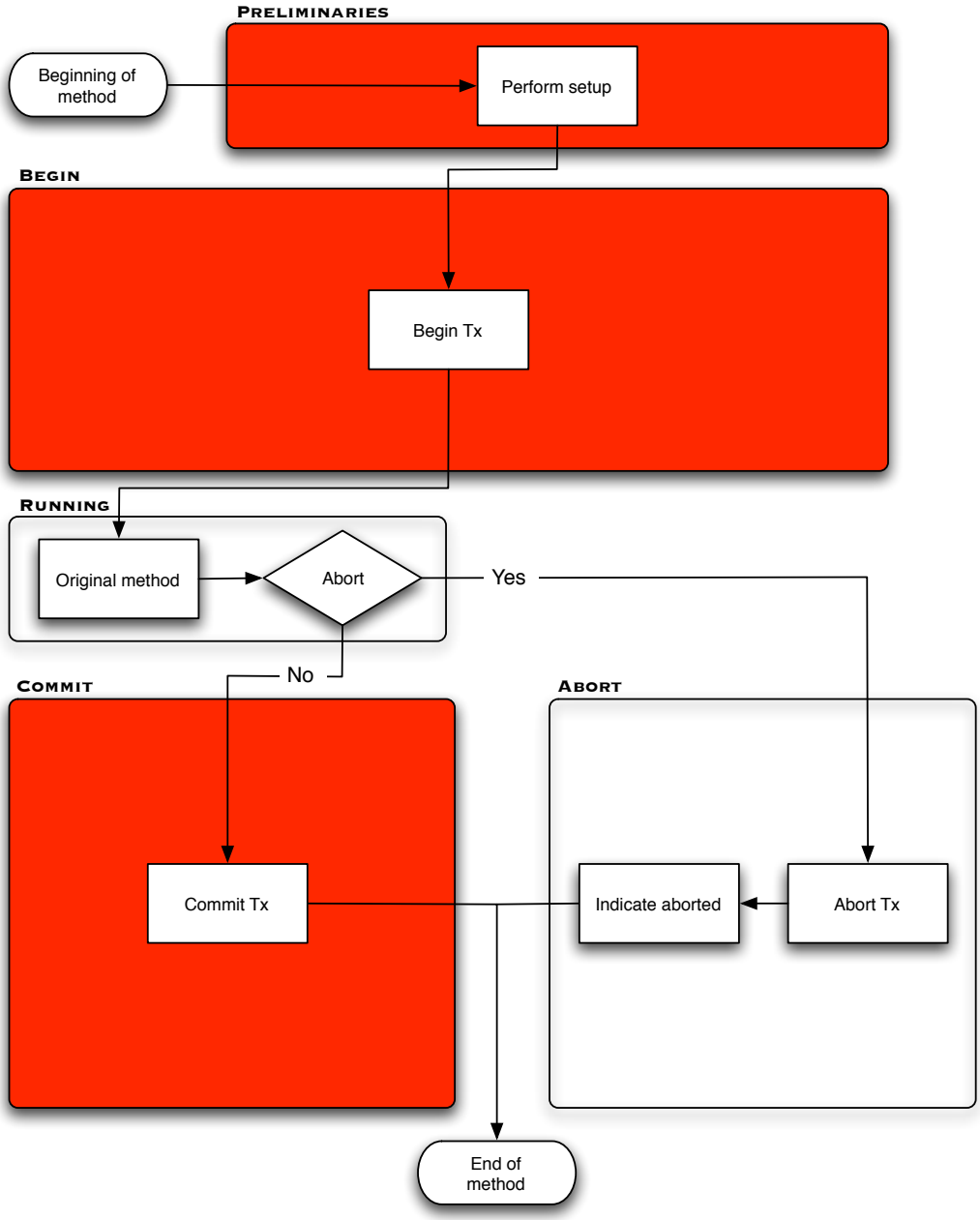

Figure 4.2: Flow chart of transaction demarcation code indicating Sagas code for the rollback concern.

demarcation code of a step in RCS for the concerns of rollback handling, view management and delegation. In this figure, we have colored the phases which will contain demarcation code for the rollback concern in red, the view management concern in green, and the delegation concern in blue.

Considering figure 4.3, we see that multiple phases have different colors, i.e. multiple concerns will be treated in those phases. We therefore discuss this code concern by concern. The red code, for the concern of rollback management is in the same locations as in the Sagas example above. This is not surprising, as we have already stated that we can consider RCS as an extension of Sagas in 4.1.2, i.e. the demarcation code for Sagas is repeated, and therefore we do not discuss it further.

Demarcation code for view management, in green, starts at the preliminaries where a reference to the top level transaction is obtained, to be used in the following phases. In the beginning phase, this reference is then used to remove the view of the top-level transaction. This will be performed before the call to the transaction scheduler to indicate the transaction begins. As a result, no other transactions see the intermediate work performed within this step. At commit time, view adding is performed, i.e. the inverse operation. The new view of the top level transaction, i.e. the new group of compatible transactions, is obtained and set right before the the call to the transaction scheduler to indicate the transaction commits. Returning to the preliminaries and the begin phase treated above, for Sagas, this entails that within the code which already tangles skeleton and rollback, an extra concern is added. Furthermore this adds to the entanglement, as this code is not added at the beginning of the end of the phase, but has to be added within the begin phase, after the rollback concern is treated and before the skeleton code.

The delegation concern, in blue, works hand in hand with the viewing concern, taking intermediate results from the top-level transaction before this step starts, and at the end of the step delegating all operations back to the top-level transaction. To implement this, first the name of the top-level transaction is retrieved in the preliminaries phase. Second delegation from the top-level to the step is performed immediately before the call to the transaction scheduler to indicate the transaction begins. Third and last, delegation from the step to the top-level is performed immediately before the call to the transaction scheduler to indicate the transaction commits. Again returning to the preliminaries and the begin phase, delegation code is also placed within these phases. This further adds to the entanglement, as the delegation code is placed before the skeleton code in the begin phase. This results in code which first is skeleton code, then rollback code (in the preliminaries), followed by code for rollback, view, delegation and skeleton code (in the begin phase), i.e. this code is tangled.

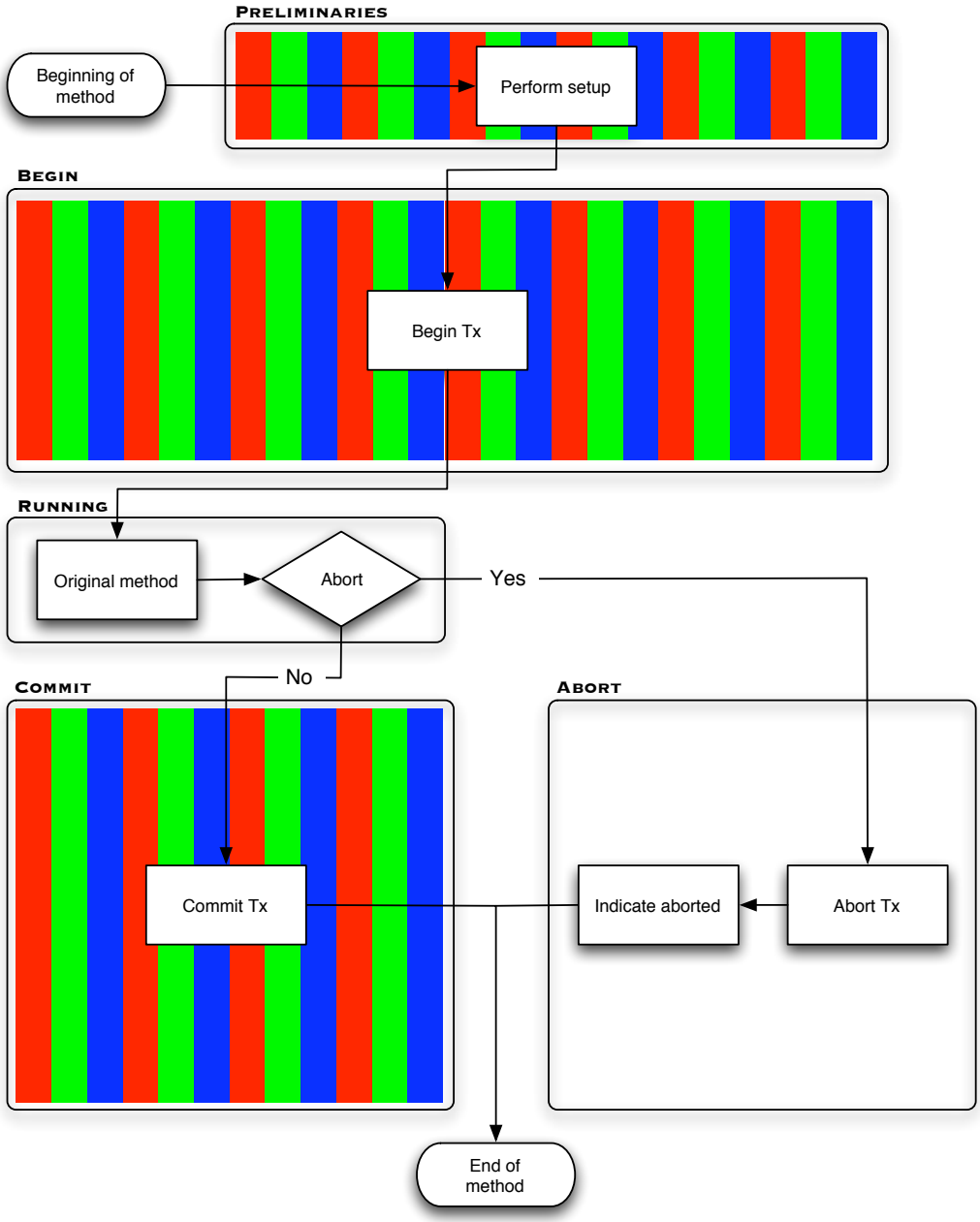

Figure 4.3: Flow chart of transaction demarcation code indicating RCS code: rollback concern in red, views in green and delegation in blue.

4.2.5 Conclusion

To conclude, we see that with an ATMS containing many concerns, the demarcation code not only will be placed in different locations. but that these different locations will treat multiple concerns in a tangled way. In other words, the demarcation code will contain multiple tangled concerns.

As we have said in 4.1.2 we want to be able to modify the code for these different concerns, tailoring an ATMS to better fit the design of a given application. This means that we want to have a modular treatment of concerns at the level of the aspect, regardless of the tangled nature of the resulting demarcation code. Therefore, the aspect weaver technology that we want to use must allow these different concerns to be expressed as different modules at the aspect level, and generate tangled demarcation code, such as illustrated in figure 4.3, which we address next.

4.3 Tangled Aspect Code

In the previous section, we deduced that we are faced with transaction demarcation code that treats multiple concerns in a tangled way. We want, however, that these different concerns are separated at the aspect level in separate modules, so we can easily provide a different implementation for a given concern. In this section we consider well-known and popular aspect languages and weavers, more specifically AspectJ-like weavers, to determine if they provide support for such separation. We find that such support is lacking, resulting in what we call tangled aspect code.

We have found that an ATMS conceptually contains multiple concerns, and that we want to be able to modify these concerns individually. Therefore, these concerns should be encapsulated in different modules. These different modules, when merged together, form the definition for one aspect, to be woven into the application code. As different modules we consider the three concerns identified above: rollback handling, view management and delegation. Each of these concerns, however, can expect the minimum skeleton code to be present in the demarcation code, as we have discussed above. Therefore, we can consider the skeleton code to be a primary concern to which the other concerns are added when merging. In other words: to build the aspect definition of an ATMS, the skeleton concern is merged with a three optional concerns: rollback handling, view management and delegation.

Concern Composition in AspectJ

There seems to be a consensus in the research community to use AspectJ [Asp05] as weaver technology to aspectize transaction management, as we have seen in 3.2. Therefore we now investigate if AspectJ-like weavers provide support for the compo-

sition discussed above. We find that AspectJ does not support merging of different concerns within an aspect into one. Instead AspectJ allows to combine different aspect definitions, where each aspect definition is considered as a full-fledged aspect. AspectJ, however, is not powerful enough to provide for a merge of ATMS concerns, as we show next.

If we define the different concerns defined above as different aspects, we need to use the facilities provided by AspectJ to combine these aspect definitions when weaving. AspectJ provides one combination mechanism, *advice precedence*, that essentially is sequencing of aspects [Asp05]. With advice precedence, a precedence relationship between aspects can be declared. Different aspects can intervene before, after or around a method definition or method invocation, and the sequence can be set by a declared precedence relationship. We find it striking that this module composition mechanism essentially only allows reordering of steps in processing, as this runs completely contrary to the observation of Parnas that "modules will not correspond to steps in the processing" [Par72], and we show below that this mechanism is indeed insufficient.

Let us consider the skeleton code for the method as the first aspect in the composition, we then apply this aspect around a method definition. We have the preliminaries and begin phases before the method, and the commit and abort phases after the method, giving us the structure in figure 4.1. Adding subsequent aspects, for example, rollback handling, to this sequence is problematic. We have given an example of rollback handling code for sagas above, in 4.2, and have seen that this code is placed within the preliminaries, begin and commit phases, as illustrated by figure 4.2. We can also split this code in a before the method and after the method part, to aid in composition. What remains now is to declare precedence for both before parts and both after parts. This should combine these parts into yield the complete Sagas implementation.

Schematically, we illustrate this as follows: we represent the method as M, the before and after part of the structure concern as BS and AS, and the before and after part of the rollback concern as BR and AR. Execution sequence of the different parts is given by $\rightarrow$ i.e. $BS \rightarrow M \rightarrow AS$ represents the structure concern woven around the method. To compose this with the rollback concern, precedence can be declared such that we have four different possibilities:

$$1 : BR \rightarrow BS \rightarrow M \rightarrow AS \rightarrow AR$$
$$2 : BS \rightarrow BR \rightarrow M \rightarrow AR \rightarrow AS$$
$$3 : BR \rightarrow BS \rightarrow M \rightarrow AR \rightarrow AS$$
$$4 : BS \rightarrow BR \rightarrow M \rightarrow AS \rightarrow AR$$

None of the above compositions, provided by AspectJ, are adequate. As we have seen above, the rollback concern is contained within the skeleton, in the preliminaries phase. As this means it is not executed before or after the skeleton, the execution precedence mechanism provided for by AspectJ does not suffice. The same argument

holds for the demarcation code executed after the method: rollback handling code is contained within the commit part of the skeleton, and not before or after this skeleton. In other words, conceptually AspectJ does not provide for a clean way in which a rollback concern can be merged with the skeleton concern to form one full aspect definition, in the case of the example, for Sagas.

It could be argued that the inability to cleanly merge the skeleton concern with a rollback concern in AspectJ is due to the artificial nature of the skeleton concern. Suppose that no ATMS will only use the skeleton concern, and therefore conceptually we must consider that at least one concern in a given composition will include the skeleton code. For example, we can consider the rollback handling concern above to include the skeleton code, yielding an aspect split in a before and after part, where each part contains the skeleton in which rollback handling code is placed. The question now remains of how to combine this aspect definition with a definition of view management and a definition of delegation, to form demarcation code that uses RCS. Again we see that the composition functionality provided by AspectJ does not allow for a merge of these different concerns, as the code for views and delegation must be placed within the code for rollback handling (which includes the skeleton code), and not before or after this code. To summarize, we can not provide for a conceptually clean way to compose the different concerns for ATMS into one aspect using AspectJ, because AspectJ only allows the different aspects to execute in sequence using advice precedence, and not for multiple aspects to be tangled into one.

Remark, however, that the above discussion centers on a conceptually clean way in which the different concerns can be merged into one aspect. We can envision a strategy in which the skeleton concern is split up into different small concern parts, for example each part corresponding to a phase given above, and the same to be performed for the other concerns. These small concern parts can then be sequenced using advice precedence declarations, which then forms a full aspect definition for a given ATMS. It is clear, however, that this is conceptually not a clean solution, as firstly it entails splitting each module for a concern in different sub-modules, and secondly providing an extensive advice precedence declaration for all these sub-modules. We should be able to keep each concern as a single module, and have a minimal composition definition.

Tangled Aspect Code

As AspectJ does not allow us to split the different concerns of an ATMS into separate modules at the level of the aspect and to automatically combine them when weaving the aspect, this entails that we will not be able to write the different concerns separately. In other words, if we use AspectJ as weaver, in the aspect we define we will have to manually compose the different concerns to form one AspectJ definition of an aspect. This aspect code will be tangled code, as we have shown that the dif-

ferent concerns can not be composed sequentially. In other words, what we have is a move from base code that is tangled to aspect code that is still tangled. We call the phenomenon where aspect code still contains different concerns of which the code is tangled *tangled aspect code*.

We note that while the phenomenon of tangled aspect code bears a resemblance to the idea of aspects on aspects [VC05], we find a fundamental difference between these two with regard to the completeness of the aspect. In aspects on aspects, multiple crosscutting concerns at the level of the base code are created as aspects, such as, for example, a persistence aspect and a logging aspect, and the execution of one aspect is the join point for another aspect. For example we can combine persistence with logging such that whenever a persistence operation is performed, this operation is logged. In our case, however, the different concerns within an ATMS are meaningless without each other and need to be composed at aspect level: for example in RCS the delegation concern has to be combined with view management, rollback and the skeleton to become one complete ATMS. This is in contrast to the examples of persistence and logging, which already separately provide a complete functionality. In other words, with tangled aspect code, the different concerns need to be combined at the level of the aspect to provide functionality, whereas in aspects on aspects the different concerns can be woven separately into the base level.

Furthermore, conceptually combining the different aspects when weaving becomes a difficult task if we would have two aspects where each consists of multiple concerns. If the concerns of both aspects are all simultaneously woven into the base code this composition can become extremely complex with a combinatorial explosion of possibilities to combine the different concerns. If, on the other hand, we first allow the two different aspects to be composed at the aspect level, and then combine these two concerns when weaving, conceptually the composition of these two aspects is much more straightforward.

Conclusion

To conclude, in this section we have seen that it is impossible to use AspectJ in a conceptually clean way to compose the multiple concerns within an ATMS. Instead we are forced to write aspect level code that itself treats multiple concerns in a tangled way, a phenomenon we call tangled aspect code. We note that tangled aspect code is different from the concept of aspects on aspects as in the latter each aspect is a crosscutting concern at the level of the base code, while in the former each concern resides at aspect level.

Therefore, if we want to use AspectJ to write an aspect definition for an ATMS, we cannot divide the different concerns within that ATMS into different modules. However, we want to have such a modularization, as we want to be able to easily

modify the implementation of a given concern of an ATMS. To support this, in this dissertation we provide an aspect language and weaver that allow for such a modularization, taking into account the properties of the specific domain of ATMS. Using domain information of ATMS allows us to define an aspect language and aspect weaver that does allow for splitting the concerns within an ATMS in different modules, and merging them into one aspect definition, which we show in the remainder of this text.

4.4 Conclusion

In this chapter, we deduced that, in general, an ATMS should not be considered as one monolithic block, but rather as a composition of different concerns. We have identified four such concerns: the structure of the extended transaction, handling of rollback, management of views and performing delegation. Furthermore, we have shown that we can provide a different implementation of a given concern of an ATMS, which yields an ATMS with different properties with regard to this concern. This can be used to tweak an existing ATMS to better fit a particular class of applications.

Having an ATMS composed of different concerns implies that the code for each concern should be contained in a separate module at the level of the aspect. We first, however, looked at the code at base level, i.e. traditional demarcation code, to identify what an aspect weaver needs to perform to be able to merge these concerns into the final code. In this demarcation code, we have seen that these concerns are crosscutting, i.e. the code for the different concerns is tangled, and therefore, the aspect weaver needs a composition mechanism that is able to tangle the code for the different concerns when weaving.

We investigated if AspectJ can be used to compose the different modules for an ATMS into this tangled code, as AspectJ seems to be the de-facto tool to perform aspectisation of transaction management. However, we found that the composition mechanisms in AspectJ are sorely lacking, which forces us to write code at the level of the aspect that is also tangled, a phenomenon that we name tangled aspect code.

Instead of writing tangled aspect code, it should be possible to define an ATMS as a composition of different modules, each module addressing one crosscutting concern, where merging these concerns yields the aspect code for the ATMS. In this dissertation we show how we achieve this, by taking into account the specific domain for which we are writing an aspect: ATMS, and using this domain information to create an aspect language for ATMS.

Before we detail our solution, however, we first investigate the run-time component of a transaction management system, the TP Monitor. This is required, as to be able to use a large variety of ATMS, we need this run-time component to provide support for these models.

Chapter 5

Transaction Management in Multi-Tier Distributed Systems

TP Monitors are like the Rolling Stones:
been around for a long time, but still drawing large crowds.
— **David Linthicum**

The goal and outline of the work proposed in this dissertation is to develop a means to modularize demarcation code for a wide variety of ATMS, so that an ATMS can be chosen, or a new ATMS can be constructed, that best fits the design of the application being built. In the previous chapter we have further explored this and found an extra requirement for the modularization of this demarcation code.

This chapter crosses over from demarcation code, and the application using ATMS, to the server that manages transactions at run-time, ensuring the properties of the ATMS being used are adhered to, i.e. concurrency control is performed for the application. This server is known as the transaction monitor, or TP Monitor, and will manage all transactions and their associated states, such as, e.g. locks.

In order to be able to use an ATMS in the client application, the TP Monitor will need to provide support for this ATMS, as it needs to ensure the properties of the model are adhered to. Consequently, allowing an application programmer to use a wide variety of ATMS, and even to implement new models, will require support from the TP Monitor.

Here we discuss how transaction monitors are used in the context of multi-tier distributed systems, and focus on one popular class of transaction monitors: those provided by the Enterprise JavaBeans component model. We investigate the ability

of these TP monitors to support a wide variety of ATMS, and we find this lacking. Also a significant feature of Enterprise JavaBeans is that it attempts to modularize demarcation code. However, we show that this turns out to be an inferior solution to the transaction aspects we discussed in the previous chapter.

5.1 Multi-Tier Distributed Systems

Current-day transaction monitors for large distributed systems, such as banking systems and airline ticket reservation systems, are meant to be used in a multi-tiered architecture. Therefore, to provide some relevant context, we discuss these architectures here. For these large systems the preferred architecture has evolved through the years, from a single mainframe application to 3-tier and multi-tier systems.

The term *tier* is used to describe how the architecture of a distributed system is logically partitioned across different clients and servers [Edw99]. In a *3-tier* architecture the partitioning is as follows:

- the first tier comprises clients that interact with the users,
- the second tier, also known as middle tier or application tier, contains servers that execute the business logic,
- and the third tier consists of the persistency layer, usually database servers.

Consider as example of a 3-tier system a high-traffic web shop:

1. The user interface tier consists of the web browser used by clients visiting the shop. Pages for the web browser are generated dynamically, based on stock inventory, and may contain forms to be filled out by the shopper.
2. The application tier consists of a web server, combined with a collection of programs implementing the business logic. Based on the pages requested by the shopper, or the data filled into a form, the web server calls the corresponding program, which processes the input and as a result returns a web page to be displayed by the web browser.
3. The resource tier is a relational database used by the applications in the previous tier, for example to query the amount of items available for a given product, to register an order, or to add the shoppers' information.

As a brief second example, consider an online bank: The user interface tier is a Java applet, which is downloaded to the customers' computer. This applet performs

Java RMI [Sun03b] calls to the application tier: a Java application, located on a central server. This application interacts with the bank's proprietary legacy information system that manages accounts: the third tier.

The application tier typically consists of a large body of software, which makes it difficult to maintain. To avoid this, separation of concerns, which we discussed in 3.1.1, is used and this tier is split up into a number of application components: each component is responsible for a limited amount of business functionality. Clients typically interact with multiple components during a business operation, and each of these components may, themselves, call upon other components to provide some functionality.

This 3-tier architecture, where the middle tier has been implemented as a collection of business components is often called *Multi-tier or N-tier*[Edw99], and provides a number of advantages over the 3-tier architecture [Edw99, Szy98]:

- Incremental development of the middle tier, by adding new components as needed, and the ability to incrementally upgrade this tier.
- One component can be reused by multiple applications.
- A component can be deployed on multiple servers of the application tier, for load-balancing.
- Off-the-shelf components can be bought and used, allowing a compromise between having the middle tier be fully custom-built and the middle tier consisting of solely off-the-shelf software.

Using such components, however, requires a specific runtime environment that will be able to run them. These runtime environments, which are also known as *middleware*, provide the 'glue' that sticks the components together to form the middle tier[Edw99]. Such middleware is responsible for:

Component communication: providing of inter-component communication capabilities, including support for network communications between components that are distributed over several servers.

Security: component security policies are enforced, ensuring no client or component requests services to which it has no rights.

State management: components are dynamically loaded from disk when required, and unloaded to disk to free up memory.

Transaction Management: transactions for the services implemented by the components are managed.

Database connectivity: a uniform API for a variety of databases may be exposed to the components.

In this section we have discussed the preferred architectures for large-scale distributed systems: 3-tier and multi-tier systems. In 3-tier systems the application is split up in a user interface tier, an application logic tier, and a database tier. In multi-tier systems, the middle tier is composed out of a number of business logic components, using middleware glue, which brings the advantages of component-based development to 3-tier systems.

Having discussed these two architectures, we are now able to describe the role of the transaction manager in these systems.

5.2 Transaction Monitors and Object Transaction Monitors

When considering multi-tier distributed systems, the middleware will provide a number of services to the components in the application tier. Of these services, one is of particular concern to us: transaction management. In this section we will first describe classical transaction monitors, and second detail how transaction monitors are integrated into the middleware as object transaction monitors.

5.2.1 Transaction Monitors

A *Transaction Monitor*, also known as Transaction Processing Monitor, or abbreviated to *TP monitor*, is a server application that performs concurrency control at runtime for its clients. To realise this, the TP monitor will manage all transactions and their attributes, delay reads and writes when necessary, and signal transaction aborts when needed, e.g. to break deadlocks.

Since interaction with shared data, e.g. in a database, entails management not only of that data, but also of the code, processes and memory required to make it available to the clients, it is common in transactional systems to call the database servers *resource managers*. The goal of a TP monitor is to make it easy to build software that invokes these resource managers, using an extended version of the *Remote Procedure Call* (RPC) [GR93].

The traditional RPC needs to be extended because transactional communications are an enhanced form of communications: transactions need to be started and ended, and data access needs to be associated with a transaction. To start and end transactions, the program code will include RPC's to the TP monitor, the canonical examples being `Begin_Work()` to start a transaction and `Commit_Work()` to commit it [GR93]. As we have already introduced in 2.1, explicitly declaring such transaction boundaries in the program code is also known as transaction demarcation [GR93].

Conceptually associated to the transaction is the *transaction context*. All code between the transaction boundaries, i.e. the code comprising the transaction, is said to be executed within this context. The transaction context can be referred to by using a transaction identifier, which is usually returned by the `Begin_Work()` RPC. *Transaction context propagation* entails passing this identifier as extra, possibly hidden, parameter to procedure calls, so that this code is also executed in that context. In other words, the code of these procedures is also included in the transaction. The inverse to this is *transaction suspension*: if we do not pass the transaction context to a procedure call, the work of this procedure is not included in the scope of the transaction. It is as if the transaction is suspended during the execution of the procedure, and resumed after the procedure has ended.

5.2.2 Object Transaction Monitors

Whereas previously, TP Monitors were built for use with procedural languages, current TP Monitors have grown to support and use object-oriented technology. These incarnations of the TP Monitor concept are known as an *Object Transaction Monitors* (OTM). The underlying technology that enables this transition to object technology is the *Object Request Broker* (ORB): we can consider OTM's as a combination of ORB's and TP Monitors [Edw99].

The ORB is a concept that is best known from CORBA [Vin97]: CORBA provides a standard for interoperability and portability that allows heterogeneous and open systems to integrate into one coherent system. Communication between the different systems, which are abstracted as a (collection of) object(s) is performed by means of remote method calls, which are facilitated by the ORB. The most important feature of the ORB is that it provides transparency for:

Object location: the location of the target object is not known to the invoker of the method, the target may reside at any point in the network.

Object implementation: the implementation language and operating environment of the target object is unknown to the invoker.

Object execution state: the ORB will transparently start or activate the object, i.e. load it in memory, if necessary, and stop or passivate the object, i.e. offload it to persistent storage, if it is no longer required.

Object communication mechanisms: the target of the method and the method invoker are unaware of the underlying network protocol used by the ORB to perform the remote method invocation.

Of these features, the last is of special relevance here, since it coincides with a task performed by traditional TP Monitors: communication management between client and server is now specialised into management of remote method invocations.

To these features, OTMs add, among other features, transaction management, resulting in a transaction monitor concept applicable to distributed objects.

These OTMs then, are ideally suited as middleware in multi-tier systems: we can write components as (a collection of) objects, and the OTM will perform all the services required of the middleware. Currently, there are three major industry standards for OTM's as middleware: Enterprise JavaBeans [Sun05], introduced in 1998 the CORBA 3.0 standard [Obj04], defined in 2002 and Microsoft's .net standard [Mic05]. In this dissertation, we have chosen Enterprise JavaBeans as the middleware for our experiments. We did not opt for .net because no development environment was available for our platform, and .net is geared specifically towards web services. In the choice between Enterprise JavaBeans and CORBA 3.0, we chose to use Enterprise JavaBeans because it is widely used, it provides similar capabilities to CORBA 3.0, with which it is compatible, and we consider it easier to work with.

5.2.3 Conclusion

In this section we introduced the software entities responsible for managing transactions at run-time: Transaction Monitors a.k.a. TP Monitors. TP Monitors perform concurrency control at runtime, according to the restrictions placed by the transaction model. We have also seen that to start and end transactions, the client software explicitly demarcates transactions in the program code, and includes the transaction identifier in every transactional call.

TP Monitors have been built for procedural programming languages, and to support object-oriented languages were modified into Object Transaction Monitors. Object Transaction Monitors are well-suited as middleware in n-tier distributed systems, and a number of industry standards have been developed. We have chosen to work with Enterprise JavaBeans, and therefore, in the next section, we will introduce Enterprise JavaBeans, paying special attention to its transaction management infrastructure.

5.3 Enterprise JavaBeans

As stated above, in multi-tier distributed systems, middleware glue is needed to provide a runtime environment for the different components that comprise the application layer. *Enterprise JavaBeans* (EJB) [Sun05] is the standard Java component architecture for such middleware applications, and is also CORBA compatible.

The EJB architecture defines a number of parts, which we discuss next:

Enterprise JavaBean: The complete EJB component that is deployed.

Deployment Descriptor: A declarative description of the attributes of an EJB.

Container: The runtime environment for one component.

Home interface: An interface for management of EJB instances.

Remote Interface: An interface declaring the EJBs' business methods made available to users of the EJB.

EJB object: An object representing an instance of a EJB.

First we describe the Enterprise JavaBean and its Deployment Descriptor, afterwards we describe the container, together with the Home and Remote interface and the EJB object. This is followed by a section on servers and clients, and we conclude with a discussion of transaction management.

5.3.1 Enterprise JavaBean and its Deployment Descriptor

The EJB is a component that implements 'business methods' i.e. the code that delivers the 'business' functionality. There are two distinct kinds of EJBs: session Beans and entity Beans [Sun05].

- *Session Beans* typically are non-persistent objects that implement some business logic, as Java methods, running on a server. Session Beans execute on behalf of a single client, they can update data in an underlying database, but do not directly represent this data. Session Beans only store conversational state between themselves and the client, they are relatively short-lived and are not recovered when the server crashes.
- An *Entity Bean* is a persistent object that represents an object view of some data stored in a database, or an entity that is implemented by an existing business application. Entity Beans can be shared by multiple users, are transactional, are long-lived (e.g. as long as the data exists in the database), and survive crashes of the EJB server.

Associated with an EJB is its *deployment descriptor*, the intent of which is twofold: first, the deployment descriptor declares the beans' environment properties, and second, it also allows the bean to be customized at deployment time. Deployment descriptors include all the declarative attributes associated with an enterprise Bean such as the type of bean, its name, the names of the home and remote interfaces, transaction specification, and method security attributes. This meta-data is used by the

Container when the EJB is deployed on the server to determine how the EJB should be managed.

Because this meta-data is separated out from the enterprise Bean class, it can be changed when an EJB is deployed. This allows the EJB deployer to adapt, for example, security or some transactional settings to the specific settings relevant for that deployment of the bean.

5.3.2 Container, EJB Objects, Remote Interface and Home Interface

Because an EJB is a middleware component, it cannot be used on its own, it must be contained within a *Container* [Sun05]. Containers contain and manage a number of EJB classes, i.e. instances of EJBs are created by the container, and the container mediates each access by the client to the EJB instances. Containers provide a standardized number of middleware services to the EJB, such as making the bean available to clients, performing transaction management, and handling activation and passivation.

The container exports the two following elements to possible clients: the EJB objects, through their Remote Interface, and the Home interface.

EJB objects are objects created by the container for a client, they implement the EJBs *Remote interface*, which declares the business methods that are callable by a client. A client never calls methods on an instance of a EJB, instead it calls methods on an EJB object exported by the container. The EJB object delegates the calls to the container, which eventually performs the corresponding method on the actual instance of the EJB. This construct ensures that each method call on the EJB is mediated by the container, which allows for service management by the container.

The *Home interface* is an interface provided by the developer of the EJB. This interface provides for creation, destruction, and lookup (only for entity EJBs) of EJBs.

Creation of EJB objects is achieved by calling one of the `create()` methods defined in the home interface. The arguments of this method are typically initialisers for the EJB object. The EJB class must implement an equal number of `ejbCreate()` methods, with signatures equal to the `create()` methods defined in the interface. When a `create()` method is called on the interface exported by the container, the equivalent `ejbCreate()` method will be called on the EJB class to create an EJB object that will be returned to the client.

5.3.3 Servers and Clients

Containers are located on servers that comprise the middle tier of the multi-tier architecture. The servers implement the necessary communication protocols between the client of the EJB and its container and between the container and the back-end

systems. These protocols primarily include RMI [Sun03b], and CORBA running over IIOP [Vin97].

The services provided by the EJB server can be called from an arbitrary number of clients, which might also be other EJBs. Clients locate EJBs by using the Java Naming and Directory Interface (JNDI) [Sun02] to look up the home interfaces of the EJB. It is the responsability of the container to make these classes available to the client through JNDI. Through the use of JNDI, the actual locations of an EJB class and an EJB container are transparent to the client.

Once a client has located a Home interface for a particular EJB class, it can use one of the instance creation methods defined in the interface to create an EJB object, to which it will receive a reference. The client can subsequently call the methods declared in the EJBs remote interface. Finally, the client can remove the EJB by using the remove methods defined in the EJBs home interface.

Note that all method invocations of the client to the Home interface and the EJB object are remote method invocations (either through RMI or CORBA). This combined with the lookup of EJBs through JNDI makes location and usage of an EJB location transparent. Recall that the abstraction of object location and communication are two defining features for an ORB, as we stated in 5.2.2.

5.3.4 Transactions

In EJB, transaction management is provided by the container, and this behavior is usually determined by the transaction attributes set in the EJB's deployment descriptor. For every business method of the Bean, the deployment descriptor declaratively states its transaction requirements. If a method is included in a transaction, all accesses to shared data in that method, i.e. reads and writes to entity beans, will be made transactional.

Table 5.1 details the possible transaction requirements that can be declared in the deployment descriptor and their meaning. If a transaction is created upon execution of a method, the Container will commit or rollback this transaction when the method ends. The decision to rollback the transaction is primarily based on exceptions thrown. If the method (or a nested method called by this method) throws a *system exception*: a `RuntimeException`, a `RemoteException`, or a subclass of these exceptions, the transaction will be rolled back, and the method will throw a `TransactionRolledbackException`. Also, a transactional method can mark itself for rollback by calling the `setRollbackOnly()` method on the Container, and determine its rollback flag by calling the `getRollbackOnly()` method on the Container. When a transactional method that is marked for rollback ends, the transaction will be rolled back but no exception will be thrown.

Defining transaction types in the deployment descriptor, and letting the Con-

Declared Attribute	No Trans.	Client Trans.
`NotSupported`	-	E
`Supports`	-	P
`Required`	NP	P
`RequiresNew`	NP	NP
`Mandatory`	F	P
`Never`	-	F

Legend:
E: Operations of this method are excluded from the transaction.
P: Transaction context is propagated to this method.
NP: A new transaction is started, enclosing the method.
This new transaction context is propagated to the method.
F: The method immediately fails and throws an exception.

Table 5.1: EJB deployment descriptor transaction declarations.

tainer take care of transaction creation and ending is known as *declarative transaction management* [MH01]. This is because we now only need to declare what kind of transaction support is required, in contrast to traditional transaction demarcation. As said in 5.2.1, with transaction demarcation, the method code also includes calls to the TP monitor to manage the transaction. Because all calls to an EJB Object are mediated by the Container, this Container can now transparently start and end transactions.

It is said that using declarative transaction management achieves a greater separation of concerns, resulting in cleaner business method code [MH01], allowing the method to be oblivious of its transactional behavior. Furthermore, the EJB specification [Sun05] claims the main advantage is that this allows the transactional behavior of the EJB to be modified without needing to change the implementation of the business logic. Therefore, the same Bean can be reused over different applications with different transactional requirements.

Note, however, that we can choose not to use declarative transaction management, and perform standard transaction demarcation. For this, the Container provides an API for explicit transaction management, similar to the transaction management API's exported by OTMs [MH01].

Having described the EJB architecture, and its transaction management, we will now study this transaction management in some more detail, and outline some critiques on this model.

5.3.5 Critiques on EJB Transactions

The EJB architecture introduces a new concept in the domain of transaction management: declarative transaction management, through the use of transaction attributes defined in the deployment descriptors. While this is a promising evolution in transaction management, we will now show that this new concept, as implemented in EJB, has a number of significant drawbacks.

There are three further critiques on this model that are worthwhile to make here: First we consider the usage of advanced transaction models, and the danger of defining transaction types in the deployment descriptor. Second we investigate the claim of separation of concerns. Third we detail an alternative implementation for EJB transaction management, supporting ATMS: Bourgogne transactions.

On Advanced Transaction Models and Changeable Transaction Attributes

As we specify transaction attributes per method, it seems obvious that some combinations of transactional method invocations, with their associated attributes, imply the usage of an advanced transaction model. For example, consider the `RequiresNew` demarcation attribute: it specifies that a new transaction must always be started by the container, and the method to run within the scope of that transaction. If we encounter the situation where the client is already in a transaction, it would seem logical that this new transaction is started as a sub-transaction of the running transaction. In other words, we have a natural mapping to nested transactions.

The EJB architecture, however, specifies that only flat transactions are used [Sun05]. Therefore, these natural mappings are disregarded, and an alternative interpretation is used: in this case the client transaction is simply suspended during the execution of the EJB's method. Instead, it would be worthwhile to allow these natural mappings.

According to the EJB specification [Sun05], defining transaction types in the deployment descriptor has an important advantage, unrelated to separation of concerns: Because the deployment descriptor can be modified at deployment time, by the person installing the EJB on the server, the transactional properties of the EJB can be tweaked. In other words, should the deployer decide that, for example, certain methods of an EJB whose property is `Required`, do in fact not need to be transactional, because this adds too much overhead, he may change the property to `Never`, eliminating the overhead.

Changing the transactional properties, however, is a tricky task that may "lead to incorrect programs" [KG02]. In our example above, this will lead to the throwing of an exception if the method is not executed within a transaction context. This behavior was not present in the EJB when developed. Therefore callers of this bean will most

probably not handle this exception, which results in erroneous application behavior. Furthermore, if we consider the semantics of the method, we have two more cases where things can go wrong: First, if the method at some time decides to rollback the transaction because some application constraints can not be met, this is impossible as the method is not included in a transaction at all. Second, if multiple instances of the EJB are concurrently executing, and the EJB relies on isolation between these instances, errors will occur.

In general, we can say that changing transaction policies should be avoided, and therefore this EJB feature is of no use. Kienzle and Guerraoui [KG02] aptly summarize this "feature" of EJB as follows "Based on our experience, changing the transaction policies [...] is highly error-prone. Only the implementor of the bean knows the exact semantics of the methods, and is qualified to select the appropriate policies. Allowing a different program to fiddle with these properties at deployment time will inevitably lead to incorrect programs."

Separation of Concerns?

As noted above, declarative transaction management is said to achieve a greater separation of concerns, resulting in cleaner business method code. This is important, as separation of concerns is a major factor in our choice to use AOP as a means to modularize ATMS demarcation code. If declarative transaction management provides a similar degree of separation of concerns, it should also be considered as a candidate to modularize ATMS demarcation code. However, if we further investigate how declarative transaction management is currently implemented in EJB, we see that separation of concerns is not really achieved, and we detail this here.

First, we recall the discussion in 3.2, where we specified that using transaction management in the application is only possible if the design of the application takes this into account. Because transactional methods may be rolled back, no side-effects may occur in these methods and the callers of these methods must be prepared to handle the consequences of such a rollback, i.e. make amends for the lost work. Separation of concerns of transaction management at the code level, which is what we discuss here, is therefore only possible if the concern of transaction management has also been taken into account at design time of the application. Consequently, we assume that the design does indeed take transaction management into account, and investigate if separation of concerns at the code level can truly be achieved as in 3.2.

Separation of Concerns: Deciding to Rollback

Consider commits and rollbacks: the decision to commit or rollback a transaction is made primarily based on exceptions thrown during method execution. If a system

exception is thrown, the transaction will be rolled back when the method ends, if not, the transaction will be committed. Also, the method may call the `getRollbackOnly()` and `setRollbackOnly()` methods on the container to obtain and set the rollback flag.

To call the above container methods, or to manually throw a system exception, however, breaks the separation of concerns aimed for by declarative transaction management. Since the method now contains code whose concern is to handle a section of the transaction, transaction management is not cleanly separated out. In other words: to cleanly use declarative transaction management, the method may never get or set its' rollback flag. Manipulating this flag not only taints the method with the transaction management concern, but also splits this concern in two disjoint parts. On the one hand, we have the declaration in the deployment descriptor that the method is transactional and, on the other hand, we have the java code within the method which manipulates the flag.

In contrast, with Aspect-Oriented Programming, as we discussed in chapter 3, it is possible to keep the concern of transaction management as one entity at the level of the code, by including manipulation of the rollback flag in the aspect definition. None of the work we discussed in 3.2 includes this, but nothing prohibits us from adding support for such a pointcut to the aspect.

Separation of Concerns: Handling a Rollback

Consider what should be done in case of a rollback. Conceptually, handling the rollbacks of transactions is a part of the concern of transaction management. Therefore, in order to have a clean separation at code level, such error-handling code should also be defined when stating the methods' transaction requirements, as we have discussed in 3.2. However, where we have been able to create an aspect that includes such error-handling, in EJB this is not the case. Instead, when a transaction is rolled back due to a system exception, this will throw a `TransactionRolledbackException` to the caller of the method, and when a transaction is rolled back due to the use of `setRollbackOnly()` the caller will not be informed of this in any way. So, when using the `setRollbackOnly()`, the method needs to additionally signal this to the caller by either returning an 'error' value or throwing an exception.

The above implies that handling a rollback can only be done from within the method caller's code. This means not only that the caller is now tainted with the error-handling part of transaction management, but also that transaction management is split up in three disjunct sections: the transaction declaration, manipulation of the rollback flag and handling of rollbacks. Furthermore, since the callers to the transactional method need to specify the error-handling code, this can lead to code duplication if there are multiple callers to the method.

Inadequate Separation of Concerns

Splitting up transaction management code in three parts when handling rollbacks is an important issue if we consider larger-scaled systems. In these systems, typically many transactions will execute concurrently, and therefore we have a significant chance of a deadlock. Also, we can expect the transactional system to be distributed over different machines, which implies a possibility for transactional system error, e.g. due to network failure. Furthermore, an advanced application will probably want, in some cases, to get and set the rollback flag to manually abort a running transaction, for example because some application constraints have not been met. In either of the above three situations, transaction management code will be spread out in three different places. Therefore we must state that the separation of concerns obtained by declarative transaction management, as currently implemented in EJB, is inadequate for larger-scaled systems. Note that this is an important drawback, as a goal of the EJB architecture is to provide support for these systems.

To conclude, declarative transaction management, as currently implemented in EJB, only provides a clean separation of concerns in the most trivial cases. This is when a transaction never rollbacks: no transactional system errors may occur, deadlocks may not be broken though a rollback, and the application itself may not decide to perform a rollback. In all non-trivial cases, declarative transaction management, as currently implemented in EJB, provides a worse separation of concerns than when using AOP. The transaction concern is forcefully split up in three distinct parts, in different sections of the application, whereas if we use AOP, we have the ability to keep this concern as one: inside the transactional method. Therefore, our original choice of using AOP to modularize ATMS demarcation code remains valid.

An alternative: Bourgogne Transactions

If we consider Table 5.1, we see that we have five different possible entries for each cell in these columns. As remarked by Procházaka [Pro01], the reason for this is that multiple criteria are folded into the "No Transaction" and "In Transaction" columns. In fact, for each of these two columns, three extra criteria are taken into account, as is shown in Table 5.2:

1. Is the client transaction propagated to the called method? (column 2 of table 5.2)

2. Is a new transaction started for this method? (column 3 of table 5.2)

3. Considering the client transaction, does the method fail and throw an exception? (column 4 of table 5.2)

Trans.	Prop.	New Trans.	Exceptions	Declared Attribute
No	-	No	In Trans.	`Never`
No	-	No	No Trans.	`Mandatory`
No	-	No	Never	`NotSupported`,`Supports`
No	-	Yes	In Trans.	$?_1$
No	-	Yes	No Trans.	`-`
No	-	Yes	Never	`Required`,`RequiresNew`
In	Yes	No	In Trans.	`-`
In	Yes	No	No Trans.	`Mandatory`
In	Yes	No	Never	`Supports`, `Required`,
In	Yes	Yes	In Trans.	`-`
In	Yes	Yes	No Trans.	$?_2$
In	Yes	Yes	Never	$?_3$
In	No	No	In Trans.	`-`
In	No	No	No Trans.	$?_4$
In	No	No	Never	`NotSupported`
In	No	Yes	In Trans.	`-`
In	No	Yes	No Trans.	$?_5$
In	No	Yes	Never	`RequiresNew`

Table 5.2: Analysis of table 5.1, copied from [Pro01].

If we structure the table according to the presence of a client transaction and the above criteria, instead of starting from the available declared attributes, we have as overview Table 5.2, as in [Pro01].

Remark that in this table we see that there are a number of locations for which no attribute is declared. Of these, the combinations resulting in cells marked with `-` can not occur, and the cells marked with `?` can, but no attribute has been defined. For example, the first question mark is the situation where, as a policy we start a new transaction for the method and throw an exception if the client is already in a transaction, and this occurs when the client is not in a transaction. This situation is indeed possible, but EJB does not allow us to specify this behavior. More exotic are the second and third question mark: while we are already in a client transaction, a new transaction is started and both contexts are propagated to the method, which is similar to nested transactions, as seen in 2.2. As we have said above, these kinds of transactions are not supported in EJB, which is why no attribute has been defined for these cases. As for the fourth and fifth question marks, it is not clear if these situations

make sense [Pro01].

A second remark is that, except for the `Never` attribute, all attributes are duplicated in the table. This is because all the other attributes are applicable both in the case where a client is in a transaction and when it is not.

NT Attribute	EJB Equivalent
`ThrowException`	`Mandatory`
`DoNothing`	`Supports, NotSupported`
`CreateNew`	`Required,RequiresNew`
CT Attribute	**EJB Equivalent**
`ThrowException`	`Never`
`Suspend`	`NotSupported`
`Propagate`	`Supports, Required, Mandatory`
`SuspendAndCreateNew`	`RequiresNew`
`Advanced`	`?`

Table 5.3: `NT` and `CT` attributes in Bourgogne Transactions.

Procházaka proposes an extension to the EJB transaction manager, called Bourgogne transactions [Pro01] that addresses the above remarks. In Bourgogne transactions, each method has to specify two transaction attributes: an `NT` and an `CT` attribute, which are shown in Table 5.3. The former specifies transactional behavior if the client is not in a transaction, and the latter if the client is in a transaction. This allows us to specify a superset of the EJB behavior: the first question mark in Table 5.2 is covered by declaring `NT = CreateNew` and `CT = ThrowException`.

In addition to the above, we can declare `CT = Advanced`, which implies that we suspend the running transaction and want to employ an advanced transaction model. Indeed, Bourgogne transactions contain support for using a fixed number of advanced transaction models. For this, six extra `CT` sub-attributes are defined that, according to their values declare resource delegation, and introduce a dependency between transactions. Such a *transaction dependency* is a relation between two transactions, linking the beginning or termination of one transaction to the beginning or ending of the other. We will discuss the concept of transaction dependencies in detail later, in 6.1.2. For now consider the following example: a "Begin On Commit" dependency implies that the second transaction will begin when the first commits. With resource delegation, which we return to in detail in 6.1.5, a transaction $T1$ delegates a number of objects to the other transaction $T2$, as we have seen in 2.5. Recall that, after the delegation, all the actions performed by $T1$ on the object will be considered as being performed by $T2$. Furthermore, $T2$ is now responsible for performing the commit or

abort of the object.

Although Bourgogne transactions are a powerful extension to the EJB transaction management infrastructure, there are still some downsides to this model:

- As declarative transaction management is performed on method boundaries, we can not start multiple transactions within one method and declare dependencies or component delegations between them.
- If we want to perform a kind of advanced transaction management, we are limited to those models that only use dependencies or delegation. This excludes models relying on semantic information such as those described in 2.4.
- Only a fixed number of advanced transaction models are supported, as only six `CT` sub-attributes are defined. Therefore, we are not able to provide support for new ATMS, or support for modifications to existing models, as they are not covered by these attributes.
- The error-handling is still the same as standard EJB error-handling, the downsides of which we discussed above. The only change here is that if an advanced transaction fails, after the rollback a system exception always will be thrown to the caller. This is in contrast to EJB transaction management, where, if a transaction indicates failure by setting the rollback flag on the Container, no exception is thrown when the transaction rolls back.

Because of these downsides, we could not use Bourgogne transactions in this dissertation, and had to develop our own solution, which we detail starting with the next chapter.

5.3.6 Conclusion

In this section, we introduced the Enterprise JavaBeans (EJB) architecture, the standard architecture for Java middleware applications, which we use in this thesis.

An EJB is the implementation of a business component. Associated with the EJB is its deployment descriptor, which declares properties such as the type of bean, its name, transactional properties and so on. Transaction management for EJBs is performed by the container, which provides the runtime environment in which EJB's reside. This management is steered by declarations placed in the EJB's deployment descriptor, a scheme which is called declarative transaction management.

Having discussed the architectural specification, we proceeded with studying the transaction management provided by the EJB architecture. We have seen that EJB has no support for advanced transaction models, instead simply limiting support to flat transactions. Also we have discussed the danger of allowing transaction attributes

to be modified at deployment time, which might lead to incorrect behavior of the component. We have seen that declarative transaction management, as implemented in the EJB architecture actually results in a worse separation of concerns than using traditional transaction demarcation. This rules out the use of declarative transaction management in this dissertation. Finally we discussed an existing alternative EJB transaction manager: Bourgogne transactions [Pro01], which does allow a fixed number of advanced transaction mechanisms. However, this has its drawbacks in that only a fixed number of ATMS are supported, and that it inherits the drawbacks inherent to EJB transaction management. As a result, Bourgogne transactions are also inadequate for this research.

5.4 Conclusion

In this chapter we talked about how transaction management is performed in multi-tier distributed systems. We have shown how the theoretical transaction models, introduced in the previous chapter, are enforced in practice: through a Transaction Processing Monitor (TP Monitor).

We first gave an overview of the architecture for the most common types of large distributed systems: multi-tier systems. In these systems the application logic has been implemented as a collection of business components.

Having seen this architecture, we discussed how the TP Monitor fits in this architecture. The TP Monitor is server software that will perform concurrency control according to the constraints imposed by the transaction model. We also detailed how an application interacts with the TP Monitor. Furthermore, we described how the TP Monitor is integrated into multi-tier architectures as an Object Transaction Monitor.

Finally, we have discussed the Enterprise JavaBeans architecture, which is the standard architecture for Java applications for the middle tier, and which we use further on in this text. We have discussed how transaction management is performed in this architecture. Although it is claimed that EJB provides for a larger degree of separation of concerns, we have shown this not to be the case. Combined with the ability to support only a few ATMS, this rules out the use of the TP Monitor for EJB in this dissertation.

To be able to allow for customizations of existing models and creation of new models, a TP Monitor must not only support a wide variety of existing ATMS, but must also be extensible in some way, and EJB is lacking in both respects. What we need therefore is a general TP Monitor for ATMS, which not only provides support for existing models but one which is extensible, so that new models can be supported. In the following chapter we present a formal model for ATMS, giving us a formal foundation on which to build such a TP Monitor.

Chapter 6

ACTA

> They're specialists, the whole lot of them [...] They limit their own scope to dig in the depths for details with more concentration. Modern research demands that every special branch shall dig in its own hole.
>
> — **Thor Heyerdahl, a sympathetic scientists' observation in "The Kon-Tiki Expedition"**

We have now seen a wide variety of Advanced Transaction Models, and also discussed TP Monitors and the available TP Monitors for EJB that support ATMS. We found that ATMS TP Monitors for EJBs are lacking, only supporting a small sub-set of possible ATMS.

If we want to be able to support a wide variety of ATMS and allow extension of the TP Monitor with new models, we need an analysis of all of these models, deconstructing them into primitive operations common to multiple models. If we base a TP Monitor on these primitives, we can ensure support for many ATMS through reconstruction of the required primitives, and support for new models through a new combination of these primitive operations, which we show later in the text, in 9.4. We find such an analysis and deconstruction in ACTA [CR91, CR92], which is an existing formal model, known within the ATMS community, that allows us to specify a large number of ATMS. In this chapter we discuss ACTA, starting with the formalism, then illustrating the use of ACTA by giving formal definitions of a number of ATMS, and we end with a note on implementation.

6.1 The Formalism

The ACTA formalism was created as a common framework in which it is possible to specify different ATMS. The goal of this work is to be able to reason about many ATMS, to infer the properties of the different models regarding recovery, visibility, consistency and permanence, which are concepts we discussed in 4.1.2. This allows for deduction of commonalities and variabilities between the different models, and synthesis of new models, by combining properties of different existing ATMS [CR91].

In ACTA, an ATMS is formally defined by stating a number of axioms that pertain to the transaction history. In this way, the formalism enables the specification of interactions between different transactions with regard to beginning and ending transactions, and the specification of the allowed operations of transactions on objects at a given point in time.

Note that ACTA allows us to specify ATMS which do not adhere to the classical concepts of consistency. ACTA does not provide any guarantees with regard to consistency of the data as a result from using an ATMS defined in ACTA. This is to be expected, since some ATMS forgo classical consistency in favor of semantically defined forms of consistency, as we discussed in 2.4.

In this section, we will introduce the ACTA formalism, starting with the fundamentals: the definition of events and history. To more easily reason about these fundamentals, a number of abstractions have been defined, which we will discuss from the second subsection on. The second subsection introduces dependencies between transactions, the third talks about conflicts, the fourth deals with view and access sets, and the fifth concludes with delegation.

6.1.1 Events and History

The fundamental premise of ACTA is the following: "The correctness properties of different transaction models can be expressed in terms of the properties of the histories produced by these models" [CR91]. Recall that in 2.1.1 and 2.1.2, we discussed schedules and how these relate to correctness of transaction execution. In ACTA such schedules are termed histories and to be able to reason about these histories, ACTA first defines the concepts of events and history, which we present here.

In ACTA, the transactions work on a database which is considered as a collection of objects. These objects behave similarly to the objects concept as known from object-oriented programming: each object contains an encapsulated state and is of a given type. This type defines a number of operations that can be invoked on an object. Invoking an operation on an object always produces a return value (which might be void), and this value is dependent on the state of the object. Furthermore, there are two operations defined on every object: Commit and Abort, which we will discuss

later.

Transactions access and manipulate the database by invoking operations on these objects. Such operation invocations are called *events*, and are subdivided into two kinds: object events and significant events. Object events are the invocation of an operation by a transaction, and significant events are the invocation of transaction management primitives. We discuss both in more detail next.

Definition 1. Operation invocations of an operation p, within a transaction Ti, on an object *ob* are called *object events* and denoted as $p_{Ti}[ob]$.

The effects of p, are not automatically persisted in the database, this only happens when the $Commit_{Ti}[ob.p]$ object event occurs. Conversely, the effects of p are destroyed when the $Abort_{Ti}[ob.p]$ object event occurs. Invoked operations which are not yet committed or aborted are known as *ongoing operations*. The straightforward use of the commit and abort operations is to commit or abort a transaction when it has ended, by committing or aborting all the operations. Furthermore, since commit and abort are defined at the level of operations, and not at the level of objects, this makes it possible for an ATMS to selectively commit or abort parts of the performed operations within a transaction.

Definition 2. A *significant event* occurs when a transaction management primitive is invoked. Each ATMS defines the possible significant events that can be invoked by a transaction Ti, denoted SE_{Ti}.

The list of significant events for classical transactions consists of the events begin, commit. and abort. Recall that in chapter 2 we have seen that an advanced transaction model can define new kinds of primitives, such as the breakpoint in 2.4 and the split in 2.5.

Both object events and significant events are logged in the sequence in which they were issued, which is analogous to the concept of schedules, which we detailed in 2.1.1. ACTA then reasons about such a *history* H of events of a concurrent execution of a set of transactions on a database of objects. An interesting projection of H is the history H_{ob} of an object *ob* that indicates the order of execution of the different operations on *ob*. The state of *ob*, after executing a number of transactions, then equals the state produced by applying H_{ob} to the initial state of *ob*.

To define an ATMS, a number of *axioms* are usually formally stated which declare the properties to which the resulting histories adhere [CR91, CR92]. Such properties of histories are defined by constraints on the events in these histories, which fall in three different groups:

1. An event ϵ' can be constrained to only occur after an event ϵ, denoted by $\epsilon \rightarrow \epsilon'$

2. An event ϵ can only occur in a history H if a condition $Cond$ is satisfied, denoted by $(\epsilon \in H) \Rightarrow Cond_H$

3. An event ϵ has to occur in a history H if a condition $Cond$ is satisfied, denoted by $Cond_H \Rightarrow (\epsilon \in H)$

Using these three kinds of constraints we are able to formally define the advanced transaction models by specifying the histories which are allowed in a list of axioms. However, although we are now able to formally specify many ATMS, there is an important drawback regarding size and complexity of these specifications. Using the above constraints, specifications are overly large and very complex, and furthermore many constrains of several distinct kinds are to be found in different specifications. To address these drawbacks, i.e. to ease the axiomatic specification of an ATMS, a number of abstractions have been defined in the papers [CR91, CR92], and we will review these next, starting with dependencies.

6.1.2 Dependencies

If we consider the behavior of multiple, possibly concurrent, transactions in the different ATMS, we see that some types of constraints between the significant events of these transactions are common to many different models. To facilitate reasoning about these constraints, a first abstraction was created: *dependencies*. Twelve kinds of dependencies have been defined in [CR91, CR92], all of which are constraints resulting from the structure of the transactions defined by the ATMS. Each of these *structural dependencies* places a relationship between two transactions, defined in terms of the significant events of these transactions. We will not detail all twelve dependencies here, instead we pick five of them which are relevant to this dissertation:

Commit Dependency $(Tj\ \mathcal{CD}\ Ti)$ If Ti and Tj commit, Ti must commit before Tj, or formally: $(Commit_{Tj} \in H) \Rightarrow ((Commit_{Ti} \in H) \Rightarrow (Commit_{Ti} \rightarrow Commit_{Tj}))$

Weak-Abort Dependency $(Tj\ \mathcal{WD}\ Ti)$ If Ti aborts and Tj has not yet committed, then Tj must abort, or formally: $((Abort_{Ti} \in H) \wedge \neg(Commit_{Tj} \rightarrow Abort_{Ti})) \Rightarrow (Abort_{Tj} \in H)$

Begin-on-Commit Dependency $(Tj\ \mathcal{BCD}\ Ti)$ Tj cannot begin executing until Ti commits, or formally: $(Begin_{Tj} \in H) \Rightarrow (Commit_{Ti} \rightarrow Begin_{Tj}))$

Begin-on-Abort Dependency $(Tj\ \mathcal{BAD}\ Ti)$ Tj cannot begin executing until Ti aborts, or formally: $(Begin_{Tj} \in H) \Rightarrow (Abort_{Ti} \rightarrow Begin_{Tj}))$

Compensation Dependency ($Tj\ \mathcal{CMD}\ Ti$) If Ti aborts, Tj must commit, or formally: $(Abort_{Ti} \in H) \Rightarrow (Commit_{Tj} \in H)$

The commit and weak-abort dependencies are used in nested transactions: Firstly, a parent transaction Tp has a commit dependency on each of its children Tc, so that Tp does not commit before Tc terminates or, vice versa, Tc aborts if Tp commits before Tc terminates. Secondly, a child transaction Tc has a weak-abort dependency on its parent Tp which guarantees that if Tp aborts and Tc has not yet terminated, Tc will abort. Similarly, compensating transactions are formalized by placing begin-on-commit, begin-on-abort and compensation dependencies between the transaction, the compensating transaction and the overall action that has to be undone, as we detail in 6.2.5.

Definition 3. The dependencies that are formed as transactions are executed, are kept in a *dependency set*, denoted $DepSet$.

$DepSet$ is relative to a history H. Of particular interest is the current dependency set, $DepSet_{ct}$ that is relative to the current history, i.e. all events that have occured until this given point in time.

After considering how the structure of transactions within an ATMS gives rise to structural dependencies, we now see how the run-time behavior of different transactions over a shared object leads to a different kind of constraints: conflict relationships.

6.1.3 Conflicts

In addition to structural dependencies, ACTA also defines *behavioral dependencies*, which are caused by the run-time behavior of different transactions over a shared object. Consider, for example, what can happen if we were to use a transaction manager which does not use locks for concurrency management (such as an optimistic concurrency strategy). If first a write access of a value $Wi(x)$ occurs and then a read access $Rj(x)$ occurs in a different transaction, the concurrency manager should introduce an abort dependency between Ti and Tj so that if Ti aborts, Tj should also abort because it has used the value of x which is no longer correct.

More in general, the effects of transactions on objects are formalized in the form of conflict relationships, which we will discuss here.

The basic building block for formalizing conflicts is the conflict relationship between two operations p and q, which is similar to the notion of conflicts we introduced in 2.1.2.

Definition 4. Operations p and q on object *ob conflict* if their effects on the state of *ob*, or their return values, are dependent on the order in which they are executed. Also, two operations that do not conflict are said to be *compatible*.

Since operations have a return value, we can further refine the conflict relationship, taking into account the return values of operations. This leads to *return-value dependent* and *return-value independent* relationships between operations.

Definition 5. Given a conflict relationship between p and q on ob, on a history H, noted $conflict(H_{ob}, p, q)$, $return-value\ independent(H_{ob}, p, q)$ is true if the return value of q is the same wether or not p precedes q in H_{ob}. If this is not the case, then q is return-value dependent on p, noted as $return-value\ dependent(H_{ob}, p, q)$.

Note that, since H_{ob} can usually be derived from the context, in [CR91], where possible, this argument is dropped from *compatible*, $return-value\ dependent$ and $return-value\ independent$ and we do the same.

Returning to 2.1.1 and 2.1.2, we formalize how the ACTA definition of conflicts relates to serializability by inducing dependencies between operations. Considering a sequence of conflicting operations $p \rightarrow q$ in H (where $conflict(p, q)$ is true), and p is invoked within Ti, while q is invoked within Tj, two kinds of dependencies can be introduced, depending on $return-value\ dependent(p, q)$:

1. If $return-value\ dependent(p, q)$ is true, Tj must abort q if Ti aborts p(which is a refinement of the example of the use of the abort dependency we have given above). Formally this is stated as: $(return-value\ dependent(p, q) \wedge (p_{Ti}[ob] \rightarrow q_{Tj}[ob])) \Rightarrow ((Abort_{Ti}[ob.p] \in H) \Rightarrow (Abort_{Tj}[ob.q] \in H))$

2. If $return-value\ independent(p, q)$ is true, Tj cannot commit q until Ti commits or aborts (because Ti must be serialized before Tj). Formally this is stated as: $(conflict(p, q) \wedge return-value\ independent(p, q) \wedge (p_{Ti}[ob] \rightarrow q_{Tj}[ob])) \Rightarrow ((Commit_{Ti}[ob.p] \in H) \Rightarrow ((Commit_{Tj}[ob.q] \in H) \Rightarrow (Commit_{Ti}[ob.p] \rightarrow Commit_{Tj}[ob.q])))$

To enforce serializability, a concurrency manager ensures that no errors occur by either keeping track of dependencies as they are formed and guarantee that these constraints are met, or by ensuring that they are never formed. The former is the approach taken by the optimistic concurrency schemes, where at commit time dependencies are verified which may lead transactions to abort. For the latter, avoiding dependencies being formed in case of conflicts, we can either immediately abort Tj, or wait until p terminates, which removes the conflict. This last strategy essentially boils down to the usage of locks, which we have discussed in 2.1.2.

In ACTA, all conflict relationships are written down in the *generalized conflict relationship* form: $(p_{Ti}[ob] \rightarrow q_{Tj}[ob]) \Rightarrow Condition_H$, where $Condition_H$ typically is a dependency between Ti and Tj. Therefore, in the remainder of the text we shall also use this notation.

Straightforward examples of $Condition_H$ are the dependencies introduced in 6.1.2. Note that nothing precludes $Condition_H$ to also use other significant events, such as, for example, the split in 2.5 or even object events. This leads to great flexibility in the definition of conflicts, which allows us to formalize and tailor many different types of concurrency control [CR91].

One extra condition that recurs often is the case in which isolation between different transactions is not strict, and therefore an apparent conflict relationship between two transactions turns out not to be formed. To treat this, view and access sets have been defined, which we discuss next.

6.1.4 View and Access

In a number of ATMS, isolation between different transactions is relaxed and it is possible for one running transaction Ti to see the results of another transaction Tj while Tj is still executing. This *visibility* of transactions is formalized in ACTA using two sets: the view set and the access set.

The *view set* is a set which contains all the objects which are potentially visible to the transaction and the *access set* contains all the objects which have been accessed or created by the transaction.

Definition 6. $AccessSet_{Ti} = \{ob \mid \exists p\ (p_{Ti}[ob] \in H)\}$

Definition 7. $ViewSet_{Ti} = AccessSet_{Ti} \cup \{\cup AccessSet_{Tj} \mid Tj \in Tset_{Ti}\} \cup DB$ where the contents of $Tset_{Ti}$ are defined per ATMS.

If we examine the view set in some more detail, we see that the view set of Ti is constituted of different other sets: first, the access set of Ti, second, the access sets of all the transactions which are visible to Ti, and third, the database. While the inclusion of the database might seem to render the other sets in this relationship redundant, since all objects in the access sets are also contained in the database, this is not the case. Consider for example an object $ob \in DB$ that has been modified by a visible transaction Tj. The state of this object visible to Ti will be the state as modified by Tj, and not the state in DB which is older because Tj has not yet committed. For example, consider nested transactions: a child transaction Tc can see all of the intermediate work performed by its parent Tp, recursively all the way up to the root transaction Tr. We formalize this by defining $Tset_{Tc} = \{Tp,\ \ldots,\ Tr\}$.

As visibility between transactions allows different transactions to concurrently work on the same objects as if they were one and the same transaction, this implies that conflicts between these transactions should not be taken into account. This means that the conflict relationships from the previous section should be reconsidered, and we should take the view set into account.

Definition 8. For operations $(p_{Ti} \rightarrow q_{Tj})$, dependency relationships between Ti and Tj specified by a conflict condition $Condition_H$ should only be induced if $Ti \notin Tset_{Tj}$, i.e. if there is no visibility from Tj to Ti. Formally, we write the new generalized conflict relationship as follows: $((p_{Ti}[ob] \rightarrow q_{Tj}[ob]) \wedge \neg(Ti \in Tset_{Tj})) \Rightarrow Condition_H$.

In our nested transaction example, an operation performed by Tc on an object *ob* should not conflict with any operations performed on *ob* by any of its parent transactions $Tp, \ \ldots, \ Tr$.

Also, we should note that when considering the view set, we have assumed that a transaction Ti would always want to see the most recent changes to an object made by a transaction $Tj \in Tset_{Ti}$. However, in some ATMS this may not be the case and a conflict at a deeper level in the tree is ignored if at a higher level it does not produce a conflict. To enable this, an explicit sequence of traversal of the elements of $Tset_{Ti}$ is taken into account by the definition of an *order of conflict checks* relation *AccessOrder* for the ATMS. In the formal definitions of the ATMS we encountered, we have however not encountered a real use of this access order, and therefore, we do not elaborate further on it here.

6.1.5 Delegation

A final feature of ATMS that needs to be considered is delegation, such as, for example, in nested transactions: If a child transaction Tc commits, its effects are not written to the database but are instead delegated to its parent Tp, and from there on the system behaves as if the work was performed by Tp. Only when the root transaction commits, the effects of Tc will be made permanent (unless Tp or any of its parents have aborted, in which cases the work of Tc is lost).

In ACTA, this behavior is captured by the *delegate* operation, where one transaction Ti delegates the responsibility for committing or aborting a specified number of its objects $ob \in DelegateSet(Ti, Tj)$, to another transaction Tj[1]. Note that delegation is defined on objects and not on operations, contrary to the other features of ACTA. Delegation is realized by rewriting the history H so that all operations of Ti on *ob* are regarded as having been performed by Tj. This has as a consequence that all dependencies toward Ti, induced by operations $p[ob]$, where $ob \in DelegateSet(Ti, Tj)$, will now no longer be toward Ti, but toward Tj. Furthermore, all these operations will be removed from $AccessSet_{Ti}$ and placed in $AccessSet_{Tj}$.

In our nested transactions example, when committing the child Tc will delegate all the objects in its access set to its patent Tp, i.e. $DelegateSet(Tc, Tp) = AccessSet_{Tc}$.

We now have seen all fundamental concepts of ACTA, and we can proceed with specifying the properties of a number of advanced transaction models.

[1] Where, of course, $ob \in AccessSet_{Ti}$

6.1.6 Conclusion

In the preceding subsections, we have introduced the ACTA formalism, that allows for specification of the properties of ATMS by placing constraints on transaction histories.

Such transaction histories consist of operations on objects, termed events, and transaction primitive invocations, called significant events. To define an ATMS in ACTA, a number of axioms are stated to which such a transaction history must comply. Using simple constraints as axioms, however, leads to overly large and very complex specifications, with many repetitions. To facilitate the definition of an ATMS, a number of abstractions have been created.

Constraints between the significant events of different transactions that occur in multiple ATMS have been abstracted into dependencies. Conflicting operations on objects and isolation between different transactions also introduce constraints that can be weakened through the use of view and access sets.

Transaction histories can also, virtually, be re-written through the delegation operation, which delegates a source transactions' responsibility of the operations on a number of objects to another, destination, transaction. The result is as if the destination transaction has performed these operations, and not the source transaction. This implies that constrains formerly placed on the source transaction due to these operations will now be placed on the destination transaction.

Using the above ACTA concepts of dependencies, view and access sets and delegation, we can now formally define ATMS, by stating a number of axioms for transaction histories, as we see in the next section.

6.2 Formal definitions of ATMS

Using all the elements we introduced above, we are now able to proceed with formally defining a number of transaction mechanisms. As we have said, in ACTA, the specification of an ATMS is performed by stating a number of axioms that relate to the transaction's history. These axioms will use the abstractions we defined above: dependencies, conflicts, view and access sets, and delegation. Note that we will not concern ourselves here with proving correctness of the ATMS specifications, instead we refer to [CR91, CR92] for more information regarding the correctness proofs.

In the first and second section, we will define two properties which are common to many ATMS: the object properties of serializability and correctness and the transaction property of failure atomicity. In section three, we will present the first formal definition, not strictly of an ATMS, but of Atomic Transactions. Section four will discuss the formal definition of the Nested Transactions ATMS. Finally we will excerpt some axioms from the definition of Sagas in section five, to demonstrate transaction sequencing and the use of compensating transactions.

6.2.1 Serializability and Correctness

Before we formally define the properties which must be met by transaction mechanisms, we first turn our attention to the objects which are contained in the database. Multiple ATMS expect these objects to have the serializabilty and correctness properties, and therefore we discuss them here.

Definition 9. Given a history H, operations p and q, transactions Ti and Tj and an object ob, $\mathcal{C}$ is a conflict relation defined as follows:
$Ti\ \mathcal{C}\ Tj \Rightarrow \exists ob, \exists p, q, (conflict(p, q) \wedge (p_{Ti}[ob] \rightarrow q_{Tj}[ob]))$

Definition 10. The transitive closure of $\mathcal{C}$ is denoted $\mathcal{C}^*$: $Ti\ \mathcal{C}^*\ Tk \Rightarrow Ti\ \mathcal{C}\ Tk \vee (\exists Tj, (Ti\ \mathcal{C}^*\ Tj \wedge Tj\ \mathcal{C}^*\ Tk)$

Definition 11. A set of transactions T is serializable $\Leftrightarrow \forall Ti \in T \neg(Ti\ \mathcal{C}^*\ Ti)$ i.e. if no $\mathcal{C}$ cycles are present.

Using the above, we can define when an object *behaves serializably*: this can be ensured if we can define a commit order between different transactions that invoke conflicting operations on this object, without forming cycles in the conflict relationship.

Definition 12. *ob* behaves serializably $\Leftrightarrow$

1. $\forall Ti, Tj\ Ti \neq Tj\ \forall p, q(conflict(p, q) \wedge (p_{Ti}[ob] \rightarrow q_{Tj}[ob])) \Rightarrow$
   $((Commit_{Tj}[ob.q] \in H_{ob}) \Rightarrow$
   $((Commit_{Ti}[ob.p] \in H_{ob}) \rightarrow (Commit_{Ti}[ob.p] \rightarrow Commit_{Tj}[ob.q])))$
2. $\forall Ti, \forall p(Commit_{Ti}[ob.p] \in H_{ob}) \rightarrow \neg(Ti\ \mathcal{C}^*\ Ti)$

Note that the above definitions harken back to 2.1.2 where we introduced conflict-serializability, as a stricter form of serializability, and indeed is what is formally redefined here.

A second object behavior defined in ACTA is correctness: an object that *behaves correctly* ensures that when an operation p aborts, any return-value dependent operation q that follows p is also aborted.

Definition 13. *ob* behaves correctly $\Leftrightarrow \forall Ti, Tj\ Ti \neq Tj\ \forall p, q,$
$(return - value\ dependent(p, q) \wedge (p_{Ti}[ob] \rightarrow q_{Tj}[ob])) \Rightarrow$
$((Abort_{Ti}[ob.p] \in H_{ob}) \Rightarrow ((Abort_{Tj}[ob.q] \in H_{ob}))$.

In ACTA, objects that behave serializably and correctly are called *atomic objects*. Note that this atomicity property is at the level of objects, and not at the level of transactions. Atomicity at the level of transactions is called failure atomicity in ACTA, and we discuss this next.

6.2.2 Failure Atomicity

In [CR91], the term *failure atomicity* is used for what we know as the atomicity property of classical transactions. In other words, failure atomicity means that either all or none of the operations of a transaction are executed.

Definition 14. Ti is failure atomic $\Rightarrow$

1. $\exists ob(\exists q\ Commit_{Ti}[ob.q] \in H) \Rightarrow (Commit_{Ti} \in H)$
2. $(Commit_{Ti} \in H) \Leftrightarrow \forall ob \forall q((q_{Ti}[ob] \in H) \Rightarrow (Commit_{Ti}[ob.q] \in H))$
3. $\exists ob(\exists q\ Abort_{Ti}[ob.q] \in H) \Rightarrow (Abort_{Ti} \in H)$
4. $(Abort_{Ti} \in H) \Leftrightarrow \forall ob \forall q((q_{Ti}[ob] \in H) \Rightarrow (Abort_{Ti}[ob.q] \in H))$

In the above definition, (1) and (2) state, respectively, that if one operation commits, the transaction must commit, and that the transaction commits if and only if all operations in the transaction commit. In other words, if one operation commits, the transaction must commit, and therefore, all operations must commit. (3) and (4) are analogous to (1) and (2), showing that the same rules apply to operation and transaction abort.

Having seen serializabilty and failure atomicity, we can now proceed with the first formal definition of the classic transaction model: atomic transactions.

6.2.3 Atomic Transactions

Before showing an ACTA definition of an ATMS, it makes sense to first consider the base level, as it were, which are atomic transactions. Atomic transactions, i.e. the classical transaction model, is defined in ACTA using 8 axioms for an atomic transaction Ti:

1. $ES_{Ti} = \{Begin, Commit, Abort\}$
2. $(Begin_{Ti} \in H) \Rightarrow (\neg(Commit_{Ti} \rightarrow Begin_{Ti}) \wedge$
   $(\neg Abort_{Ti} \rightarrow Begin_{Ti}) \wedge \neg(Begin_{Ti} \rightarrow Begin_{Ti}))$
3. $(Commit_{Ti} \in H) \Rightarrow ((Begin_{Ti} \rightarrow Commit_{Ti}) \wedge \neg(Abort_{Ti} \in H))$
4. $(Abort_{Ti} \in H) \Rightarrow ((Begin_{Ti} \rightarrow Abort_{Ti}) \wedge \neg(Commit_{Ti} \in H))$
5. $(Commit_{Ti} \in H) \Rightarrow \neg(Ti\ \mathcal{C}^*\ Ti)$
6. $(Begin_{Ti} \in H) \Rightarrow ((ViewSet_{Ti} = AccessSet_{Ti} \cup DB) \wedge$
   $(AccessOrder_{Ti} = (AccessSet_{Ti}, DB)))$

7. Ti is failure atomic

8. $\forall ob, (\exists p, p_{Ti}[ob] \in H) \Rightarrow (ob$ is atomic)

First, the significant events are defined in (1). (2) states that a transaction must begin before it commits or aborts, and that it may only begin once. In (3) and (4) constraints are placed on commit and abort: the transaction must have started and it must either commit or abort. (5) states that a transaction can not commit if it forms a $\mathcal{C}$ cycle with itself. The view of T is defined in (6) as first the Access set, and then the database. Failure atomicity is specified in (7), while object atomicity is specified in (8).

This specification is quite straightforward, with simple axioms, a trivial view set and no use of delegation. In the next section, we will discuss a more complex ACTA definition for the example ATMS we have used frequently in this chapter: nested transactions.

6.2.4 Nested Transactions

In this section we give an example definition of an ATMS, as given in [CR91], to show the size and form such definitions can have. As an example we have chosen Nested Transactions, because it uses dependencies, non-trivially defines the view and access set, uses delegation, and still remains conceptually quite simple.

The formal definition of nested transactions, as given in [CR91] is somewhat longer than atomic transactions, with nineteen axioms instead of eight. Luckily, however, the first eight axioms are similar or nearly identical to the axioms for atomic transactions, as can be seen in the following formal definition, where Tr is the root transaction, Tp a parent, and Tc a child transaction (i.e. a nested transaction).

1. $ES_{Tp} = \{Begin, Spawn, Commit, Abort\}$

2. $(Begin_{Tr} \in H) \Rightarrow (\neg(Commit_{Tr} \rightarrow Begin_{Tr}) \wedge$
   $(\neg Abort_{Tr} \rightarrow Begin_{Tr}) \wedge \neg(Begin_{Tr} \rightarrow Begin_{Tr}))$

3. $(Commit_{Tr} \in H) \Rightarrow ((Begin_{Tr} \rightarrow Commit_{Tr}) \wedge \neg(Abort_{Tr} \in H))$

4. $(Abort_{Tr} \in H) \Rightarrow ((Begin_{Tr} \rightarrow Abort_{Tr}) \wedge \neg(Commit_{Tr} \in H))$

5. $(Commit_{Tr} \in H) \Rightarrow \neg(Tr\ \mathcal{C}^*\ Tr)$

6. $(Begin_{Tr} \in H) \Rightarrow ((ViewSet_{Tr} = AccessSet_{Tr} \cup DB) \wedge$
   $(AccessOrder_{Tr} = (AccessSet_{Tr}, DB)))$

7. Tr is failure atomic

8. $\forall T, T = Tr \vee T = Tc \forall ob(\exists p, p_T[ob] \in H) \Rightarrow (ob \text{ is atomic})$

9. $ES_{Tc} = \{Spawn, Commit, Abort\}$

10. $(Spawn_{Tp}[Tc] \in H) \Rightarrow (\neg(Commit_{Tp} \rightarrow Spawn_{Tp}[Tc]) \wedge$
$\neg(Abort_{Tp} \rightarrow Spawn_{Tp}[Tc]))$

11. $(Commit_{Tc} \in H) \Rightarrow ((Spawn_{Tp}[Tc] \rightarrow Commit_{Tc}) \wedge$
$\neg(Abort_{Tp}[Tc] \rightarrow Commit_{Tc}))$

12. $(Abort_{Tc} \in H) \Rightarrow ((Spawn_{Tp}[Tc] \rightarrow Abort_{Tc}) \wedge$
$\neg(Commit_{Tc} \rightarrow Abort_{Tc}))$

13. $(Spawn_{Tp}[Tc] \in H) \Leftrightarrow ((Tc\ \mathcal{WD}\ Tp) \wedge (Tp\ \mathcal{CD}\ Tc))$

14. $((Commit_{Tc} \in H) \Leftrightarrow (Delegate_{Tc}[Tp] \in H)) \wedge$
$(DelegateSet(Tc, Tp) = AccessSet_{Tc}))$

15. $\exists ob(\exists q\ Abort_{Tc}[ob.q] \in H) \Rightarrow (Abort_{Tc} \in H))$

16. $(Abort_{Tc} \in H) \Rightarrow \forall ob \forall q((q_{Tc}[ob] \in H) \Rightarrow (Abort_{Tc}[ob.q] \in H))$

17. $(Spawn_{Tp}[Tc] \in H) \Rightarrow ((ViewSet_{Tc} = \{\cup AccessSet_{Ti} | Ti \in Tset_{Tc}\} \cup DB) \wedge$
$(Tset_{Tc} = \{Tn, Tn-1, Tn-2, \ldots, T0\}))$
where $Tn = Tc, Ti\ \mathcal{WD}\ Ti - 1$

18. $\forall T_n, t,\ t\mathcal{WD}^*\ Tn, \forall ob \forall p(p_t[ob] \in H) \Rightarrow$
$\nexists((p_T[ob] \rightarrow q_{Tn}[ob]) \wedge conflict(p, q))$

19. $(Commit_{Tc} \in H) \neg(Tc\ \mathcal{C}^*\ Tc)$

In the above definition, the first eight axioms define the root transaction as an atomic transaction, with one extra significant event: the spawn event, which spawns a child transaction. The remaining axioms define the properties of the child transactions, which we will discuss next.

The significant events of children, defined in (9), do not include a *Begin* because child transactions are spawned by a parent. Also, the *Spawn* defined here will start a new child of the current transaction. Because child transactions do not start with a begin, axiom (10) redefines (2) for child transactions, using *Spawn* from the parent instead of *Begin* as the start of the child transaction: A child transaction may not be spawned by the parent after it has committed or aborted. Similarly, (11) and (12) redefine (3) and (4) for child transactions: a child transaction that commits or aborts must have been spawned by a parent first, and the child either commits or aborts.

In (13) the structural dependencies between parent and child are defined: The child has a weak-abort dependency on its parent, so that if the parent aborts and the child has not yet terminated, the child will also abort. Also, the parent has a commit dependency on the child, so that it will not commit before the child has terminated. Note that specifying the presence of a weak-abort dependency between transactions suffices to indicate that one is a child transaction of the other. Therefore, in the remaining definitions, the weak-abort dependency is used to identify parent-child relationships.

Delegation of the resources accessed by a child, if it commits, is defined in (14), by identifying the parent as delegatee, and the $DelegateSet$ as the $AccessSet$. Strangely, this definition does not include any temporal information: according to this definition, delegation may occur at any time within the transaction, or even before or after the child transaction has started or ended! This is probably a simple oversight by the authors of [CR91], and assume that the intent here is for delegation to happen at commit time.

The abort of a child transaction is defined in (15) and (16), which are essentially the same axioms as conditions (3) and (4) of failure atomicity. The first two conditions of failure atomicity are not included here, because at commit time of a child, the operations are not committed but delegated to the parent. Therefore we can not simply repeat axiom (7) for child transactions.

Child transactions see all the intermediate work performed by parent transactions, which is defined in (17), where $Tset_{Tc}$ includes all parents, up to the root. No $AccessOrder$ is defined, which means that the elements of $Tset_{Tc}$ may be traversed in any order, i.e. Tc sees the most recent changes made to an object.

Axiom (18) prevents the occurrence of cases where a child transaction causes the abortion of its parent, or any of its ancestors. Consider the following: if p and q conflict, and Tp invokes q after Tc invokes p, then Tp would be abort-dependent on Tc, because otherwise it could keep on working with data that has become invalid. With this dependency in place, whenever Tc aborts, Tp would also need to abort. This does not comply with the description of nested transactions, therefore such abort dependencies may not be formed, which is achieved by (18). Note that $\mathcal{WD}^*$ is nowhere defined in [CR91], but we assume that it is analogous to the definition of $\mathcal{C}^*$.

Finally, serializabilty for child transactions is specified in (19) by disallowing $\mathcal{C}$ cycles, analogous to (5).

We have now shown and discussed the, quite extensive, formal ACTA definition of nested transactions. This definition demonstrated the use of dependencies, which link parent and child transaction, and delegation at commit time of child transactions. Furthermore, view and access sets were non-trivial, in contrast to the definition of atomic transactions.

6.2.5 Transaction Ordering and Compensating Transactions

Two common techniques of different ATMS are specifying an order or sequence of transactions, and the use of compensating transactions to perform a semantic undo. In this section we will show how these two elements can be formalized in ACTA by placing dependencies.

A good example of the use of these two techniques are Sagas, which we talked about in 2.3. Recall that a Saga is a sequence of atomic transactions, where each transaction is also associated with a compensating transaction. Saga rollback is achieved by rolling back the currently executing transaction, and performing the compensating transactions in the inverse sequence of the committed transactions.

To show how this is formalized using dependencies, we show the axioms taken from the formal specification of Sagas in [CR92], which we do not include here in full, as it is not required for this discussion. In these axioms, $pre(\epsilon)$ and $post(\epsilon)$ are used to specify preconditions or postcondition of an event ϵ that should hold, S is the saga that consists of a sequence of transactions $T1 \ldots Tn$ and the compensating transaction for Ti is Ci $(i : 1 \ldots n)$.

1. $post(Begin_S) \Rightarrow (Ti\ \mathcal{BCD}\ Ti - 1) \in DepSet_{ct}$
2. $post(Begin_{Ti}) \Rightarrow (Ci\ \mathcal{BCD}\ Ti) \in DepSet_{ct}$
3. $post(Commit_{Ti}) \Rightarrow (((Ci\ \mathcal{BAD}\ S) \in DepSet_{ct}) \wedge (Ci\ \mathcal{CMD}\ S) \in DepSet_{ct}))$

We can set a strict partial ordering between two transactions by placing begin-on commit dependencies between them, as in (1), where this is used to sequence the different steps Ti of a saga. Compensation of these steps, by a Ci, should only be executed if Ti did commit, which is specified in (2). Aborting the saga requires the compensating steps to start, as specified by the begin on abort dependency ($\mathcal{BAD}$) in (3), and to ensure that the compensating steps commit, a compensation dependency ($\mathcal{CMD}$) is stated in (3). Note that a strict ordering of compensating transactions can be achieved by placing begin-on commit dependencies as in (1).

6.2.6 Conclusion

In this section, we have given a formal ACTA definition for two transaction models: atomic and nested transactions by stating all the axioms that pertain to the transaction history.

Both these models rely on a formal definition of the auxiliary properties of serializability, serializable behavior and failure atomicity. Serializability and serializable

behavior were defined by five axioms. Both are, in fact a formalization of the conflict-serializability concept which we defined in 2.1.2. Failure Atomicity is equivalent to the atomicity property of classical transactions, and was formally defined in for axioms.

We gave a formal definition for atomic transactions, i.e. the classical transaction model, to use as a base-line definition, to compare definitions of ATMS to this base-line. The formal ACTA definition for atomic transactions is stated in eight, quite straightforward axioms. We have shown the formal definition of nested transactions, which modifies a few of these definitions, and adds eleven more, resulting in nine-teen axioms. This shows that although the nested transactions model is conceptually simple, the formal definition becomes quite extensive.

ACTA can also be used to define an ordering of transactions by using dependencies. We have illustrated this by taking the relevant axioms from the Saga definition and showing how the specification of the order of compensating transactions is achieved.

ACTA, however is a purely formal model, and therefore does not concern itself with possible implementation issues. If we want to provide an implementation of the model, such issues do indeed arise, as we will discuss next.

6.3 Comparing a Formal Model with an Implementation

Because ACTA allows the formal specification for a wide variety of ATMS, it is worth-while to use it as a basis for the implementation of a TP Monitor, which then can, in turn, support a wide variety of ATMS.

ACTA, however, is solely a formal model and was not conceived with such an implementation in mind. This is evident from the fundamental premise of ACTA as repeated in 6.1.1: “The correctness properties of different transaction models can be expressed in terms of the properties of the histories produced by these models” [CR91]. In other words, ACTA is a formal model which verifies a posteriori if the produced transaction history is correct. This implies that the record of this history is kept for ever, leading to a potentially infinitely large transaction history. It is clear that an implementation of a TP Monitor based on ACTA needs to do away with this, to restrict memory usage to a reasonable amount. Therefore, in general, it is best to keep as little information about the transaction history as possible, and remove it from the system when no longer needed.

A common technique to enforce correct transaction histories, without needing to record them, is the use of locks, as we discussed in 2.1.2. Using locks implies that we will need to do this verification while the transactions are running, in contrast to the a posteriori approach of ACTA. If we wish to use locks in a TP Monitor based on ACTA, we need to take whatever effort we can, while the transaction is running, to ensure that the resulting history is correct. There are four main topics to consider:

performance; dependency checking and significant events; transaction life-cycle; and naming and grouping. We outline these next.

Performance

The first implementation topic to consider is performance: the expressive power of ACTA is a possible performance hurdle for the implementation. In a number of places in the axioms an arbitrary logic expression can be used as a specification, which can take an arbitrarily large amount of time to execute. A complete implementation of ACTA, therefore, requires the TP Monitor to be able to not only reflect on the transaction history, but also to evaluate logic expressions given by the model specification. This is not only a significant implementation task, but would also have an, in general, undefinably large impact on transaction performance. This runs contrary to the expectations for TP Monitors, which is to process transactions in a short timeframe. It is therefore worthwhile to consider limiting the expressiveness of the framework somewhat in favor of a faster TP Monitor, which will also be easier to implement. We therefore take a minimal, feasible, implementation and can consider expanding this as required later.

Reviewing the model specifications published in [CR91, CR92], we can consider the following simplifications:

Dependencies: These have only been defined solely on significant events, therefore only these events need to be accessible to the part of the TP Monitor which verifies dependencies.

Conflicts: As we have stated in 6.1.3, conflict-checking is already performed in current TP Monitors through the use of locks. Therefore, we do not need access to the transaction histories to determine conflicts.

View traversal: While stated to be of use in multilevel transactions [WS92], this ATMS has not been formalized, and none of the published formal definitions of other models use a view traversal. Hence it may be omitted.

Delegation: Rewriting the transaction history is only performed simultaneously with a significant event, and most frequently the entire contents of the access set is delegated to another operation. Combined with conflict checking using locks, as above, where owning a lock implies responsibility of an operation, this means that delegation boils down to changing the ownership of all the locks of the source transaction to the target transaction.

Overall, this means that we can remove the explicit transaction history, and the expression evaluation, while still guaranteeing that a number of models are covered.

Of course, if we want to guarantee full compliance with the model, this removal is not allowed, but this will have a performance penalty. To assess, in general, the full impact of such removal is a hard task, as there are an unlimited amount ATMS which can be specified in ACTA. We can however relate the removal of the explicit transaction history, and the expression evaluation to the ATMS we have covered in chapter 2, and we will discuss this later in the dissertation, in 12.3.

Dependency Checking and Significant events

The second topic to consider are dependencies, which indirectly will also impact performance. As an example, consider the Commit Dependency ($Tj\ \mathcal{CD}\ Ti$): If Ti and Tj commit, Ti must commit before Tj. This is very easy to verify a posteriori, but this verification is more intricate if performed at runtime. If Ti wants to commit while Tj is running, commitment has to be postponed until Tj has committed to satisfy the dependency. Also, this postponement will turn out to be unnecessary if Tj aborts, as this abort implies no commit restriction on Ti. It is clear, however, that it is, in general, impossible to predict the behavior of Tj with regard to commits or aborts, and therefore this postponement is always required in case of a Commit Dependency. As a result, the performance of the TP Monitor will not be optimal in all cases, trading off performance for guaranteed correctness in all possible cases.

Tied to dependencies are the number of significant events defined by an ATMS, as these dependencies are specified relative to the significant events. In ACTA, the list of significant events is open, as each model may add its own significant events to the classical list consisting of begin, commit and abort. For example, in nested transactions, the spawn significant event is added, indicating the spawning of a new transaction. Such an open list of significant events requires a certain flexibility of the dependency mechanism, as it may need to be invoked at any point of the execution of a transaction.

It is clear that having a fixed set of significant events eases implementation of the dependency mechanism, which will probably also open avenues for performance optimization. Investigating the published models in [CR91, CR92], we see that this is indeed possible. This is because wherever a new significant event is introduced, such as for example the spawn event in nested transactions, this can be mapped to one of the events of the classical list, in the example the begin event, on a different transaction. Therefore, we can limit the significant events to begin, commit, and abort, and implement the dependency checking mechanism such that it only intervenes at these moments in the execution of a transaction.

Transaction Life-cycle

Third we need to concern ourselves with the underlying forces that steer the life-cycle of a transaction: Transactions are started and ended by the underlying application as it executes. However, ACTA, as a formal model, does not concern itself with the application's execution, but solely with the resulting history. Therefore, to ensure that the resulting history is correct, we need to consider how we can ensure that transactions start and end when necessary. It is clear that the main part of the responsibility for this lies with the control flow of the underlying application: at some points a transactional unit of code will run, resulting in the beginning of a transaction, and when the unit ends, the transaction will also end. A major assumption that we need to make here is that this control flow does not run contrary to the control flow mandated by the ATMS. We need to make this assumption because, in general, this is impossible to verify statically. For example, in nested transactions, the method of a sub-transaction has to end before the method of its parent has ended. If these methods run in different threads, and the child transaction depends on user input to terminate, we can not, at compile-time, verify termination of the child prior to the parent.

To ensure the application behaves correctly, it should be possible to let the TP Monitor perform some runtime checks through dependencies, and either take corrective action if possible, or produce an error if not. For example, in nested transactions two possible runtime errors may occur: a parent transaction may end before the child transaction has ended, or a child may declare to spawn from a non-existent transaction. In the first case, we can take corrective action by aborting the child, as specified though the appropriate dependencies. However in the second case, we cannot take any corrective action and will therefore be forced to produce an error.

Recall that, conceptually separate from the main control flow of the application, some ATMS require certain secondary transactions to run automatically when their dependencies are satisfied. For example in Sagas, compensating transactions are such secondary transactions because they have to start executing in the correct sequence when the Saga aborts. These secondary transactions are not related to the application logic per se but are a result of using the ATMS, and run automatically. Therefore, we should not require the application code to verify these dependencies and run the appropriate code, but treat this separately, somehow placing this extra responsibility at the conceptual level of the model, and not of the application.

Naming and Grouping

The fourth and last topic to consider is the naming and grouping of transactions: while formally representing transactions as Ti and Tj, such identifications are not simply translatable to a transaction at runtime. What we will need is to be able to

unambiguously identify a transactional unit of code at runtime, i.e. we will need some form of naming support. Also, as the view set of a transaction consists of a group of other transactions, we need to be able to define such groups to name them and also to add and remove transactions.

We have now briefly outlined four important topics relevant for the implementation of a TP Monitor inspired by ACTA. In the next chapter, where we discuss the implementation of our TP Monitor, called ATPMos, based on ACTA, we will return to these four topics in more detail.

6.4 Conclusion

In this chapter we discussed ACTA, a model which can be used to formally describe a wide range of ATMS. ACTA was developed in order to be able to reason about many ATMS, and to be able to synthesize new models, by combining properties of different existing ATMS. We first discussed the ACTA formalism before giving the formal definitions of a number of ATMS, and we ended with a short note on implementation.

We have shown that in ACTA, an ATMS is formally defined by stating three kinds of axioms that constrain and modify the transaction history. The first kind of axiom on transaction histories are dependencies, which place a relationship between two transactions, usually defined in terms of the significant events of these transactions. The second kind of axioms pertain to visibility between transactions, which allows different transactions to concurrently work on the same data as if they were the same transaction, relaxing isolation. The third kind of axioms modify the transaction history though the use of delegation, which delegates the responsibility for committing or aborting a specified number of operations operations from the delegator to the delegatee.

We discussed how the above kinds of axioms are augmented with the auxiliary properties of serializability, serializable behavior and failure atomicity, to allow the formal specification of a wide range of ATMS. Of the published ACTA specifications, we have discussed two in detail: the classical transaction model and nested transactions, the former requiring eight axioms, and the latter nineteen. Subsequently we have indicated how ordering of transactions is specified by using dependencies through a discussion of the relevant axioms from the Saga definition.

Finally, we have given a short note on implementation, indicating where the translation from a purely formal model to a working TP Monitor requires attention. We considered four topics: dependency checking, performance, transaction life-cycle and naming. We return on these topics in more detail in the next chapter, where we discuss our implementation of a TP Monitor based on ACTA. This TP Monitor allows us to support a wide range of existing ATMS, including newly created models.

Chapter 7

Demarcation Code for an ATMS TP Monitor

If I have seen farther, it is by standing on the shoulders of giants.
— **Newton (Attributed to Newton although the saying can be traced back to at least the 12th century)**

ACTA, which we discussed in the previous chapter, provides us with a formal model that covers a wide variety of ATMS, and which can be used to synthesize new models. A TP Monitor that is implemented based on the ACTA model, will therefore not only be able to support the ATMS of which a formal ACTA specification is available, but also be extensible to support new models that are expressible in ACTA. This gives us run-time support, on the one hand for many existing ATMS, and on the other hand for models that have not yet been created. As a result, use of this TP Monitor will allow an application to employ a transaction model that is tailored to the specific transactional properties of the application.

The TP Monitor we built is called *ATPMos* (**A**TMS **TP Mo**nitor), and is outlined first in this chapter. The focus of ATPMos is to provide an interface for the use of ATMS, inspired by ACTA, to client programs, i.e. the middle tier in multi-tier systems.

After providing an outline of ATPMos, we illustrate its use by means of a bank transfer example which is part of a middleware component. We outline how adding transaction demarcation code leads to tangled code for the bank transfer method when using traditional approaches. We then show how AOP achieves a cleaner implementation regarding separation of concerns.

Reconsidering the bank transfer example, we then see that it is better to use

the Sagas ATMS, and we proceed with implementing the bank transfer as a saga. We outline the demarcation code required to have the bank transfer operation as as Saga, when using ATPMos. We then further discuss this demarcation code, focussing on its large size and significant complexity, confirming the observation of 4.2 that demarcation code for ATMS itself is composed of multiple concerns. This leads us, in the next chapter, to create an aspect language and weaver which explicitly address these issues.

7.1 ATPMos: A TP Monitor for ATMS

In this section we introduce ATPMos, our TP Monitor which supports a wide variety of ATMS. We first consider ATPMos solely as a TP Monitor that supports classical transactions. We then identify the features required for ATMS support, and give an overview of how these features are provided by ATPMos. This shows how, conceptually, a TP Monitor can be extended to support a wide variety of ATMS by implementing the ACTA primitives. Finally, we give an overview of the interface which ATPMos presents to the client program, enabling the client program to employ an ATMS. Note that, as the goal of this dissertation is not to write a fully ACTA compliant TP Monitor we do not discuss the implementation of ATPMos in detail here. Instead a sketch of the implementation of ATPMos is given in appendix A.

ATPMos is designed as a TP Monitor for EJB Entity beans, which is our choice of middleware and was discussed in 5.3. ATPMos assumes that state is contained within such objects and that state modification happens through getter and setter methods. As a consequence, concurrency management is performed when these methods are called: calling a getter or setter on an Entity Bean when in a transaction requires that first a call is made to ATPMos. These calls inform ATPMos that a read or write will be performed, and contain as parameters the current transaction identifier, the object and the field on which the read or write occurs. Internally, ATPMos uses a locking strategy to determine if this operation is allowed, acquiring locks if necessary. If the operation may proceed, the call will simply return. If the operation is not allowed at the moment, i.e. the required lock could not be acquired, the call will block until the lock is acquired.

To perform a read or write within a transaction, however, this transaction will first have to be started, which is done through additional demarcation code. The call to ATPMos to start a transaction will return a transaction identifier that uniquely identifies the started transaction. All other demarcation code then uses this identifier to indicate to which transaction they pertain. Finally, to end a transaction, also a call to ATPMos is made to either commit or abort the transaction.

Note that ATPMos is not a priori restricted to EJB Entity beans. As long as data

access occurs using getters and setters, for example in Java Data Objects [Sun03a], these data accesses can be managed by ATPMos. Also, at this time, ATPMos only provides the concurrency management features of transactions. Recovery of ongoing transactions when the server running ATPMos is down is not implemented, since this lies outside of the goal of this research.

We do not talk about the implementation of ATPMos as a TP Monitor for classical transactions here. Instead, we refer to Appendix A.1, where we discuss this topic in some depth. In table 7.2 (located in section 7.1.3), however, the interface to ATPMos for classical transaction management is given.

7.1.1 From Classical Transactions to ATMS support

Considering ATPMos as a TP Monitor for classical transactions, we now ascertain what needs to be added to provide support for ATMS. To determine this, we need to look back at the previous chapter, and see which features from ACTA we have not yet addressed.

To determine the features required for ATMS support, we consider each of these features, as we introduced them in 6.1: first we look at events and history, second at dependencies, third at conflicts, fourth at views and access, fifth at delegation, and sixth we reevaluate the issues we raised in 6.3.

Events and History: In ACTA, a history is kept of all operations on data, performed by the different transactions. In contrast, a TP Monitor does not keep such an explicit history. However, as we discussed in 6.3, we can do away with this explicit history altogether, and just need to concentrate on a representation of the significant events of the transaction.

Dependencies: Dependencies place constraints between the events of different transactions, support for which is totally absent in a TP Monitor for classical transactions. Recall that we have seen in 6.3 that we only need to support constraints on significant events. Therefore, what we will need is some way in which running transactions, immediately before a significant event, verify if this event can be sent or not. In other words, immediately before a begin, commit or abort, we need to verify if this operation is allowed given the current dependencies, as we have indicated in 6.3.

Conflicts: Conflict-checking is performed in ATPMos through the use of locks, implementing the observation we made in 6.1.3. Therefore, no extra work will be required here.

Views and Access: Views relax the isolation between different transactions, allowing one transaction to view the intermediate results of another transaction. Ac-

cess sets contain all the data that has been seen or accessed by a transaction. The access set typically is already present in a TP Monitor for classical transactions, as the collection of resources which are locked by the transaction, and this is also the case in ATPMos. Views, however, are not present in classical transactions, so in order to support ATMS the view set of a transaction is kept, and the locking algorithm takes these sets into account. Note that in 6.3 we remarked that if we opt to leave out view traversal order, we still are able to support a large number of ATMS.

Delegation: Delegation allows one transaction to delegate the responsibility for committing or aborting a specified number of its objects to another transaction. This is not present in classical transactions, and therefore will need to be provided to have ATMS support. Note that we can fix the selection of objects to be delegated as the access set, which we mentioned in 6.3.

From a Formal Model to an Implementation: One topic discussed in 6.3 gives rise to a new requirement: Naming. As we show later, we need some naming infrastructure to be able to, from one transaction, easily refer to other transactions. This is useful, for example, when we need to specify a dependency.

Regarding the implementation of these features, we note here that a full implementation of a TP monitor that is completely compliant with the ACTA model would require a very large amount of work, due to the high complexity. Also, significant research has already been performed on this subject [AC95, ASSR93, BDG+94, BP96, GHKM94]. However, the focus of this dissertation is not on building TP Monitors for ATMS, but on how to program the software that interacts with them. Therefore, we have chosen not to pursue this in depth. Instead we built a TP Monitor the implementation of which is not fully ACTA compliant, but the interface does provide all the required features for the basic implementation we considered in 6.3.

To summarize, conceptually we need to add four elements to ATPMos to be able to support ATMS: views, delegation, dependencies and naming. We discuss these extensions next.

7.1.2 Implementing ATMS support

Having identified the conceptual extensions to classical transactions which are required to enable ATMS support, we now outline how these extensions are implemented in ATPMos. We begin with naming and grouping, proceed with views and delegation, and end with dependencies.

Again, we do not discuss the implementation of these concepts here, but refer to appendix A for a more detailed treatment of this subject.

The Need for Naming and Grouping

As we have already mentioned in 6.3, from the standpoint of a formal model using names such as Ti and Tj, and defining a set of transactions formally works just fine, but when going to an implementation of the model, this raises a number of technical issues. Dependencies, view and delegation take as arguments transaction identifiers of actual running transactions or groups of transactions, and therefore these identifiers have to be obtained from somewhere. We now discuss, conceptually, how we address this issue in ATPMos using a naming service. This service provides support for naming and grouping transactions, as we see next.

As an example of the use of naming, consider nested transactions. In the ACTA formal model, we can simply state that "Tj is a child transaction of Ti", but at runtime, we need a way in which we can ascertain the identity of the parent from within the method code of the child, and vice versa.

Classical transaction management seems to provide a solution for this problem through transaction context propagation, which we discussed in 5.2.1. To summarize, the identifier of the currently running transaction is passed as a hidden parameter with every method call. This would allow a child transaction, called by the parent, to first obtain a reference to the parent before starting. However, context propagation is only performed by a method call, so if a child transaction is not called by the parent, e.g. when it runs in a different thread, this parameter is not passed. In that case the child will not be able to obtain a reference to the parent. Furthermore, transaction context propagation only allows one transaction identifier to be propagated. If we would want a given transaction to place dependencies between itself and two other transactions, this would be impossible as only one other transaction can be referred to. Therefore we need to provide a more general solution.

A first reflex would be to graft this information onto the control flow of the application. For example, again consider nested transactions, where sub-transactions of a parent transactional method correspond to the methods called from within the parent transactional method. This would imply that we need to modify each method's signature to now also contain the identifier of the parent transaction. Arguably, this can become troublesome when a large amount of methods and method calls is involved. Furthermore, passing transaction identifiers from a called method to the caller is even more awkward in Java, since only one return value is allowed. If we want to pass the original return value, combined with a number of transaction identifiers, we need to make a value object [ACM01], and change the return type of the method. Another solution, avoiding changes in signature, would be to use a global variable to keep the identifier of the parent transaction, but this would not work if two nested transaction hierarchies are running concurrently. Therefore, we need to separate naming of transactions and transaction groups from the control flow of the application, and provide

this as an extra feature of ATPMos.

The ATPMos Name and Group Service

To address the issue of naming, we have chosen to implement a *name service* in ATPMos, inspired by the wide variety of name services available in distributed systems. The name service acts as a global dictionary, allowing transactions to bind their identifier to a key of their choosing, and to retrieve identifiers of other transactions based on their key. We do not include the interface definition for the naming service here, instead we refer to table 7.1 (in section 7.1.3).

Note that the name service can be used to implement classical transaction context propagation, as follows: In our nested transaction example, the method of the parent transaction can bind its identifier using the current thread as key. Called methods correspond to sub-transactions and wish to obtain the identifier of the parent. They perform this by performing a lookup using as key the current thread, which is the same thread as the parent methods' thread.

Grouping of transactions can be found in ACTA in, for example, the view sets. Consequently, ATPMos also provides support for groups of transactions, where a key now refers to a set of transactions. Groups are separate from the naming of transactions and can also be named, in a registry separate from the naming of transactions. This avoids ambiguity of lookup for transactions and groups, i.e. if a transaction is looked up no group will be returned and vice-versa. Also, this allows both a transaction and a group to have the same name, giving the application programmer more flexibility. As for the naming service, the interface definition is given in table 7.1 (in section 7.1.3), and we do not include it here.

This concludes our introduction of the naming service of ATPMos, which provides for a global registry of transactions and transaction groups. Using this service the code from one transactional method can obtain the identifier of other transactions and transaction groups. Using those identifiers, this transaction can then set views, place dependencies and so on, as we will discuss next.

Views and Delegation

Naming and groups, which we have just seen, do not immediately correspond to any ACTA concepts, but will be used by the features that are covered in the remainder of this chapter. The most straightforward of these are the implementation of the ACTA concepts of views and delegation, which we discuss here.

Views allow one transaction to see the intermediate results of another transaction. We have implemented two distinct ways in which views are set, first considering views between two transactions, and second the use of a group to define the view of a

transaction.

Views are used when accessing data from within a transaction. When attempting to read or write a datum in a transaction Ti, ATPMos decides if a lock should be acquired for this operation. When using views ATPMos also checks if another transaction Tj already has a lock of the required type and is in the transitive closure of the view of Ti. If this is the case, Ti does not need to acquire any lock and can simply proceed. Using groups to define a view from one transaction to a set of other transactions is analogous. When building the transitive closure of all transactions in the view of the current transaction, the name service is called to obtain a list of all transactions in the viewed groups. The resulting transactions are added to the closure, and used as roots for further building this closure.

Through delegation, a delegating transaction Ti delegates responsibility for committing or aborting data modifications to another transaction Tj. As we have already observed in 6.3 delegation boils down to changing the ownership of all the locks from Ti to Tj and therefore we do not discuss it further here.

The interface definition of the ATPMos methods for views and delegation are given in table 7.2 (in section 7.1.3), and therefore we do not include this definition here.

Dependencies

The use of dependencies effectively adds a number of constraints to the execution of significant events of transactions, as we have discussed above. More specifically, in 7.1.1, we summarized that what we need is a way in which a running transaction Ti, immediately before a significant event, asks if this is allowed. As we introduced in 6.3, Ti can be made to wait until a dependency with another transaction Tj is satisfied. The example we used is the Commit Dependency ($Tj\ \mathcal{CD}\ Ti$): If Ti wants to commit while Tj is running, commitment has to be postponed until Tj has committed. However, it is possible that waiting is not sufficient: take for example the Compensation Dependency ($Tj\ \mathcal{CMD}\ Ti$): If Ti aborts, Tj must commit. If Tj requests to abort, it not only has to wait until Ti has ended, but in case Ti aborts, the request to abort will also be denied, and Tj is forced to commit to satisfy the dependency.

It is important that the types of dependencies that can be enforced by ATPMos is open-ended, because of the general-purpose nature of ATPMos. We cannot foresee all dependencies that will be necessary to support all possible ATMS, therefore we must be able to add new dependencies to ATPMos when needed. We have achieved this by modeling transactions and dependencies as Petri nets [Pet77] in separate modules, and allowing new modules, representing new dependencies, to be added to the system at any time. We maintain a clean separation between the representation of a transaction and the representation of dependencies, which allows new dependencies to be easily created and introduced. To introduce such a new kind of dependency, we only need

to create a representation of that dependency as a module and add it to the system. At runtime, we can then use this new representation, without even needing to restart ATPMos.

We don't discuss the implementation of the dependency mechanism here, but we do give a detailed overview in A.5. To summarize, ATPMos keeps a run-time model of transactions and their dependencies, which is used to verify and enforce dependencies. A link is kept between the model and the running transactions, and is used at the significant events of the transaction. Before a transaction begins, commits or aborts, the state of the model is updated by calling ATPMos. Upon updating the model dependency checking is performed and the caller is blocked until either all dependencies are satisfied, or the transaction must immediately proceed to a committed or aborted state. If the update call to ATPMos ends normally, the transaction can effectively begin, commit or abort. If the update call ends with an error indication, this requires the transaction to commit or abort immediately, as specified by the error indication, to satisfy a dependency.

Again, we do not include the interface definition for dependency enforcement here, and instead we refer to table 7.2.

7.1.3 ATPMos Interface

As we said above, the focus for ATPMos is to provide an interface for the usage of ATMS, inspired on ACTA, to client programs. In the previous sections, we have deduced the features required of ATPMos to provide ATMS support, and given a general outline of their implementation. We end the introduction to ATPMos by giving an overview of the interface of ATPMos, which allows the clients to use these features.

We present the interface here split in two tables, first we give an overview of the interface for naming and groups, and second we provide the interface for classical transaction management, views, delegation and dependency management. As all these features have already been discussed, we do not further elaborate on them here.

7.1.4 Conclusion

In the previous sections, we have introduced ATPMos, the TP Monitor we created for this dissertation. ATPMos provides for transaction management on EJB Entity beans through the use of locks. Within a transaction, concurrency management is performed when getter and setter methods of the bean are called by first calling the read or write methods on ATPMos. These methods block until that datum is locked for reading or writing, respectively, effectively performing concurrency management.

ATPMos was first introduced considering it solely as a TP Monitor for classical transactions, and later conceptually extended to support ATMS by adding the required

Name		Summary
void bind	(Object key, Integer id)	Register transaction identifier.
Integer lookup	(Object key)	Lookup transaction identifier.
void unbind	(Object key)	Unregister identifier key.
void unbindAll	(Integer id)	Unregister all identifier names.
void addToGroup	(Object key, Integer id)	Add identifier to group.
Integer [ ] lookupGroup	(Object key)	Lookup identifiers in group.
void removeFromGroup	(Object key, Integer id)	Remove identifier from group.
void removeGroup	(Object key)	Unregister group.
void bindGroup	(Object key, Object group)	Register group.
Object lookupGroupBinding	(Object key)	Lookup group.
void unbindGroup	(Object key)	Unregister group key.
void unbindGroupAll	(Object group)	Unregister all names for group.

Table 7.1: ATPMos interface overview for naming and groups.

ACTA primitives. We identified four extensions that need to be made to provide support for ATMS: naming, views, delegation and dependency support. We sketched the implementation of these features, but did not provide a detailed discussion about the implementation, as it lies outside of the scope of this thesis. We have however provided an overview of the interface which is presented by ATPMos to client programs wishing to use an ATMS. For more information regarding the implementation of ATPMos, we refer to appendix A, which further goes into depth on this topic.

Having introduced ATPMos, we outline in the remainder of this chapter how software using an ATMS interacts with ATPMos, i.e. we talk about the transaction demarcation code that is written when using ATPMos as a TP Monitor for ATMS.

7.2 Demarcation Code for Classical Transactions

Applications using transaction management will steer ATPMos through transaction demarcation code (we introduced the concept of transaction demarcation code in 2.1). This code is added to the application, defining the start and end of transactions, what operations are involved in the transaction, and when using an ATMS, setting dependencies, views and performing delegation. By writing the appropriate demarcation

Name		Summary
Integer newID	()	Returns new identifier.
void begin	(Integer id)	Start transaction.
Integer begin	()	Start a new transaction, returns id.
void commit	(Integer id)	Commit transaction.
void rollback	(Integer id)	Rollback transaction.
void read	(Integer id, Object key)	Block until object can be read.
void write	(Integer id, Object key, RestoreData res)	Block until object can be written, keeps backup value.
void addView	(Integer source, Integer dest)	Add a view from source to dest.
void removeView	(Integer source, Integer dest)	Remove the view from source to dest.
void addViewGroup	(Integer source, Object dest)	Add a view group to source.
void removeViewGroup	(Integer source, Object dest)	Remove the view group from source.
void delegate	(Integer source, Integer dest)	Delegate all resources from source to dest.
void addTransaction	(Integer id)	Add transaction to the model.
void removeTransaction	(Integer id)	Remove transaction from the model.
void addDependency	(Integer left, String type, Integer right)	Add dependency of given type between two transactions.
Forcing mayBegin	(Integer id)	Block until transaction may begin.
Forcing mayCommit	(Integer id)	Block until transaction may commit.
Forcing mayAbort	(Integer id)	Block until transaction may abort.

Table 7.2: ATPMos interface overview for ATMS.

code, i.e. using these building blocks in the correct way, the application programmer ensures that transaction management for the application is performed conforming to the model of choice.

As we already argued in chapter 3, programming such demarcation code is not a trivial task, even if we only consider classical transactions. In this chapter we illustrate this problem using an example banking application, of which we render the money transfer operation transactional. We see that in the resulting code both the core concern of the application and the transaction management concern are very tangled, therefore making the application difficult to write and maintain. Furthermore, we illustrate that the design of the application should take transaction management into account. If this is not the case, adding transaction management can require more than adding just demarcation code, in some cases even new features have to be implemented.

We start by describing the application we use as an example: a bank transfer operation. We then describe three ways to add the concern of transaction management to this operation, and the impact this has on the code with regard to separation of concerns. Later in the text, after considering classical transactions, we show that in advanced transaction management, programming demarcation code becomes even more of a burden, using the same example code.

7.2.1 The Example Application

The example application we choose to illustrate our problem statement is inspired by the classical example we used in chapter 2: a banking application. The example application is conceived as a part of a multi-tier distributed system, using Enterprise JavaBeans (EJB), which we introduced in 5.3, as the middleware server architecture.

In this banking application, the database layer is handled by the EJB server, and persistent data is referenced through EJB Entity beans. This allows us, later on, to use ATPMos to perform advanced transaction management for this application. The application we have written represents just one middleware component in the entire banking application, and is intended to be used by cashiers at a bank till. This `Cashier` component allows the cashier to deposit money into an account, withdraw money, transfer money between different accounts, et cetera.

As we wish to demonstrate the impact of transaction demarcation code on an application, and not how to build a middleware component, we choose to focus on just one operation: transferring money from one account to another account. This operation is performed by the `moneyTransfer` method, which takes a source and destination `BankAccount`, which are EJB Entity Beans, and the amount to be transfered.

We do not include the full code for the `moneyTransfer` method here, we refer to appendix B, which contains all the Java code discussed in this chapter, and more

specifically to B.1. Suffice it to say that the code for the `moneyTransfer` method consists of three distinct steps. The first step performs the actual bank transfer, the second step prints a receipt to be given to the customer, and the third step logs the transfer operation. Note that the sequence of these three steps has been explicitly fixed in the requirements of the application, which will be important later.

Instead of including the complete code of the transfer operation to illustrate the effect of demarcation code on this method, we show the code for one step of the bank transfer. In the following sections we repeat this code with demarcation code added. The code we show is for the first step of the bank transfer: the actual transfer itself. As this code is quite straightforward, we do not discuss it in detail, and simply include it below:

```
4     int from_amount = from.getAmount();
5     int to_amount = to.getAmount();
6     to.setAmount(to_amount + amount);
7     from.setAmount(from_amount - amount);
```

A major problem with the code for the money transfer operation is that it is abound with concurrency issues. The first step, the code of which is given above, has even been used as a motivating example for the need of transactions. This is because unchecked concurrent transfer operations can cause money to, as if by magic, appear or disappear from the accounts [CDK94]. Therefore, to remove these concurrency issues, we make this transfer operation transactional, which we see next.

7.2.2 Making the Transfer Operation Transactional

The `moneyTransfer` method, which we have introduced above, contains many possible sources of race conditions. Therefore, to add concurrency management, we make this method transactional, using our TP Monitor, ATPMos, which we introduced above. This entails manually adding transaction demarcation code to this method, which makes the required calls to ATPMos.

The code for the transactional money transfer is not included here, instead we refer to B.2. Most important for this code is that a lot of transaction demarcation code has been added, and that it is spread out all over the application code, cross-cutting the original concern. As a result, the size of the code has more than doubled and readability is significantly decreased because transaction demarcation code tangles an extra concern into the code.

This is most apparent in the three steps of the transfer operation, where the programmer has to switch concerns every few lines. The programmer needs to address both the core concern of the code, in this case performing a transfer operation, and the

secondary concern of transaction management in a tangled way. We illustrate this by repeating the code from the first step of the transfer operation, now with demarcation code added, and emphasised like this:

```
11    txmgr.read(tx_id, from);
12    int from_amount = from.getAmount();
13    txmgr.read(tx_id, to);
14    int to_amount = to.getAmount();
15    RestoreData tores = new RestoreData(to,"amount");
16    txmgr.write(tx_id, to, tores);
17    to.setAmount(to_amount + amount);
18    RestoreData fromres = new RestoreData(from,"amount");
19    txmgr.write(tx_id, to, tores);
20    from.setAmount(from_amount - amount);
```

The cause for this tangled code is that before every read or write of an Entity Bean, a call must be made to ATPMos, as we discussed above. This runs counter to the separation of concerns principle we detailed in 3.1.1; instead of only focussing on one concern, the programmer now needs to keep two concerns in mind and even constantly switch between them. As said in 3.1.1, writing such code is difficult, as is extracting the original concerns from the produced code, which results in code that is difficult to maintain.

Furthermore, making the method transactional requires extra code to be defined by the base application, which is not illustrated by the example code above. Because a transaction may be rolled back after the transfer receipt has been handed to the customer, this receipt needs to be annulled by printing out a cancellation notice. However, it makes no sense always to print out such a notice, as in some cases no receipt has been printed. Therefore demarcation code defines a flag which is set after the receipt is printed and checked when rolling back the transaction. When rolling back the transaction and the flag is set, an extra method call is made to print a cancellation. This sub-concern of printing the cancellation notice introduces yet more code tangling.

Note that the cancellation notice was not present in the original banking application code, and is the result of introducing transactions, combined with the prohibition to rearrange steps. Were we allowed to print out the receipt as last, no cancellation would be required. This is an example reason for the statements made in [KG02, RC03], which we discussed in 3.2, asserting that applications should not be constructed totally obliviously from the concern of transaction management. However, because this was not the case, we need to make amends, and extend the base application with new functionality: printing out a cancellation notice.

Traditional solutions exist that attempt to reduce the complexity of adding demarcation code to method code, more specifically using wrappers. However, wrappers are not a complete solution and provide only partial relief for the issue of tangling of demarcation code, as we shall see next.

7.2.3 Simplification Through Wrappers

As we have seen above, an issue in the demarcation code in the previous section is issuing calls to ATPMos before each read and write of the data. This introduces a high degree of tangling between the core concern and the concern of transaction management and furthermore is quite repetitive. To address this issue, a traditional solution is to use a wrapper object. Instead of making each getter and setter call directly to the affected object, we make the call to its wrapper. The wrapper first performs the required call to ATPMos, followed by the call to the real getter or setter.

Again, we do not include the complete code for this version of the money transfer operation, it is given in B.3. Important in this code is that, while the number of lines of code has not dropped dramatically, the three steps of the transfer operation have become much more legible. This part of the method has almost reverted back to the original form and the role of the demarcation code is much less pronounced here, as we can see from the example code below (again with demarcation code emphasised like this):

```
11    BankAccountWrap from = new BankAccountWrap(from_orig);
12    BankAccountWrap to = new BankAccountWrap(to_orig);
13    int from_amount = from.getAmount(tx_id);
14    int to_amount = to.getAmount(tx_id);
15    to.setAmount(to_amount + amount, tx_id);
16    from.setAmount(from_amount - amount, tx_id);
```

The differences here are that we now create wrappers for each datum, and in the getter and setter calls we pass the transaction identifier as extra parameter. Although this implies that demarcation code is still tangled with the core concern, it has become more implicit. Consequently, the programmer does not need to switch between the two concerns as much above, and the concern of transaction management takes on an inferior role. This results in a greater separation of concerns, making the code easier to implement and maintain.

Working with wrappers has a downside, however, because when handing out references to Entity Beans, for example to print out the receipt, we should not pass the wrappers, but pass the original objects. Also, we have to take care that calls that are not transactional are not performed on the wrapper, but on the original object.

In other words, adding wrappers to handle transaction demarcation implies adding an extra concern to the code: managing these wrappers. Therefore, adding wrappers requires that a trade-off must be made between making transaction management more implicit and adding the extra sub-concern of wrapper management to the code.

So, to summarize, while the use of a wrapper significantly improves the code, it is not a complete solution. This is because while it makes transaction management more implicit, it also adds the extra sub-concern of wrapper management.

Note that EJB, as discussed in 5.3, does provide for automatic wrapper management, and some form of abstraction of the demarcation code, by making method boundaries coincide with transaction boundaries, which we discussed in 5.3. However, as we have already remarked in 5.3.5, this does not allow us to perform exception handling at the level of that method, making it hard, if not impossible, to specify in that method that a transfer cancellation printout is needed if a receipt has been printed. This issue, combined with the lack of ATMS support, prohibits us to choose that path to achieve further simplification of the example application. Therefore, the code above is the most concise transactional form of the `moneyTransfer` method we can achieve without using AOP.

To conclude, we see that going from code oblivious to transaction management to code with transaction management has led to significantly tangled code. Furthermore, not only demarcation code was added, but the existing code needed to be extensively modified to work with wrappers and to provide exception handling, for which even an extra feature had to be added in the application. It is therefore clear that making code transactional is not always as straightforward a task as it would seem to be.

7.2.4 Transaction Management as an Aspect

We have seen above that using traditional software engineering, the concern of transaction management remains tangled with the core concern of the application. Transaction demarcation code remains interleaved with the code performing the actual bank transfer, leading to a poor separation of concerns. As a result of this poor separation of concerns, the code is difficult to write and maintain, as asserted in [HVL95] and which we discussed in 3.1.1.

To obtain a better separation of concerns for this code, existing work, including some of our own, uses Aspect-Oriented Programming (AOP), which we introduced in 3.1.2, and we discussed this existing work in 3.2. The use of AOP allows for a much stronger separation of concerns, leading to code that is easier to develop and maintain. We have therefore implemented a `moneyTransfer` operation, in which transaction management has been aspectized. The original `moneyTransfer` method is now accompanied by an aspect definition, which is placed in a separate file that describes the transactional properties of the method, i.e. that defines transaction demarcation.

As aspect technology, we use our previous work, which we discussed in 3.2, the goal of which was to modularize transaction management as an aspect, specifically focusing on exception handling within this aspect. The base aspect (i.e the application code) is written in plain Java, while the transaction properties are written down separately, in a little language created specifically for declaring transactional properties.

The code for the base aspect is not included here, instead we refer to B.4. Most noticeable about this code is that it is almost unchanged from the initial code of the `moneyTransfer` method. The code for the first step of the transfer operation is, in fact, the same as the version without transaction demarcation, which we have shown in 7.2.1, and therefore we do not include it here.

In the code for the `moneyTransfer` method only two small changes are required, and both changes are a consequence of the design of the application, which did not take transaction management into account. We return to this after discussing the aspect code. As a result, this code provides for the cleanest separation of concerns yet encountered. Except for a few statements, the base aspect code is untainted by the transaction aspect.

The description of the transactional properties of the method is discussed in detail in B.4, where the full code is given in the little aspect language we built for this aspect. Instead of requiring a significant amount of java code to make this method transactional, the code in B.4 uses but a few statements. The majority of the code, in fact, consists of a rollback handler. Without this rollback handler, the `moneyTransfer` method can be made transactional using the two statements below, which are self-explanatory:

```
1 transactions Cashier {
2    moneyTransfer(BankAccount, BankAccount, int) new;
3 }
```

This illustrates that stating the transactional properties of the method using an aspect-specific language is very concise and short, raising the abstraction level of the code and hiding implementation details.

Having virtually untainted base code and a separate description to declare the transactional properties of a method, confirm the observations we made in 3.2. AOP is indeed a superior solution to specify transaction management that allows for modularization of this concern. Using AOP, we have achieved the highest separation of concerns, resulting in code in which the different concerns are almost completely separated, i.e. almost no code tangling is present.

The cause for the slight tainting of the Java code by the transaction concern, is again the issue of the design of the application: if the application were designed with transaction management in mind, the construct to print out the cancellation notice

would not be required. As it is, we have a partial solution, limiting the impact of this construct to two statements in the base aspect. This is a confirmation of the observation, which we discussed in 3.2, that the design of the application has to take the concern of transaction management into account.

Finally, we note that using a concern-specific aspect language, as above, has the advantage that conceptually the code is at a higher level which hides implementation details. This, combined with the conciseness of the code further aids the implementation of the application.

7.2.5 Conclusion

In the above sections, we have illustrated what the impact is of adding transaction management to existing code, showing how this new concern interleaves with the core concern of the application, leading to very tangled code. Consequently, according to the principle of separation of concerns, this code is hard to write and maintain, and a better modularization of transaction management is called for. As a first attempt to modularize transaction management we have used wrappers. We have shown that, when using wrappers, while the concern of transaction management is less spread out over the code, we still have code tangling, and furthermore the extra sub-concern of wrapper management is also added to the application. A second attempt was made using AOP, and we have just seen that this results in a much cleaner separation of concerns, conferming the observation in 3.2. Now the code for the base aspect has almost reverted back to the original form.

The question which now remains to be answered is what happens if we repeat this exercise for advanced transaction management. In other words, what is the impact on the application code considering separation of concerns if we want to use an ATMS instead of classical transaction management. We provide an answer to this question next.

7.3 The Transfer Operation as a Saga

We have now illustrated the impact on complexity and size of application code as a result of adding demarcation code for classical transactions. However, when we consider the example transfer operation from a run-time point of view, we see that it would make sense to use an ATMS for this code, namely Sagas, which we discussed in 2.3. This is because a major issue with the transfer operation is the printing out of the receipt: to make a printout, even on a fast printer, takes a few seconds, which effectively turns the transfer operation into a long-lived transaction. It therefore makes sense to split up the transfer operation into a Saga, allowing for more concurrency within the banking application.

In this section, we gauge the impact this has on the application, considering transaction demarcation code. We see that the demarcation code is more scattered than when using classical transactions and that a lot more demarcation code is required. We also discuss the higher complexity of the demarcation code and its tangled nature, which was predicted in 4.2. We show how this tangled code confirms the conceptual discussion in 4.2, which lead us to the observation of tangled aspect code.

We now first talk about this demarcation code, addressing the amount of demarcation code required to use the Sagas ATMS, and the complexity of the demarcation code. This is followed by a discussion on the tangled nature of this demarcation code, which confirms our observation in 4.2.

7.3.1 Demarcation Code for the Saga

To render the transfer operation into a saga, we first split up the transfer into three parts: a `transfer` method, a `printReceipt` method, and a `logTransfer` method. Each of these methods coincides with the steps we used when first describing the transfer operation above. The transfer example code we have used previously to illustrate the impact of demarcation code now is the body of the `transfer` method, as can be seen below:

```
11 private void transfer
12    (BankAccount from, BankAccount to, int amount)
13 {
14    int from_amount = from.getAmount();
15    int to_amount = to.getAmount();
16    to.setAmount(to_amount + amount);
17    from.setAmount(from_amount - amount);
18 }
```

This split into a saga has as a result, that when printing the receipt in the `printReceipt` method, less locks are held on the bank account objects, which allows them to be concurrently accessed by other transactions. Splitting up this long lived transaction into a saga has effectively added extra concurrency to the system, increasing transaction processing speed.

To use the saga ATMS for the money transfer operation without using AOP, we need to add transaction demarcation code to the different methods introduced above. In addition to the classical transaction demarcation code given above the bulk of this demarcation code manages dependencies: it will declare dependencies, based on the formal description of sagas in [CR92], and verify these dependencies.

Note that, as a result of splitting the saga up into different methods, the demarcation code for the saga will now be scattered over these different methods. This

contrasts with classical transactions, where the demarcation code for one transaction is not scattered at all.

We do not include all the code for the different methods `moneyTransfer`, `transfer`, `printReceipt`, and `logTransfer` here, neither with nor without demarcation code. All this code is contained within appendix B, more specifically in B.5. Instead we outline the most significant parts of this code next, and we use the `transfer` method as an illustration, after discussing the top level `moneyTransfer` method.

The top level of the saga corresponds to the `moneyTransfer` method, the body of which calls the three steps in sequence. Although this method does not perform any work that is transactional, we still need to declare it as a transaction because the different steps in the saga place dependencies on this top-level method. This is done to enable roll-back of the saga: if the saga is rollbacked, the top-level method will be marked to roll-back, which will trigger the compensating transactions due to the fact that their dependencies on the saga will be satisfied. Also, at the end of the saga, the internal representation of transactions and dependencies kept in ATPMos is no longer required and is therefore cleaned up by the demarcation code at the end of the `moneyTransfer` method.

It is striking that of these forty-eight lines of code, only three lines are not concerned with transaction demarcation. Although this implies that not much code tangling takes place, the transaction concern still has a large impact, due to the size of the demarcation code, and of the complexity of the demarcation code itself.

Considering the different steps of the saga, each of these, save for the last one, needs to define a compensating transaction that is run when the saga is rollbacked after that step has already committed. The compensating transaction performs a semantic "undo", such that at the end of saga rollback the database is consistent, as we have discussed in 2.3. Recall that in 2.2, where we have described secondary transactions, we identified compensating transactions in Sagas as such secondary transactions.

For the first step in this example, the `transfer` step, we created an `undoTransfer` method, which performs the inverse of the transfer operation, the code for which is given in B.5.3. Note that as this method is also a transaction, it also includes demarcation code. The code for the `transfer` step itself was already given early in this dissertation, in 4.2, where we discussed the concerns within the demarcation code. For convenience, however, we also include this code here (again with demarcation code emphasized):

```
120 private void transfer
121     (BankAccount from_orig, BankAccount to_orig, int amount)
122     throws TxException
123 {
124     TransactionManager txmgr = TransactionManager.getCurrent();
```

```
125     Integer tx_id = txmgr.newID();
126     txmgr.addTransaction(tx_id);
127     Integer saga = txmgr.lookup(Thread.currentThread());
128     txmgr.addToGroup("Saga"+ saga + "Step",tx_id);
129
130     final Integer comp_id = txmgr.newID(); //for compensation
131     txmgr.addTransaction(comp_id);
132     txmgr.addToGroup("Saga"+ comp_id+ "Comp",comp_id);
133     txmgr.bind("Saga"+ comp_id+ "Comp",comp_id);
134
135     final BankAccount compfrom = from_orig; //for inner class
136     final BankAccount compto = to_orig; //for inner class
137     final int compamount = amount; //for inner class
138
139     Runnable compensator = new Runnable()
140 {
141     public void run(){
142        undoTransfer(compfrom, compto, compamount, comp_id);
143     }
144 };
145
146     txmgr.addDependency(saga, "ad", tx_id);
147     txmgr.addDependency(tx_id, "wd" ,saga);
148     txmgr.addDependency(comp_id, "bcd" ,tx_id);
149
150     new Thread(compensator).run();
151
152     Forcing bf = txmgr.mayBegin(tx_id);
153     if (bf == null)
154        txmgr.begin(tx_id);
155     else {
156        txmgr.rollback(tx_id);
157        return;
158     }
159
160     try {
161        BankAccountWrap from = new BankAccountWrap(from_orig);
162        BankAccountWrap to = new BankAccountWrap(to_orig);
163        int from_amount = from.getAmount(tx_id);
164        int to_amount = to.getAmount(tx_id);
```

```
165         to.setAmount(to_amount + amount, tx_id);
166         from.setAmount(from_amount - amount, tx_id);
167
168         Forcing cf = txmgr.mayCommit(tx_id);
169         if (cf != null)
170            throw new TxAbortedException();
171
172         txmgr.addDependency(comp_id, "cmd" ,saga);
173         txmgr.addDependency(comp_id, "bad" ,saga);
174
175         txmgr.commit(tx_id);
176      }
177      catch (TxException ex){
178         txmgr.mayAbort(tx_id);//will always succeed
179         txmgr.rollback(tx_id);
180         throw ex;
181      }
182   }
```

Note that we do not discuss this code in detail here, a overall outline of the work performed by this code has already been given in 4.2, and we provide a more in depth discussion of the different tasks in this code later, in 7.3.2. Here, we limit our discussion to the demarcation code for the rollback concern.

To start the `undoTransfer` method as a secondary transaction, quite a large amount of demarcation code is required, the code starts at line 132 and ends at line 150. Furthermore, this code is quite complex, requiring an instance of an anonymous inner class to be created (as a `Runnable`), so that the `undoTransfer` method can be executed in a different thread (in lines 135 to 144). Also, a number of dependencies have to be placed on the secondary transaction (lines 146-148), i.e. the transaction to be run by this instance, before this instance is run (line 150). Otherwise the secondary transaction would immediately start executing, which is not correct as it should only run when rolling back the saga. Only after these tasks have been performed, among others, the actual `transfer` step can start.

As in the first step of the saga, the second step declares a compensation step: the printing of a transfer cancellation notice, which we already have discussed when first making the transfer operation transactional. For the sake of the example, we assume that a method implementing this has been defined on the current printer, as we did in the first transactional version of the money transfer operation above. This has as a result that we do not need to implement an extra semantical undo method for this secondary transaction, and instead call the printer method directly from the

inner class we create for the secondary transaction. The most noticeable impact in the code, as seen in B.5.4, is the growth of the inner class responsible for the secondary transaction This now makes the method of this inner class transactional in itself, as this transactional behavior should not be placed in the code of the printer.

An interesting issue of the code in B.5.4 is the receipt number, which is a unique number printed on the receipt, that is generated within the body of the `printReceipt` method. This receipt number also has to appear on the cancellation notice, to indicate to which receipt the cancellation applies. However, as the receipt number is generated within the body of the `printReceipt` method, it can not be passed as a method parameter to the cancellation code. This is because the cancellation method is called, as a secondary transaction, before the body of the `printReceipt` method is executed. An indirection trick is used to implement this: when starting the secondary transaction, we pass a container to the cancellation method that is filled in by the body of the `printReceipt` method. This indirection allows this value to be shared between the two methods, so that it can also be printed on the cancellation notice. We know that, at run time, the compensating transaction will only run after the `printReceipt` transaction has committed, which implies that the container will have been filled in, and therefore this will never produce an error.

This ends our discussion on the demarcation code for the bank transfer operation as a saga. We first split the method into four distinct methods, and added demarcation code to these four methods. We noted that this implies that this code is now more scattered than before. We have outlined the most important features of this demarcation code: the considerable size and complexity and the need to be able to share information between transactions and secondary transactions.

7.3.2 Concerns in Demarcation Code for the Saga

Having outlined some of the peculiarities of the demarcation code for the bank transfer operation as a Saga, we now take a step back and perform a conceptual analysis of this demarcation code. If we compare the amount of demarcation code to application code addressing the core concern, we first see that much more demarcation code is present. The original code for the `moneyTransfer` operation counted 37 lines, whereas the version using Sagas counts 267. This amounts to over 7 times as much code, solely as a result of using the Saga ATMS, or over 200 lines of demarcation code. However, size is not the only issue, more important still is the complexity of the demarcation code, which we discuss here, and we start by reviewing the demarcation code for the different steps of the saga.

In the different methods, i.e. steps, of the saga, we discern the five phases of the demarcation code we introduced in 4.2: preliminaries, beginning, running, commit and abort. We now elaborate on the different tasks performed in each of these phases and

locate these tasks in the transfer example code given previously, by including their line numbers:

Preliminaries The preliminaries start with obtaining a transaction identifier from ATPMos (Line 125). This identifier is then used to register this transaction in the dependency mechanism of ATPMos (Line 126). Naming is performed to obtain a reference to the top level and grouping is used to add this transaction to the group of steps of this particular saga (Lines 127-128). If a secondary transaction is required the preliminaries also include the creation of an instance of the inner class for that transaction (Lines 130-141).

Beginning First in the beginning phase is the setting of dependencies (Lines 146-148), and second is calling the `run` method of the secondary transaction, i.e. spawning the secondary transaction (Line 150). Dependencies are set before the secondary transaction is spawned, as these restrict the running of the secondary transaction. Third is verification of dependencies for this transaction (Lines 152-158), which might require the transaction to immediately abort (Lines 156-157), and fourth is the call to ATPMos to begin the transaction (Line 154).

Running The running phase simply consists of the original method code that uses wrappers for transactional data access (Lines 160-166).

Commit Following the running phase, the commitment phase first verifies dependencies (Lines 168-170), which may cause the transaction to abort (Line 170). Second the dependencies for commit time are set (Lines 172-173), and third the transaction is committed (Line 175).

Abort Aborting the transaction starts with verifying dependencies (Line 178), which is known to always succeed, then instructs ATPMos to rollback the transaction (Line 179), and ends by throwing an exception (Line 180).

We have chosen the terminology of tasks to name the work performed in the different phases above, and have not called these tasks concerns here. This is because these tasks are at a low level of the implementation of the model, and do not reflect the conceptual concerns of the demarcation work being performed. Conceptually, the demarcation code above tackles two concerns, namely the structure and rollback concerns we identified in 4.1.1, and we review both next.

The structure concern consists of maintaining the structure of the different steps of the saga. This ensures that each method corresponds to one step, starting and ending the transaction for that step when necessary, and ensuring that a representation of this transaction is present in the dependency model of ATPMos.

Management of rollbacks is the second concern we identified and is treated by the bulk of the demarcation code above. Rollback is ensured by the secondary transactions, which are all compensating transactions. All the dependencies set in the different steps of the saga ensure that these counter-steps run in the right sequence to perform rollback.

Note that we have not reviewed the code for the top level of the saga, this code is mainly responsible for the structure concern. It makes the top level of the saga available for the different steps, and removes the representations of the different steps in the dependency model of ATPMos when the saga ends.

Tangled Demarcation Code

An important issue we are faced with is that there are not only many tasks to be performed in order to implement these two concerns, but furthermore the demarcation code which implements these tasks is also tangled, as identified in 4.2. For example, the tasks in the beginning of preliminaries phase treat the structure concern and at the end of the phase end with the rollback concern. This is followed by the begin phase where first the rollback concern is treated and then again the structure concern. In other words, in one part of the demarcation code first we have code for the structure concern, then for the rollback concern, and then again for the structure concern, which is a clear case of code tangling.

In the saga ATMS the code for only two concerns is tangled, so this code tangling seems not so large an issue. If we consider other ATMS, however, as we did in 4.1.2, we find ATMS which treat more concerns, such as Relatively Consistent Schedules (RCS). In RCS, two additional concerns are present: management of views and performing delegation. In RCS, the same region of demarcation code as the example above needs also to treat these concerns, as delegation is performed and a view is set at begin time. The same phases in the demarcation code therefore now contain four tangled concerns.

Compare this to classical transaction management, which we discussed first. Not only is there less demarcation code, but there is no tangling. In classical transaction management only two concerns are present: structure and rollback. The implementation of the rollback concern, however, is non-existent. This is because the structure concern is responsible for calling the TP monitor to rollback the transaction, and no other actions need to be taken for the rollback. Therefore, as there is only one concern present within the demarcation code, this code is not tangled.

Following the principle of separation of concerns, we can clearly conclude from the above observation that the demarcation code for ATMS is much more difficult to write and maintain, due to the large volume of demarcation code in which multiple concerns are tangled. Therefore, to enable the use of an ATMS, so that the application

programmer can use a transaction model tailored to the design of the application being built, we need to address the issue of demarcation code.

7.3.3 Conclusion

In this section we described how the code for the money transfer operation was changed by splitting the `moneyTransfer` method into four parts, and by adding demarcation code to make use of the Saga ATMS.

A striking property of the resulting code is its size: with 267 lines it is over 7 times as large as the version without transaction demarcation of 37 lines. Or, from a different perspective, this code contains over 200 lines of transaction demarcation code. This transaction demarcation code is quite complex, and can be subdivided in preliminaries, beginning, running, commit and abort phases, where each of these phases performs multiple tasks.

Furthermore, considering the two different concerns present in the Sagas ATMS: structure and rollback, which we identified in 4.1.1, we confirm the analysis of 4.2. We see that the demarcation code in the above example indeed treats these concerns in a tangled way, making this demarcation code complex.

The above allows us to conclude that that the demarcation code for ATMS is much more difficult to write and maintain than demarcation code for classical transaction management. Consequently, to be able to use an ATMS, this demarcation code is in dire need of modularization. Furthermore, we now need to take into account that, in addition to modularizing the concern of transaction management, the different concerns within the ATMS also should be modularized. This allows for easier modification of the properties of the ATMS itself. This enables an existing ATMS to be tailored to better fit the design of the application being built, or a new ATMS to be more easily created specifically for that application.

To ease implementation of maintenance of the demarcation code for ATMS, we have chosen to use AOP, which allows us to achieve a better separation of concerns, i.e. a better modularization of demarcation code for ATMS. As we have shown in 3.2, and observed again above, AOP allows us to achieve a clean separation of concerns for classical transactions. Separating the aspect itself into different modules, however, cannot be achieved in a conceptually clean way using current popular AOP tools, as we have analyzed in 4.3. We therefore have to build our own aspect language and weaver, which will support modularization of the ATMS aspect itself, and we discuss this in the next chapter.

7.4 Conclusion

This chapter started with introducing the TP Monitor we built to be able to provide runtime support for a wide variety of ATMS. This TP Monitor, called ATPMos, is able to support a wide variety of ATMS because it implements the ACTA primitives, which we discussed in the previous chapter. We gave an overview of the implementation of ATPMos, starting by considering ATPMos solely as a TP Monitor for classical transactions. We then identified the required features for ATMS support, outlined how these are implemented in ATPMos, and given an overview of the interface to ATPMos.

With ATPMos as a TP Monitor, we illustrated the process of making a middleware application transactional by use of a bank transfer example. We have shown that using traditional solutions to add transaction demarcation code result in the implementation of both concerns, i.e. the base code and the transaction concern, to be tangled. We confirmed that when using AOP the amount of tangled code is reduced to a minimum, making AOP a superior solution to modularize transaction management. We also repeated the observation in 3.2 that the design of the application should take transaction management into account.

The final part of this chapter started with the observation that it would be better to use the Sagas ATMS to make the bank transfer transactional. We outlined the demarcation code that is required to employ the Saga ATMS, when using ATPMos, and talked about the important features of this demarcation code: the large size and significant complexity. We further discussed the complexity of this demarcation code, identifying the different steps in this code, as introduced in 4.2. We confirmed the observation of 4.2 that this demarcation code itself treats multiple concerns in a tangled way. This lead us to conclude that demarcation code for ATMS is even more in need of modularization than demarcation code for classical transactions.

Such modularization, however, needs to explicitly take into account that an ATMS itself is composed out of different concerns and allow these to also be modularized. While we have shown that AOP is a superior solution to achieve modularization of demarcation code for classical transactions, we have also identified earlier, in 4.3, that existing, popular, AOP tools do not provide for such modularization. We require this modularization, as we must be able to easily make modifications to the ATMS, to have a better fit to the application being built. Therefore, we have created our own aspect language and weaver, specifically for the modularization of ATMS, which do allow the implementation of the ATMS itself to be composed out of different modules. We discuss these in the next chapter.

Chapter 8

KALA: Kernel Aspect Language for ATMS

About the use of language: it is impossible to sharpen a pencil with a blunt ax.
It is equally vain to try to do it with ten blunt axes instead.
— **Edsger Dijkstra**

In the previous chapter, we illustrated how the large size and significant complexity of demarcation code for ATMS makes implementation and maintenance of applications using these models a difficult task. We have also seen that the same phenomenon, albeit on a much smaller scale, is present in classical transactions, and how Aspect-Oriented Programming successfully tackles this by modularizing transaction management. Therefore, to ease implementation and maintenance of applications using ATMS we have developed an aspect language and aspect weaver for ATMS.

An important goal for this language is to enable easy definition and modification of ATMS code, by allowing the different concerns within the ATMS to be modularized. We investigated these different concerns in chapter 4 and discussed the impact this has on the aspect language and weaver.

In this chapter, we introduce the KALA language, the aspect language we developed to enable such separate specification of demarcation code for ATMS. We first give an overview of KALA, detailing its features, and then discuss the aspect weaver we created for KALA, showing how KALA specifications are woven into the base code.

8.1 KALA: The Language

Our aspect language, called *KALA* (**K**ernel **A**spect **L**anguage for **A**TMS), was specifically created to allow for the separate specification of transactional properties of Java methods, and for these specifications to be modular with regard to the different concerns present in ATMS. KALA is based on the ACTA formal model, which ensures that a wide variety of ATMS can be implemented using KALA. In conjunction with KALA we created an aspect weaver, that will weave KALA specifications into the base code of an application, inserting transaction demarcation code that uses ATPMos as a TP Monitor, and transactionally accesses Enterprise JavaBeans.

A major goal for KALA is the ability to modularize the code for an ATMS by defining the different concerns of that ATMS in separate modules. This brings the benefits of separation of concerns to the definition of an ATMS. As a result, development of an ATMS is eased and a given ATMS also can be more easily modified to better fit a particular class of applications. In this chapter, however, we do not discuss modularization. This is discussed in the next chapter, which is dedicated to this topic.

In KALA, the programmer will declaratively state the transactional properties of a Java method in one block of statements, using the constructs provided by the ACTA formal model. In other words, in KALA, dependencies, views, and delegation of a given transaction are defined through declarations made for the corresponding method. Such declarations can be set to coincide with any of the significant events of a transaction, i .e. begin, commit and abort. Therefore, the main body of KALA declarations for a given method contains begin, commit, and abort statements, and each of these statements contains a nested block of dependency, view and declaration statements.

To be able to set dependencies, views and delegation, identifiers for source and destination will be obtained through the naming and grouping service of ATPMos, to which KALA also provides access. Naming and grouping can be performed both in blocks for the begin, commit, and abort statements as in the top-level declaration block for a method. Declarations at the top level block are executed before the method starts, and the result of name lookups are accessible within the begin, commit, and abort statement blocks.

The use of a KALA program fully automates the process of checking dependencies. The resulting executable will automatically ensure that the transaction is added to the dependency model of ATPMos. Also, dependencies are verified at begin, commit and abort time. This is implemented in the woven code by making the required calls to ATPMos, as we discussed in 7.1.2. As a result, at begin, commit and abort time, the transaction will wait either until all dependencies are satisfied and it may proceed, or until it is forced to commit or abort to satisfy a dependency. Having this process fully automated frees the KALA programmer from this burden, so that he can concentrate

solely on declaring the transactional properties of the method.

KALA greatly eases the use of secondary transactions, by allowing them to be declared through a specific `autostart` statement. The use of this statement, in the top level block of KALA statements for a method, fully automates starting of this transaction in a separate thread. The `autostart` statement also allows for separate transactional specifications, i.e. KALA code, to be given for that particular execution of a secondary transaction.

Finally, KALA also supports termination of transactions and groups of transactions, as implemented by ATPMos. Termination can be performed at begin, commit and abort time.

We will now discuss the KALA language in more detail, starting with naming and grouping, second discussing dependencies, views and delegation, third treating automatically starting secondary transactions, and fourth presenting how transactions are terminated. We will not give a formal specification of the KALA grammar here, as it is given in appendix C, instead we discuss the language more informally.

8.1.1 Naming and Grouping

KALA programs declare transactional properties of a method separately from that method's definition, in a separate program file. Therefore, a static naming scheme is required to identify for which method a block of properties is intended. Also, to place dependencies, set views, and perform delegation between running transactions, we need a dynamic naming and grouping scheme, as already discussed in 7.1.2. In this section we describe how static and dynamic naming and the use of groups is programmed in KALA.

Static Naming

Statically, we need to identify with which method a block of transactional properties is associated. To name the method, the full class name and the method signature, separated by a dot, are given. For example, consider two methods, with as signature `fatherMethod(int)` and `childMethod(float, String)`, the first of class `org.tree.RootNode`, the second of class `org.tree.ChildNode`. The static name of both methods, respectively, would be `org.tree.RootNode.fatherMethod(int)` and `org.tree.ChildNode.childMethod(float, String)`.

This already enables us to write our first KALA program, given below, in which we declare that both these methods are transactional, without declaring any specific properties, as seen by the empty block of declarations following their name.

```
1 org.tree.RootNode.fatherMethod(int){
2 }
```

```
3 org.tree.ChildNode.childMethod(float, String){
4 }
```

Dynamic Naming

Now suppose that we wish to implement nested transactions, where `childMethod` will be a sub-transaction of `fatherMethod`, and that `fatherMethod` directly or indirectly calls `childMethod`. To realize this, some dependencies, view and delegation declarations will need to be made, with as sources and targets the running instances of these transactions. Therefore, a dynamic naming scheme is required, to identify transactions at run-time.

To obtain a reference to a registered transaction at run-time, KALA code uses the ATPMos naming service, which we outlined in 7.1.2. Lookup is performed in KALA by using an `alias` statement, which takes as first argument the identifier to which the resulting reference will be bound, and as second argument the key for the lookup operation. As key a Java expression, enclosed between angular brackets, is given. Within this expression, existing aliases can be used, along with the keyword `self`, which stands for the transaction to which this declaration applies at run-time[1]. If no identifier is found for that key, the resulting reference will be to the null transaction, as in ATPMos.

Registration to the naming service is performed by adding a `name` statement to the transactional properties, with as first argument the transaction to be registered, and second the name, given as a Java expression, enclosed between angular brackets, as above. The transaction to be registered, is either the result obtained through an `alias` statement, or the `self` keyword. As in ATPMos, registering a transaction overwrites the previously held binding, and registering the null transaction for a binding, removes that binding from the system.

Note that the above implies that KALA does not provide support for detecting accidental overwriting of a name. If two different transactions unintentionally register themselves under the same name, this error will not be detected and therefore not be reported. We have decided to omit such a feature, as it is not essential to this dissertation. Instead we consider this as future work, and we discuss this in 12.3.

The `name` and `alias` statements can be used within the blocks of declarations of the begin, commit and abort statements, and also in the top-level block of KALA statements for a method. The top-level statements are executed before the transaction begins, and the result of `alias` lookups are visible within the the blocks of declarations for the begin, commit and abort statements. `name` and `alias` within begin, commit and abort is performed at that point in the life-cycle of the transaction, and the results of `alias` lookups are only visible at that specific time.

[1]The contents of both the aliases and of the result of the `self` keyword are of type Integer

Note that the run-time execution sequence of name lookup and registration is identical to the static sequence of `alias` and `name` statements in the KALA program, as within each of these statements references can be made to the results of previous statements.

As an example of dynamic naming, we can expand the code above to let the child method obtain a reference to the father method. Suppose that `fatherMethod` directly or indirectly calls `childMethod`. This has as a consequence that both are running in the same thread and that therefore the Java expression `Thread.currentThread()` will evaluate to the same value. If we let the `fatherMethod` register itself under `Thread.currentThread()`, the `childMethod` can obtain a reference to the father using the same expression, as implemented below:.

```
1 org.tree.RootNode.fatherMethod(int){
2    name(self <Thread.currentThread()>);
3 }
4 org.tree.ChildNode.childMethod(float, String){
5    alias(father <Thread.currentThread()>);
6 }
```

Group Support

Dependencies, views and termination, which we will discuss in detail in 8.1.2 and 8.1.3, can also take as arguments a group of transactions, the members of which are determined at run-time. This is required as some operations need to affect a group of transactions, the number of which can not be determined statically. As an example of such a group, we consider the children of a parent transaction in Nested Transactions. The population of this group can depend on the application logic, and is therefore, in general, impossible to determine statically. Group support is realized by using the group service provided by ATPMos, analogous to the support for dynamic names, as can be seen here.

A transaction can be added to a named group, through the `groupAdd` statement, which takes the same arguments as an `name` statement: as first argument the transaction to be registered is given, and second the name of the group is given, as a Java expression between angular brackets. References to groups are obtained though the `groupAlias` statement, which is analogous to the `alias` statement above, but now looks up the contents of groups that have been previously registered using the `groupName` statement.

The grouping statements, which we defined above, can be used at begin, commit and abort time, and in the top-level block of KALA statements of a method, which is as in the use of run-time names. The scope for `groupAlias` statement is the same

as the scope for the `alias` statement: all declarations if performed at the top level, or the declarations of a significant event if performed at a significant event.

For example, as suggested above, we can let child transactions add themselves to a group, the name of which starts with the identifier of the parent transaction. We perform this before the child transaction starts using the KALA code below.

```
1 org.tree.RootNode.fatherMethod(int){
2    name(self <Thread.currentThread()>);
3 }
4 org.tree.ChildNode.childMethod(float, String){
5    alias(father <Thread.currentThread()>);
6    groupAdd(self <""+father+"Children">);
7 }
```

This concludes our discussion of naming and grouping in KALA, where we have shown how a static name identifies the method to which a block of declarations applies, and how dynamic naming and groups, as supplied by ATPMos, are available in KALA. Using this, we can now declare dependencies, views and delegation between different transactions, which we see next.

8.1.2 Dependencies, Views and Delegation

In the ACTA formal model, the transactional properties of a method consist of the constructs of dependencies, view and delegation. These constructs are also present in KALA and can be set at begin, commit and abort time, as we have discussed in 6.3. To specify these constructs, they are stated in either a `begin`, `commit` or `abort` block, indicated by the respective keyword before a block embedded within the transactional declaration. We now present the dependency, view and delegation statements, and will subsequently show an example of their use.

A dependency is added, similar to the ACTA and ATPMos form of specifying dependencies, by a triplet of the source, the type of dependency, and the target of the dependency. Source and target are transaction identifiers, looked up using the `alias` statement, or group identifiers looked up using the `groupAlias` statement, or are the `self` keyword. The type of dependency is identical to the name used within ATPMos. A dependency statement consists of the keyword `dep`, followed by a comma-separated list of such triplets, enclosed between parenthesis.

The keyword `view` takes a comma-separated list of such tuples declaring source and destination, enclosed between parenthesis, and adds these views from source transaction to the destination. This destination of a view may be either a transaction, or a group, as above. Views are removed by prepending the destination identifier with

the minus sign '-'. Delegation is specified by the `del` keyword, which takes a tuple of source and target transaction identifiers in its' specification, as in the dependency specification. Delegation with a group as source or target is not allowed.

As an example of the use of dependencies, view and delegation, we can extend the example above into a partial implementation of nested transactions, based on the ACTA specification we discussed in 6.2.4. This is achieved by adding the required dependencies, view and delegation operations at begin and commit time of `childMethod`, as can be seen below. The dependency in line 8 ensures that the parent does not commit before the child has ended. The view set in line 9 allows the child to view the intermediate results of the parent, and delegation in line 10 ensures that the work of the child is committed to the parent.

```
1  org.tree.RootNode.fatherMethod(int){
2     name(self <Thread.currentThread()>);
3  }
4  org.tree.ChildNode.childMethod(float, String){
5     alias(parent <Thread.currentThread()>);
6     groupAdd(self <""+parent+"Children">);
7     begin {
8        dep(parent cd self);
9        view(self parent); }
10    commit { del(self parent); }
11 }
```

We have now presented how dependencies, views and delegation are set at begin, commit or abort time though the use of their respective statements within a `begin`, `commit` or `abort` block, and adapted the example of above to yield an implementation of nested transaction. The next KALA construct, for termination of transactions, will allow us to complete the above code, yielding a better implementation of nested transactions.

8.1.3 Termination of Transactions

As we have said above, an important feature of KALA is that it automates dependency checking between transactions. A consequence of this is that all transactions are automatically added to the dependency model of ATPMos before they begin executing. A transaction is, however, not automatically removed from the dependency model when it ends. This is because other, running transactions may have placed dependencies on this transaction that need to remain in place even after the transaction has ended. The same holds for a global name or group membership of a transaction:

these are not automatically removed from the naming service when the transaction ends, as other transactions may still need to obtain a reference to this transaction. The transaction programmer must therefore manually declare that these references are no longer needed, as we have seen in 7.3.1. In summary, at given times in the execution of an advanced transaction, some cleanup work has to be performed.

Such cleanup consists of ending secondary transactions, removing unneeded transaction models from the dependency sub-system and erasing remaining names from the system. Recall that in 6.3 we noted that such transaction termination is not present in ACTA, as this formal model allows us to presuppose an infinite amount of memory. In KALA, however, we must deal with a finite amount of computing resources, such as memory, and therefore such clean-up work is required. This clean-up is performed in `begin`, `commit` or `abort` blocks through the `terminate` and `groupTerminate` statements, which we discuss here.

The `terminate` statement takes as argument an identifier of a transaction, either the result of an `alias` statement, or the `self` keyword, and removes that transaction from the system. First all naming entries to that transaction are removed and the transaction is removed from all the groups it is contained in, second the transaction is removed from the dependency model, and third, if the transaction has not committed or aborted, the transaction is forced to abort.

Similar to `terminate` is `groupTerminate`, which takes as argument a Java expression as in the `alias`, statement. This terminates all transactions in the given group, as above, and removes that group from the system. A global name to a group, registered using `groupName`, can be removed from the system, without the group itself being removed, through the `groupNameTerminate` statement. This statement takes as argument a Java expression, as above.

We can now complete the code of the nested transactions example, by adding `groupTerminate` and `terminate` statements to the root of the nested transaction hierarchy, at commit and abort time. These ensure that upon commit of the parent, the children are cleaned up, and that if the parent aborts, children are also immediately aborted.

```
1 org.tree.RootNode.fatherMethod(int){
2    name(self <Thread.currentThread()>);
3    commit {
4       groupTerminate(<""+self+"Children">);
5       terminate(self); }
6    abort {
7       groupTerminate(<""+self+"Children">);
8       terminate(self); }
9 }
```

```
10 org.tree.ChildNode.childMethod(float, String){
11    alias(parent <Thread.currentThread()>);
12    groupAdd(self <""+parent+"Children">);
13    begin {
14       dep(parent cd self);
15       view(self parent); }
16    commit { del(self parent); }
17 }
```

This concludes our discussion on termination of transactions by the `terminate` and `groupTerminate` statements, which are 'clean-up' statements to be performed when a (part of) an advanced transaction has ended. Using these statements, we have finished the KALA code for the nested transaction example, giving us our first KALA program.

8.1.4 Automatically Starting Secondary Transactions

Separately from the main control flow of the application, some ATMS require what we call secondary transactions to execute. Instead of being part of the control flow, these secondary transactions run automatically, in a different thread. Dependencies are used to restrict when these transactions may begin, i.e. they only begin when some dependencies are satisfied. These dependencies have been placed on the secondary transaction before it starts executing. This is performed either in the begin phase of the secondary transaction itself, or by other transactions before the secondary transaction has begun. An example of such secondary transactions are the compensating transactions in sagas, which we have implemented as part of the bank transfer operation in 7.3.1. In this example we have seen how these secondary transactions are spawned in a different thread, and how dependencies are placed on them to ensure that they execute in the correct sequence at rollback time.

In KALA, secondary transactions can be automatically spawned by using the `autostart` statement in the main block of declarations for a method. The `autostart` statement takes four parameters:

- The first parameter is the signature of a method, and refers to the method of the same object that implements the secondary transaction.

- The second parameter is a comma-separated list of Java expressions, enclosed between angular brackets, which lists the actual parameters for executing the indicated method.

- The third, optional, parameter is a comma-separated list of local variable names, enclosed between parenthesis, listing variables to be shared, which will be detailed later.

- The fourth, optional, parameter is a block of KALA code, declaring the transactional properties for that method.

The secondary transaction will spawn, in a separate thread, after the begin dependencies of the enclosing method declaration have been set. This ordering allows dependencies to be placed on the secondary transaction, ensuring it runs when required. Spawning of the secondary transaction consists of performing a method call, with as receiver the same object as the caller, to the method indicated in the first parameter of the `autostart`, and as arguments the Java expressions given as second parameter.

Considering the implementation for secondary transactions we can not escape the fact that in some cases extra Java code must be added to the class of the method declaring the secondary transaction. As the secondary transaction performs work conceptually belonging to the application logic, this work belongs to the core concern of the application. Therefore, the implementation should not be made in the transaction aspect, i.e. in KALA, but in the Java code of the application. This is consistent with the discussion in 3.2 where we presented the observation that the design of the application has to take transaction management into account. For example, consider a compensating transaction. This secondary transaction performs an 'undo' action required by the ATMS, which is not necessarily already present in the base application. If this 'undo' action is not present, then this extra method must be implemented by the application programmer.

The method for the secondary transaction indicated in the autostart must be implemented on the same class as the transactional method. This is because if it has been implemented on another class, we can not automatically deduce for what instance of that class the method must be invoked. In these cases, the application programmer will have to write a method on the class containing the transactional method, which invokes the method for the secondary transaction using the correct instance.

Because the secondary transaction is spawned in a separate thread, before the containing method runs, the arguments for the secondary transaction can not include local variables of the containing method. However, as we have seen in 7.3.1, in some case, we need access to these variables. This is addressed by the third parameter of the `autostart` statement, which declares what local variables can be passed through to the Java expressions given as second parameter, to be shared between the two methods. Any Java expression in the second parameter, referring to such a shared variable is restricted to be the variable name. Both the secondary transaction and the

enclosing transaction will now transparently operate on the same variable, ensuring the state remains shared. Concurrent access to that shared variable can be avoided by using dependencies. As an example consider the Sagas example which we discussed in 7.3.1, where the secondary transaction is restricted to begin only after the containing method has ended. This allows the contents of the shared variable to be unknown when the secondary transaction is spawned, and it will be bound by the enclosing transaction when the local variable is assigned.

Since the secondary transaction is itself a transaction, the `autostart` statement also allows for its properties to be defined, through the fourth parameter. This parameter is a block of transaction declarations, where all KALA statements, and the begin, commit and abort blocks can be used. Note that these declarations do not conflict with any other KALA declarations given for that method, i.e. the same method can be used in different `autostart` statements with different properties, and can also be given transactional properties outside of an `autostart`. Also, within an `autostart`, the scoping of names is respected, i.e. the result of `alias` lookups in the top-level block, made before the `autostart`, are accessible in the `autostart` block.

As an example, we do not reuse the nested transactions example of above, as no secondary transactions are used in this ATMS. Instead, we show the KALA definition of the second step of the saga operation, the `Transfer` method, which we discussed in 7.3.1. We do not fully discuss this example code, as we return to it in 9.5.

As an example of the use of `autostart`, we show the KALA code for the second step of the saga example, which we talked about in 7.3.1 and the Java code for which is B.5.4. Here, we only highlight the elements of this code important for this discussion. The code below is the KALA specification for the `printReceipt` method. This code, in line five, specifies a compensating transaction (i.e. a secondary transaction), which is implemented by the `printTransferCancel` method. Line six gives the actual parameters used to call this method. In line seven the `num_receipt` instance variable of `printReceipt` is a shared variable, which allows it to be passed as an actual parameter in line six. The compensating transaction itself also has a number of transactional properties set, which is performed in lines seven through ten.

```
1 Cashier.printReceipt(Account, Account, int){
2    alias (Saga <Thread.currentThread()>);
3    alias (CompPrev <""+Saga+"Comp">);
4    group (self <""+Saga+"Step">);
5    autostart (Cashier.printTransferCancel(Account,Account,int,int)
6       <source, dest, amount, num_receipt>
7       (num_receipt) {
8           name(self <""+Saga+"Comp">);
9           groupAdd(self <""+Saga+"Comp">);
```

```
10        });
11     begin {
12        alias (Comp <""+Saga+"Comp">);
13        dep(Saga ad self, self wd Saga, Comp bcd self);
14     }
15     commit {
16        alias (Comp <""+Saga+"Comp">);
17        dep(CompPrev wcd Comp, Comp cmd Saga, Comp bad Saga);
18     }
19  }
```

In this section we have discussed the `autostart` statement, which allows secondary transactions to be spawned independently from the main program flow. Using the `autostart` statement, such secondary transactions, i.e. method calls, are executed automatically when their dependencies are satisfied. In the `autostart` statement, specific transactional properties for that method, in the context of that `autostart` can be specified, and local variables from the enclosing method can be shared with the method of the `autostart`.

8.1.5 Conclusion

In this section we have introduced the aspect language KALA, which allows for the declarative specification of the transactional properties of a method, and its related secondary transactions. The resulting transactional methods, when woven, will use ATPMos as TP Monitor.

These transactional properties mainly consist of the ACTA concepts of dependencies, view and delegation, which we defined in 6.1 and can be declared at begin, commit and abort time. To set these properties, references to transactions can be retrieved from the ATPMos naming service, providing they have registered themselves previously, which can also be performed in KALA declarations.

When transactions end, their associated datastructures within ATPMos are not automatically deleted, to allow other, running transactions to still refer to them, for example when verifying a dependency. Therefore, manual cleanup of transactions, and of transaction groups is required, and KALA also allows for such cleanup to be specified at begin, commit and abort time.

Lastly, to enable secondary transactions to run outside of the main control flow of the application, these can be automatically started from within KALA code for a method. These declarations specify a method, to be called in a separate thread, its actual parameters, optionally what local variables from the enclosing method are shared, and optionally the transactional parameters for that method.

Having presented the KALA language, we will now give an overview of the aspect weaver we have built, which weaves transaction demarcation code into the base-level code, producing transactional methods that issue calls to ATPMos to perform transaction management.

8.2 An Aspect Weaver for KALA

An aspect language itself has no use if programs in this language can not be woven into the base code by an aspect weaver. Therefore, along with KALA, we have constructed an aspect weaver for KALA programs. The weaver performs source-code modifications of the base Java code, adding transaction demarcation code that uses ATPMos, according to the specifications given by the KALA code. In this section we will give an overview of what modifications are made to the source code without giving implementation details of our weaver, as they are not important for this discussion and have already been discussed in [FM04, FG04]. We conclude the discussion by stating why we had to build our own weaver, and could not reuse an existing weaver to perform this work for us.

We discuss our weaver in four parts, for clarity: First we outline how methods are made transactional recalling the skeleton demarcation code we mentioned in 4.2, then we will talk about the `begin`, `commit` and `abort` blocks, which is followed by naming and groups, and we end with autostarts. Note that this does not imply that the weaver performs four passes in this sequence on the source code, this structuring is only for clarity of the discussion. In each part, we will return to the conceptual subdivision of the transaction demarcation code in six phases, as we have seen in 7.3.2, because the resulting code bears much resemblance to the code we presented in that section. Recall that these phases are as follows: first the preliminaries, then the begin phase, followed by the running phase, then either the commit phase or the rollback phase takes place, and the code ends with the cleanup phase. We will detail the source-code modifications made by the weaver to each phase, or skip the phase if no modifications are made.

8.2.1 Making a Method Transactional

The first task of the weaver is to make the method transactional, which in fact boils down to creating the skeleton demarcation code we introduced in 4.2. This is performed by inserting demarcation code for beginning and ending the transaction, modifying the original method's code, such that accesses to entity beans are made transactional, and adding the code that performs dependency verification. In this skeleton structure the three following tasks will add their modifications.

A flow chart illustrating this skeleton code, outlining the different phases, the sequence of operations in each phase, and the places to be filled in for each phase, is given in figure 8.1, which is essentially the flow chart in figure 4.1 (discussed in 4.2). The implementation of the skeleton in pseudo-Java is given in figure 8.2.

Conceptually, the important elements of the skeleton are the calls to ATPMos to register the transaction, begin, commit and abort the transaction, which surround the code of the original method. The remainder of the skeleton code consists of supporting logic which implements the decision points of the flow chart in figure 8.1. This logic is in place to ensure that the transaction aborts if required and that dependencies are enforced.

The skeleton structure, which is filled in by the following phases, is created by taking the original body of the method, using it as the body of a `try` statement, and creating a new body for the method with as sole statement the `try` statement just mentioned. In that `try` statement, a single `catch` clause is present, for transactional exceptions, but with an empty body. In other words, the new body of the method consists of the old body, surrounded by a `try-catch`. To this code, further modifications are performed, which we discuss phase per phase.

The preliminaries start with suffixing the signature of the method with an extra throws declaration, as the method may now throw an exception due to transaction management. We explicitly declare an exception to be thrown by this method, as callers to the method have to include code that recovers from a transaction rollback to ensure correct application behavior for these cases. Explicitly declaring this exception (i.e. not having it a subclass of `RuntimeException`), allows the compiler to verify this when it is performing type checking, and will cause compilation to abort if this exception is not caught.

Next in the preliminaries is the body of the method, where statements are inserted at the beginning to obtain the transaction identifier, and to register the transaction to the dependency model of KALA.

In the beginning phase, the `mayBegin` method is called to establish if the transaction may start and if a forcing indication is returned, it is kept as an instance variable for later use. If no forcing is returned, a call to ATPMos is made to begin the transaction.

The running phase consists of the code within the `try` block, i.e. the previous body of the method. This code is prefixed by a check on the forcing indication. If the transaction must immediately abort, an exception is thrown, to be caught in the catch block, as we will see further on. If the transaction must immediately commit, the original body of the method is skipped. If the transaction may proceed normally, the original body of the method is executed, with some modifications. These modifications boil down to the use of wrappers, as we have seen in 7.2.3, with all of the wrappers automatically generated by the weaver.

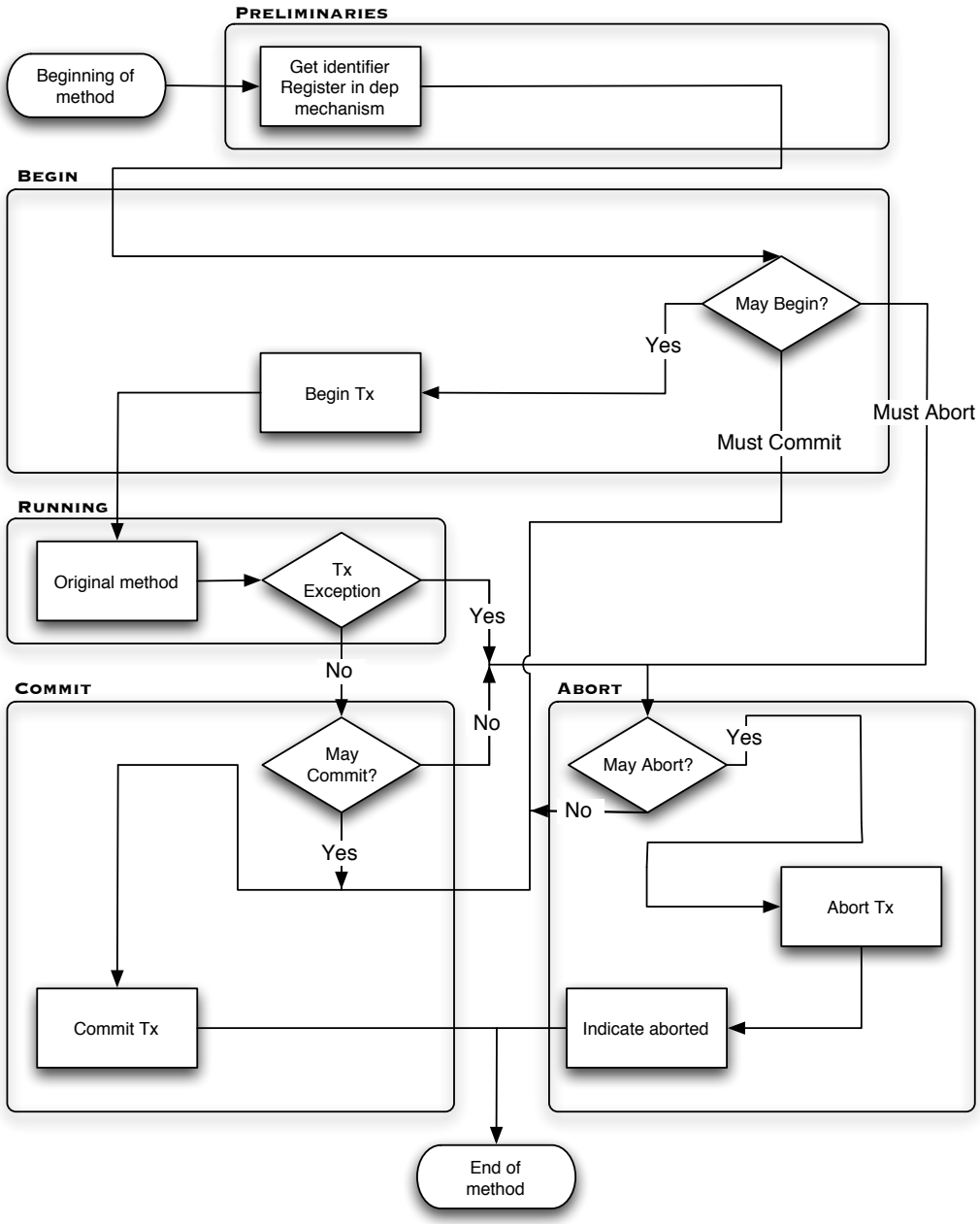

Figure 8.1: Flow chart of woven code skeleton, to be filled in by the following tasks

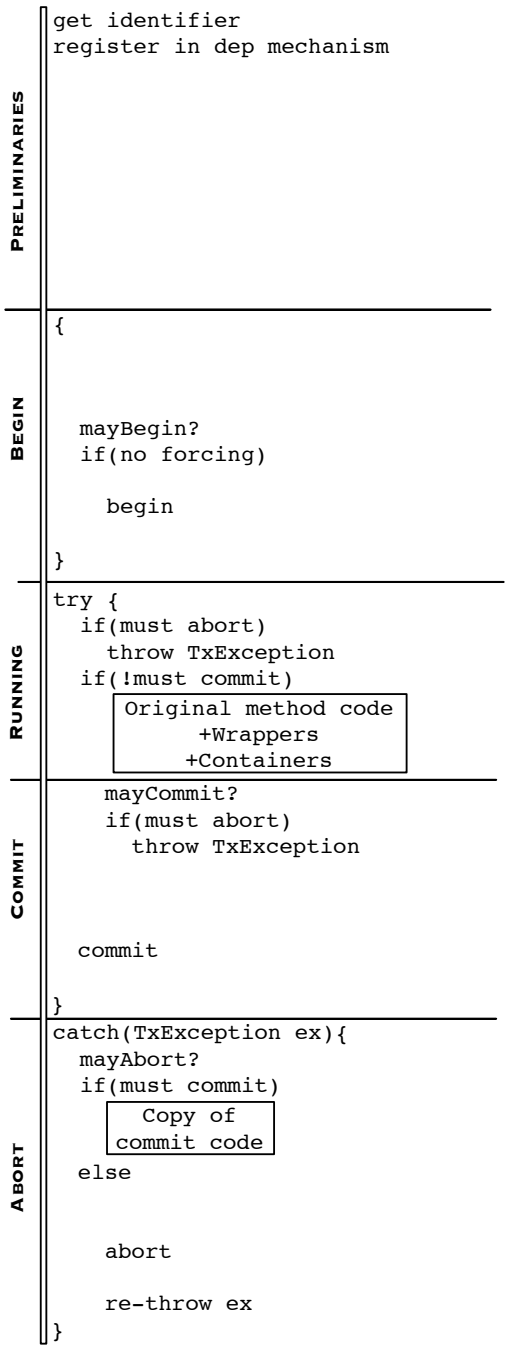

Figure 8.2: Outline of structure of woven code skeleton

This brings us to the commit phase, located at the end of the `try` block. A call to `mayCommit` is made, and if a forcing indicating to abort is returned, an exception is thrown, as above. The last statement of the try block is a call to ATPMos to commit the transaction.

The abort phase is located in the `catch` clause of the `try` statement. It starts with a call to `mayAbort`, and if a forcing indicating to commit is returned, the transaction is committed, as above. If no such forcing is returned, a call is made to ATPMos to abort the transaction and the exception is re-thrown to inform the caller.

This ends the modifications made to the code to ensure transactional access to the data, which resulted in a phased structure, shown in figure 8.2, in which the three following tasks will add their modifications. We discuss the first task: treating the `begin`, `commit` and `abort` blocks, next.

8.2.2 The `begin`, `commit` and `abort` blocks

The `begin`, `commit` and `abort` blocks of KALA declarations state dependencies, views, delegation, and termination to be set and performed at begin, commit and abort time. The code skeleton discussed previously is therefore filled in with code setting dependencies, views and performing delegation. A flow chart depicting the result of this operation is given in figure 8.3.

In the beginning phase, dependencies are inserted before the `mayBegin` call, This is because these may contain dependencies that influence the beginning of the transaction, which are verified by the `mayBegin` call. After the `mayBegin` call, views are set and delegation is performed. Finally, after the call to ATPMos to actually begin the transaction termination is performed through the appropriate calls to ATPMos.

For the commit phase, before the call to ATPMos to commit the transaction, code is inserted to add dependencies, set views, and perform delegation. After the call to commit the transaction, code is added to perform termination. Furthermore, the same manipulations are done in the abort phase, around the call to commit the transaction as a result of a forcing indication. This is because this is a duplicate location in the code for the commit phase.

In the abort phase, similar code manipulations as for the commit phase take place. Before the call to ATPMos to abort the transaction, code is inserted to add dependencies, set views, and perform delegation. After the call to abort the transaction, code is added to perform termination of transactions and groups.

8.2.3 Naming and Groups

Naming and groups are used to obtain references to transactions or groups of transactions that are used in the `begin`, `commit` and `abort` blocks, as arguments to de-

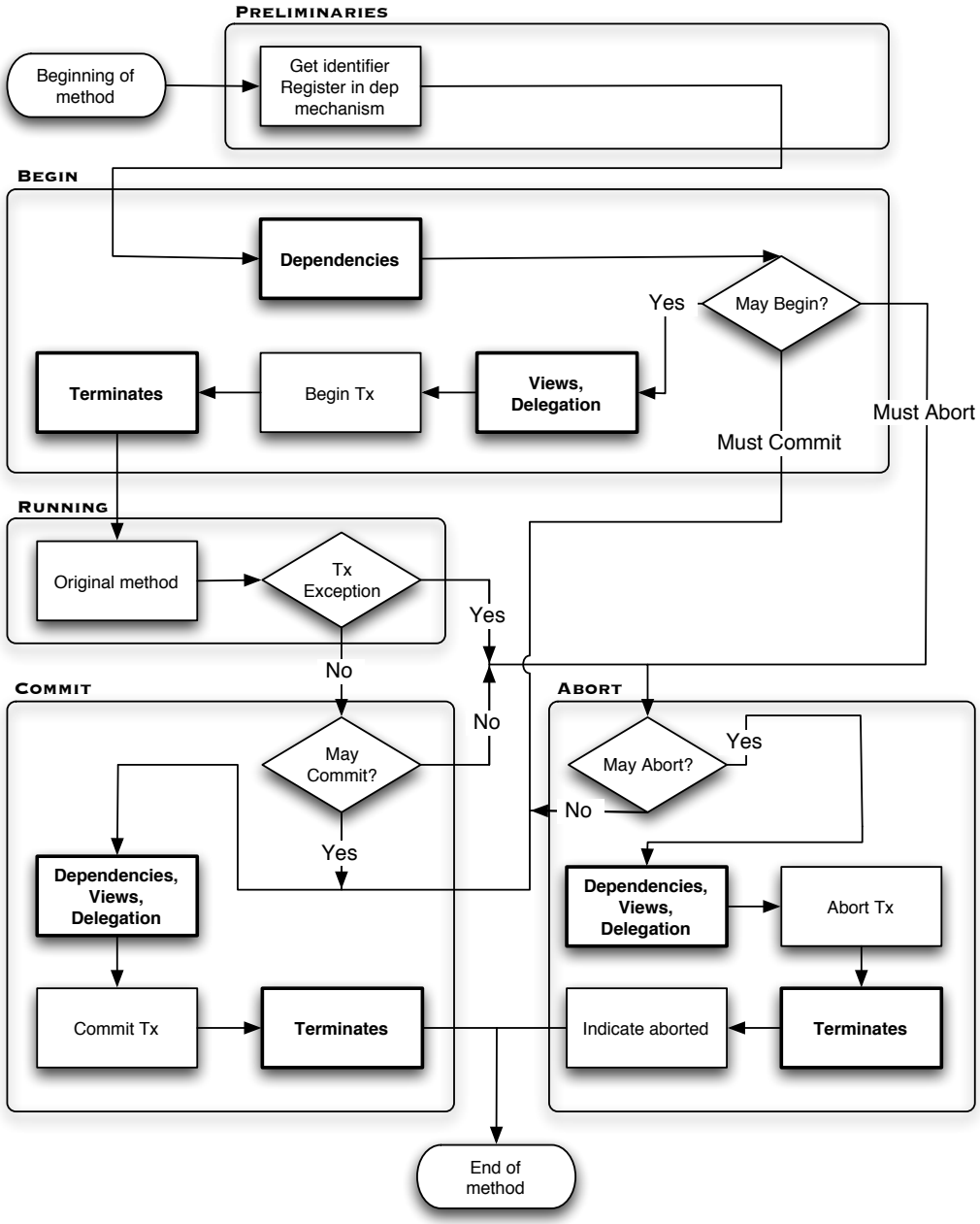

Figure 8.3: Flow chart of woven code skeleton, with dependencies, views, delegation and terminate (shown in bold) filled in.

pendencies, views, delegation and termination statements. Figure 8.4 shows the flow chart of the woven code, with naming and groups filled in.

Names and groups that are used outside of the `begin`, `commit` and `abort` blocks lead to code insertion at the end of the preliminaries phase. The result of `alias` or `groupAlias` lookups are bound to instance variables with the name given in the alias. These are therefore accessible in the `begin`, `commit` and `abort` blocks, and in any subsequent name statements, as the ordering of naming and grouping statements in the KALA program is respected. `name` statements are translated to a call to the naming service of ATPMos, with the given arguments, as are `groupAdd` and `groupName` statements.

Considering the begin, commit and abort blocks, naming and grouping are performed in the same fashion, at the beginning of the code for each block. Note that, as above, this implies that naming and groups for the `commit` block are duplicated in the abort phase.

8.2.4 Autostarts

Autostarts enable the use of secondary transactions, by calling the respective methods in a separate thread, implementing what we have seen in 7.3.1. Adding autostarts completes the code skeleton by modifications to the preliminaries phase, the beginning phase and the running phase. A flow chart of the final woven code is given in figure 8.5.

To implement sharing of variables through a container, this container is declared as an instance variable in the beginning of the preliminaries phase. At the end of the preliminaries phase, an anonymous inner class is created, in the body of which is a copy of the referenced method, where references to shared variables are modified so they make use of the container. Furthermore, this method code is then subject to weaver manipulations as declared by the KALA statements in the autostart block, i.e. the weaver recursively works on this method code.

In the beginning phase, between placing of dependencies and calling the `mayBegin` method on KALA, a new instance of this class is created and run in a separate thread, with as argument the parameters given in the autostart, passing a container if so directed. This effectively spawns the secondary transaction in a separate thread. In the running phase, all references to shared variables are modified so they use the container, implementing the shared variable.

This completes the description of the work performed by our aspect weaver, rendering a method transactional by adding demarcation code to the method according to the KALA specification for this method. As we have seen, this is quite intricate work, which raises the question of why we did not reuse an existing aspect weaver to perform this work for us, and we address this next.

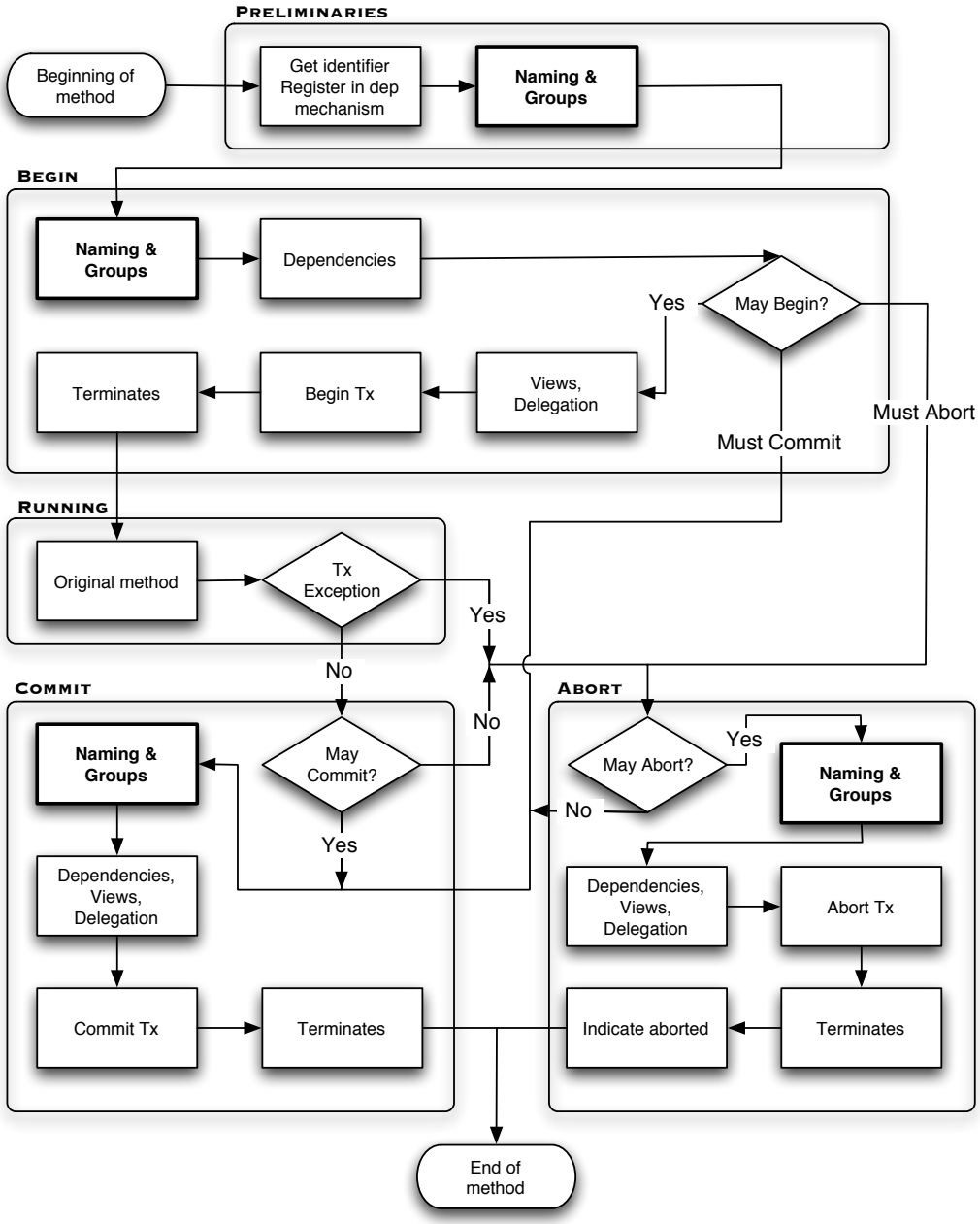

Figure 8.4: Flow chart of woven code skeleton, with naming and groups (shown in bold) filled in.

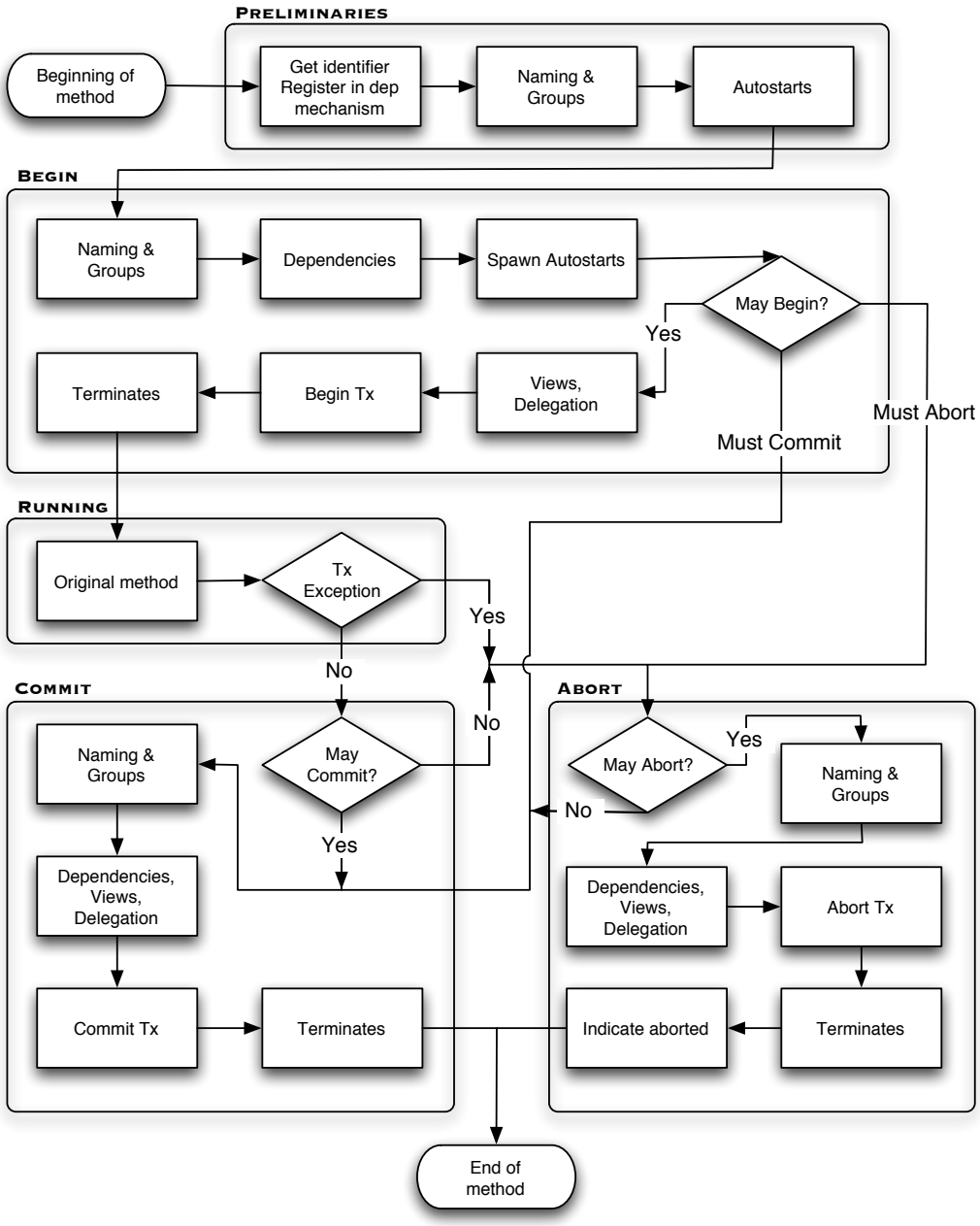

Figure 8.5: Flow chart of final woven code.

8.2.5 Why a New Weaver?

The AOP community has created several aspect languages and associated weavers, so instead of creating our own aspect weaver it would seem to be possible to reuse one of these tools to perform the weaving. We could generate aspect code for these weavers, translating the KALA specification to an equivalent specification in the language of the weaver used, and let this weaver perform the intricate task of ensuring transaction demarcation is performed correctly. The aspect code we generate would be an intermediate representation, never seen by an application or ATMS programmer, and therefore if this code would be tangled aspect code, this is not an issue.

We have investigated the aspect languages AspectJ [Asp05] and AspectWerkz [Bon04], which are arguably the most used and best known aspect languages at the moment, and the less prominent, but still well-known JAsCo [SVJ03] language and JBoss AOP framework [BFB+], to investigate if we could use one of these as a weaver for translated KALA specifications. We found, however, that none of the weavers provided us with adequate technical features to realize this. In other words, we cannot translate KALA code to equivalent aspect code for these weavers, e.g. equivalent AspectJ code. We enumerate the features we miss in these languages here:

Pointcuts for Bean Getters and Setters As we have introduced in 7.1, we perform concurrency management on reading and writing data encapsulated within Entity JavaBeans by prefixing calls to getters and setters of these beans. None of the above tools and languages, however, allow for an intentional pointcut specification defining the getters and setters of a class or interface. Instead, all getters and setters must be explicitly enumerated. Although this is not an issue restricting the possibility of using one of the above tools for this work, we find it a remarkable oversight, as an intentional specification is clearly preferable.

True Exception Throwing The above weavers do not allow the signature of a method to be modified, and this has an impact on exception throwing. If aspect code wishes to throw an exception, this exception must be a subclass of `RuntimeException`, as these exceptions do not need to be explicitly declared in the method signature. The sole exception to this is JBoss AOP, which does allow all kinds of exceptions to be thrown, however without altering the method's signature. A critical issue with throwing runtime exceptions is that type checking for these exceptions is not performed by the compiler. This means that the compiler does not verify if the thrown exceptions are caught, as is the case for all other exceptions, and therefore that the program halts if one of these exceptions is thrown but not caught. Note that this drawback, specifically for AspectJ has already been criticized by Kienzle and Gerraoui [KG02], which we discussed in 3.2. As for Kienzle and Gerraoui, this is a critical issue for our research because

we throw an exception if a transaction aborts, in order to inform the caller of this abort. With type checking for exceptions turned off, it is possible for the program to stop working, if this exception is not caught. If we are able to use exceptions that are not a subclass of `RuntimeException`, this issue would be detected at compile time, and compilation would fail.

Autostarts An autostart runs a given method in a separate thread, and most importantly, weaves demarcation code specific to that execution in that method only. We are unable to define a pointcut specification that expresses the above. Note that using `cflow`-like pointcuts is not applicable here, as the called method runs in a different thread and therefore lies in a separate control flow which we have no access to.

Method Local Variable Access For sharing variable access through a container, we need to be able to define a pointcut on all reads and writes of this variable within the method. None of the above tools allow this. An argument could be made that this breaks the encapsulation of the method by exposing too much of the internal details of the method. However, we note that the same pointcut for the parameters of the method is also impossible, which arguably would not break this encapsulation. In either case, we need to intercept reads and writes to these variables to redirect them to the container, and this is impossible with the above tools.

Method Signature Change In addition to being able to declare extra classes of exceptions thrown by the method, we also need to be able to change the signature of the method. One example is when sharing variable access through a container, the type of the parameter being contained needs to change to the type of the container.

Exception Information Tunneling A number of the above weavers allow aspect code to fulfill the role of an exception handler, i.e. to run if the given method throws the stated exception. This exception handler, however can only access the parameters of the method, and has no access to local variables of the method, as it is argued that this would break encapsulation of the method. But an exception handler can require more information than just the parameters of the method, as is the case for our work. In these cases, it should be possible to allow for some kind of information tunneling, whereby the exception handler is able to break through the encapsulation barrier to inspect the internal state of the method, so it can perform the required corrective action.

Given all the above issues with a number of existing and well-known aspect weavers, we have instead opted to create our own aspect weaver, which does pro-

vide support for all the above, and allows KALA specification to be woven in the method code.

8.2.6 A Word on Language-Independence

We wish to emphasize that although the KALA weaver performs source-code modifications on Java code, this research is also applicable to other programming languages for the base level. Conceptually, the design for the KALA weaver can be used for other object-oriented or procedural programming languages, if the following three elements are taken into account:

Method Signatures in KALA are currently given as Java signatures. These need to be changed to reflect the method signatures of the base language.

Exception Handling is used for transaction rollback. To rollback the transaction, an exception is thrown for which the exception handler is responsible for rollback. The base language will have to provide this level of exception handling support.

Autostarts are implemented through Java multi-threading and inner classes. This implies that first the base language needs to provide multi-threading support. Second, the Java scoping for inner classes must be emulated: an instance of an inner class can see all the local variables of the declaring method[2]. This allows the body of the autostart to see the aliases of the declaring transaction. The same scoping can be achieved by explicitly passing all the aliases to the autostart as extra parameters when spawning the autostart.

8.2.7 Conclusion

In this section we have talked about our aspect weaver, giving an overview of how the Java code of the application is modified to include transaction demarcation code, an outline of which is given in figure 8.6.

We detailed the working of the weaver by conceptually dividing up the work in four phases. First we outlined how making the method transactional creates a code skeleton, in which the phases add their code. For the second phase we detailed adding dependency, view, delegation and termination statements to this skeleton. The third phase consisted of adding naming and groups. The fourth and last phase completed the skeleton with code for autostarts. We ended this discussion with an enumeration giving the reasons why we had to create our own weaver for KALA, and by discussing the language-independence of the KALA weaver.

[2] if these are declared `final`

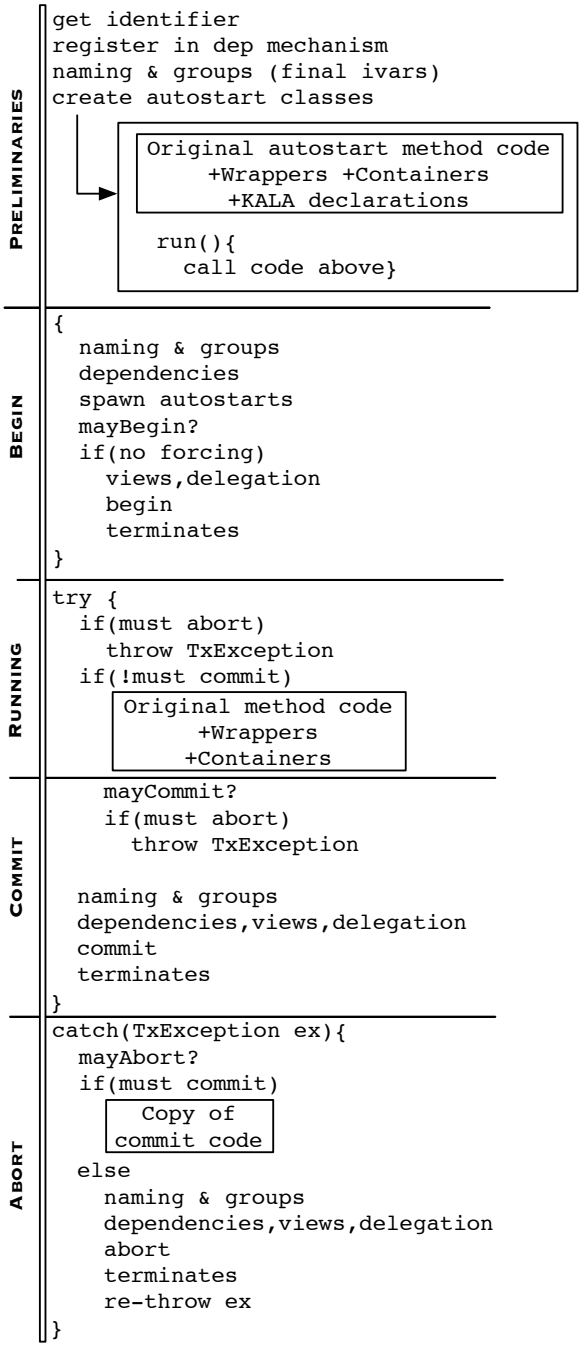

Figure 8.6: Outline of woven code

8.3 Conclusion

This chapter introduced KALA: a **K**ernel **A**spect **L**anguage for **A**TMS. In KALA, transactional properties of Java methods are specified separately, with these specifications highly similar to the ACTA specifications of transaction models.

KALA allows the application programmer to declare the transactional properties of a Java method in one block of statements. This has as a first advantage that the transaction aspect is separated from the core application logic. Furthermore, KALA offers as an additional advantage that checking of dependencies is automated. The programmer no longer needs to implement the code which ensures that dependencies are adhered to. Also, KALA provides explicit support for declaring secondary transactions, which significantly eases their use.

We detailed how the different ACTA constructs: dependencies, views and delegation are available in KALA, and at what points in the transaction life-cycle they can be used. We have shown how ATPMos names and groups are reified in KALA, in support of the ACTA constructs present. We also talked about termination of transactions and groups, which automates the removal of names, groups and dependency models in ATPMos. Lastly we introduced automatically starting of secondary transactions, as provided by KALA.

Accompanying KALA is an aspect weaver, which weaves transactional specifications given as KALA code into the corresponding Java methods. We gave an outline of the modifications that are made to the source code, inserting the required demarcation code. We also detailed why we could not reuse existing, popular aspect weavers to perform these modifcations. Finally we briefly stated how the weaver can be implemented for other programming languages.

Now that we have an aspect language for ATMS and its associated weaver, there is one important feature of this language which was intentionally left undiscussed, which is modularization of KALA specifications. As we said in the beginning of this chapter, KALA specifications can be modular with regard to the different concerns present in ATMS, and we discuss this feature in depth in the following chapter.

Chapter 9

Composing ATMS Concerns With KALA

You did something because it had always been done,
and the explanation was "but we've always done it this way."
A million dead people can't have been wrong, can they?
—**Terry Pratchett, "The Fifth Elephant"**

In the previous chapter, we introduced KALA, the aspect language we created for ATMS. One of the goals of KALA is to enable a modular specification of the different concerns of a given ATMS, as identified in 4.1. This eases the implementation of the ATMS and allows its properties to be more easily modified. As a result, creation and adaptation of an ATMS is facilitated, which enables an ATMS to be tailored to better fit the transactional properties required by the application being developed. This chapter discusses how this goal is achieved through a separate specification of each concern of the ATMS.

KALA allows for the different concerns that compose an ATMS to be written down separately, in multiple KALA files. When weaving KALA code, these separate specifications are first composed to form one ATMS aspect, that afterwards is woven into the source code. In this chapter we first discuss how the different concerns identified in 4.1 can be defined separately, and introduce how composition of these different concerns is performed through a straightforward merging process. As an example for the separate specification and merging of KALA code, we use a KALA implementation of the demarcation code in 7.3.1. After introducing separate specification and merging of KALA code, we further illustrate this for the Nested Transactions and Relatively Consistent Schedules ATMS. This is followed by a demonstration of how this separate specification enables us to create a new ATMS based on Nested Transactions. We

end this chapter by considering how, once an ATMS has been created, the application programmer will write KALA code. This leads us to the following chapter, which introduces ATMS-Specific aspect languages.

9.1 Separate Definition of Concern Code

We wish to modularize the definition of an ATMS because we consider an ATMS not as a fixed monolithic block, but as a composition of multiple concerns, as we analyzed in 4.1. We want to apply the known benefits of separation of concerns, which we discussed in 3.1.1, to the process of creating and modifying an ATMS. Applying separation of concerns here, i.e. programming an ATMS in different modules, will greatly ease implementation and modification of this ATMS. This is a key factor for this dissertation, because we want to enable that a new ATMS is built, or an existing ATMS is adapted, to fit the application being developed.

We need to use KALA to be able to modularize an ATMS because of the identification of tangled aspect code we performed in 4.3: modularization as provided by popular solutions such as AspectJ is lacking and leads to aspect code which itself is tangled. Instead of having to write tangled aspect code, we can now write different KALA modules defining an ATMS, and these modules will be composed to form one aspect definition for the ATMS, as we see later.

It is important to remark that in modularized KALA code, each module implements one concern of an ATMS with regard to the application using this ATMS. In other words, each KALA module describes the transactional behavior of the application solely with regard to one ATMS concern. For example, in an application using the Sagas ATMS, the KALA module for rollbacks will describe the behavior of that application with regard to rollbacks of the saga. In general, a module of KALA code does not implement an ATMS concern as such, but rather how the implementation of that concern is linked to an actual application. As we show in the final part of this chapter, this has its drawbacks, and these are addressed in the next chapter.

In this section, we show how, applied to a given application, KALA can be used to define the different concerns of an ATMS separately. We assume here that an analysis has first been made of the different concerns present in the ATMS being used, as we have performed in 4.1 for a number of ATMS. The different concerns identified in such an analysis, applied to an application, can then be written down separately in different KALA files. We show this by taking the Sagas ATMS we analyzed in 4.1.1, and writing KALA code for this ATMS.

As a concrete example of KALA code for the use of Sagas, we revisit here the example we have shown in 7.3.1. Recall that this example consists of a bank transfer operation, split in three steps: a `transfer`, a `printReceipt` and a `logTransfer`

method, all called from a `moneyTransfer` method. Note that we don't repeat the Java code of the bank transfer operation, instead we refer to B.5.

We identified in 4.1.1 that the Sagas ATMS is comprised of two concerns: first the management of the structure of the overall transaction, and second the management of how rollback is performed. To have an implementation of these transactional concerns for the bank transfer operation, we now write KALA declarations for all four methods first for the structure concern, and second for the rollback concern.

9.1.1 Sagas: Structure

The first concern identified in 4.1.1 which we implement here, is the structure of the saga. Recall that we identified this concern as the management of the overall structure of the saga, in which the different steps perform their work. The structure concern codifies the subdivision of the saga into different steps, allowing each step to obtain a reference to the top-level saga, and ensures that after the saga has ended all steps are removed from the dependency model of ATPMos.

Note that although we subdivide the discussion of the implementation of this concern into two parts, all the code for the structure concern is implemented in one file, as is indicated by the continuity in line number counting.

Saga Top-level

The first KALA declarations we show are for the top-level `moneyTransfer` method and are given below. This code registers itself, using the same technique we have seen in 8.1.1, so that the different steps in the saga, shown later, can obtain a reference to the saga. At commit and abort time, the unique identifier of this saga is used to refer to a group name which is therefore guaranteed to be unique for this saga. In this group, the different steps of the saga will have registered themselves. As a result, termination of this group implies termination of all the steps of the saga, and together with termination of the saga itself ensures proper cleanup is performed.

```
1 Cashier.moneyTransfer(BankAccount, BankAccount, int) {
2    name(self <Thread.currentThread()>);
3    commit {   groupTerminate(<""+self+"Step">); terminate(self); }
4    abort {    groupTerminate(<""+self+"Step">); terminate(self); }
5 }
```

Saga Steps

The code of all the steps of the saga is virtually identical, the only difference being the identification of the method corresponding to each step. Each of these steps first

require a reference to the top-level transaction so as to, second, add itself to the group of steps. By adding itself to the group of steps, it ensures that it will be terminated when the saga ends, by the code either in line 3 or line 4.

```
6  Cashier.logTransfer(BankAccount, BankAccount, int) {
7     alias (Saga <Thread.currentThread()>);
8     groupAdd(self <""+Saga+"Step">);
9  }
10 Cashier.transfer(BankAccount, BankAccount, int) {
11    alias (Saga <Thread.currentThread()>);
12    groupAdd(self <""+Saga+"Step">);
13 }
14 Cashier.printReceipt(BankAccount, BankAccount, int) {
15    alias (Saga <Thread.currentThread()>);
16    groupAdd(self <""+Saga+"Step">);
17 }
```

This concludes the code for the structure concern of the sagas ATMS, applied to the bank transfer example. This code implements the structure of the saga in different steps, with termination of the different steps when the saga ends. The following concern, rollback handling will add the handling of rollbacks of this structure, yielding the behavior of the Sagas ATMS.

9.1.2 Sagas: Rollback Handling

The second concern identified in 4.1.1 which we implement here, is rollback handling for the saga. Recall that in order to rollback a saga, the currently executing step is aborted, and that all committed steps are compensated for by executing compensating steps in the reverse sequence of step execution.

The KALA code below is an implementation of the above concern, and is defined in a separate KALA file. Again, we subdivide the discussion of the implementation in different parts, and the line numbers show this code all belongs to one file.

Saga Top-level

The top level of the saga registers itself, as in the structure concern, because the different steps and compensating steps will place dependencies on the sagas, as we see later. At commit and abort time, the group of compensating steps is aborted, similar to what is performed in the structure concern.

```
1  Cashier.moneyTransfer(BankAccount, BankAccount, int) {
2     name(self <Thread.currentThread()>);
```

```
3     commit {   groupTerminate(<""+self+"Comp">); }
4     abort {    groupTerminate(<""+self+"Comp">); }
5  }
```

Last Step

In the last step of the saga, to implement the rollback concern, a number of dependencies have to be set between the step and the saga when the step begins. Note that, as in 7.3.1 we do not discuss these dependencies in detail, but instead refer to [CR92]. To set these dependencies, in line 8, a reference to the saga has to be obtained, which is performed in line 7.

```
6  Cashier.logTransfer(BankAccount, BankAccount, int) {
7     alias (Saga <Thread.currentThread()>);
8     begin { dep(Saga ad self, self wd Saga, Saga scd self); }
9  }
```

First Step

The first step of the saga needs to spawn a secondary transaction, to be used as compensating transaction when the saga rollbacks. It achieves this by using an `autostart` statement in lines 12 thru 16, which compensates a bank transfer simply by performing the inverse transfer operation. The secondary transaction registers itself under a unique name in line 14, so that in line 18 and 21 a reference can be obtained to this transaction to set the required dependencies. Also, the secondary transaction adds itself to the group of compensating transactions in line 15, ensuring it is properly terminated in line 3 or 4, when the saga ends.

```
10 Cashier.transfer(BankAccount, BankAccount, int) {
11    alias (Saga <Thread.currentThread()>);
12    autostart (transfer(BankAccount, BankAccount, int)
13       <dest, source, amount> {
14          name(self <""+Saga+"Comp">);
15          groupAdd(self <""+Saga+"Comp">);
16       });
17    begin {
18       alias (Comp <""+Saga+"Comp">);
19       dep(Saga ad self, self wd Saga, Comp bcd self); }
20    commit {
21       alias (Comp <""+Saga+"Comp">);
```

```
22        dep(Comp cmd Saga, Comp bad Saga); }
23  }
```

Second Step

The second step of the saga is highly similar to the first step of the saga, the only differences being a different `autostart`, and dependencies being placed on the previous compensating transaction. The `autostart` declares the compensating step for this step, and also uses a wrapper, in line 29 to print the receipt number on the cancellation notice. A reference to the compensating transaction of the first step of the saga is obtained in line 26, which allows an extra dependency on this transaction to be placed in line 38.

```
24  Cashier.printReceipt(BankAccount, BankAccount, int){
25     alias (Saga <Thread.currentThread()>);
26     alias (CompPrev <""+Saga+"Comp">);
27     autostart (printTransferCancel(BankAccount,BankAccount,int,int)
28        <source, dest, amount, num_receipt>
29        (num_receipt) {
30            name(self <""+Saga+"Comp">);
31            groupAdd(self <""+Saga+"Comp">);
32        });
33     begin {
34        alias (Comp <""+Saga+"Comp">);
35        dep(Saga ad self, self wd Saga, Comp bcd self); }
36     commit {
37        alias (Comp <""+Saga+"Comp">);
38        dep(CompPrev wcd Comp, Comp cmd Saga, Comp bad Saga);}
39  }
```

Note that an `autostart`, as we have discussed in 8.1.4, requires the specified method to be defined in the same class as the method for which it is defined, we are now required to add an implementation of the `printTransferCancel` method to the `Cashier` class.

This completes the KALA code of the concern of rollback handling for the bank transfer operation using the sagas ATMS. As there are no more concerns in this ATMS, this concludes the KALA code for this example.

9.1.3 Conclusion

In this section we have shown how the implementation of a chosen ATMS for a given application is modularized using KALA code. In KALA, each module is implemented in a separate file. As an example we have reused the bank transfer operation from 7.3.1, which uses the Sagas ATMS. We have taken the decomposition of the Sagas ATMS, perfomed in 4.1.1, which identified two concerns, and given the KALA code for each of these concerns.

This modularization frees us from having to write tangled aspect code, which brings the benefits of separation of concerns to the process of defining an ATMS as an aspect. Instead of having to write aspect code which itself is tangled with multiple concerns, with all the impediments this entails, we now cleanly separate each concern in a separate KALA module.

To form a complete aspect specification, however, the different KALA files for one ATMS will have to be combined. This composition of separate KALA files to form the full ATMS specification is shown next.

9.2 Composing KALA Code

In the previous section, we have illustrated how different concerns within one ATMS can be written down separately, in different modules. This allows for easy modification of the implementation of a concern, and allows other concerns to be added by adding new modules to the implementation.

To form a complete specification of an ATMS, however, the different modules have to be composed into one KALA program, as we discussed in 4.3. Recall that the different concerns within the ATMS are conceptually combined before the ATMS aspect is woven, because it is the combination of these concerns that forms the definition of the ATMS. The KALA weaver, therefore, is not built to weave each concern of an ATMS separately into the base code. Instead all KALA modules are combined into one KALA file, describing how the ATMS is used, and this full description is woven into the base code. Although this aspect code will be tangled aspect code, this is not an issue since this code is but an intermediate representation which is not presented to a programmer. An ATMS programmer will solely work with the modularized ATMS definitions, we consider the composition of modules as just one phase of the weaving process.

The above is possible because the composition of different concerns was taken into account when designing the KALA language. As a result, composition is a straightforward process which does not require any extra input from the programmer. In fact, composing different KALA specifications in essence boils down to a simple merge, as we show here.

Conceptually, different KALA specifications declare that different actions need to take place at a given time in the life-cycle of a transaction: before the transaction begins, at begin time, at commit time or at abort time. In the composed file, therefore, for each of these moments in the life cycle all the actions defined for that point need to be performed. In other words, for the composition, all the declarations that pertain to one moment in the life time of the transaction have to be gathered into one block of the resulting specification.

The sequence of statements for naming and grouping within this composition matters, however, as a name referred in a KALA statement needs to be defined in a previous naming or grouping statement. Therefore, when composing different KALA specifications, the partial ordering of naming and grouping statements within each KALA file needs to be preserved in the global file.

Considering in more detail the `begin`, `commit` and `abort` blocks of KALA code, we can state that the sequence of the code for setting dependencies, placing views, performing delegation and termination, however, is irrelevant, as their sequence is fixed in the final woven code. Therefore, when composing a number of different `begin`, `commit` or `abort` blocks for the same method, their dependency, view, delegation and terminate statements can be simply joined into one sequence which respects the partial ordering of names and groups. The same observation holds for `autostart` statements, as their sequence in the KALA code is of no importance, all `autostart` statements for one method, in different modules, are placed before the begin block of the composed KALA specification.

We can implement the above composition by a simple merge, the implementation of which is outlined next. Given that we have a number of KALA specifications for one method and we need to generate an output file:

1. Start the output file with the method signature suffixed with {.

2. For each specification, take the sequence of top-level declarations and add them to the output file.

3. Write the start of a begin block to the output file.

4. For each specification take the sequence of begin declarations and add them to the output file.

5. Write the close of the begin block, and the start of the commit block to the output file.

6. For each specification take the sequence of commit declarations and add them to the output file.

7. Write the close of the commit block, and the start of the abort block to the output file.

8. For each specification take the sequence of abort declarations and add them to the output file.

9. Write the close of the abort block and the closing } to the output file.

There is one downside, however, to this simple merging, which is name clashes: it must be avoided that different modules redefine one name. If these modules redefine the name with the same target, as in line 2 of both modules in the sagas example above, this is not an issue. But if different modules define the same name for a different target this will lead to wrongly placed dependencies, views, delegation, and so on, in other words, yielding faulty code. Conceptually, this issue can, however, be easily solved through a renaming or a merge of names. Therefore we do not provide an outline of such an implementation here.

The above is all which is required to compose different KALA modules into one full program, and we show the result of such a composition next.

9.2.1 Sagas

To illustrate the composition process outlined above, we now show how the saga code we defined in separate modules in 9.1 is integrated into one KALA program. As in 9.1, we subdivide the discussion of the saga in different steps, starting with the top level of the saga.

Saga Top-level

The top level of the saga shows how the different commit and abort blocks are joined, and how name resolving leads to one `name` statement in line 2.

```
1 Cashier.moneyTransfer(BankAccount, BankAccount, int) {
2    name(self <Thread.currentThread()>);
3    commit {
4       groupTerminate(<""+self+"Comp">);
5       groupTerminate(<""+self+"Step">); terminate(self); }
6     abort {
7       groupTerminate(<""+self+"Comp">);
8       groupTerminate(<""+self+"Step">); terminate(self); }
9 }
```

Saga Steps

The different steps of the saga illustrate name resolving for the `alias` statement, but beyond that are a straightforward merge.

```
10 Cashier.logTransfer(BankAccount, BankAccount, int){
11    alias (Saga <Thread.currentThread()>);
12    groupAdd(self <""+Saga+"Step">);
13    begin { dep(Saga ad self, self wd Saga, Saga scd self); }
14 }
15 Cashier.transfer(BankAccount, BankAccount, int) {
16    alias (Saga <Thread.currentThread()>);
17    groupAdd(self <""+Saga+"Step">);
18    autostart (transfer(BankAccount, BankAccount, int)
19       <dest, source, amount> {
20          name(self <""+Saga+"Comp">);
21          groupAdd(self <""+Saga+"Comp">);
22       });
23    begin {
24       alias (Comp <""+Saga+"Comp">);
25       dep(Saga ad self, self wd Saga, Comp bcd self); }
26    commit {
27       alias (Comp <""+Saga+"Comp">);
28       dep(Comp cmd Saga, Comp bad Saga);}
29 }
30 Cashier.printReceipt(BankAccount, BankAccount, int){
31    alias (Saga <Thread.currentThread()>);
32    alias (CompPrev <""+Saga+"Comp">);
33    groupAdd(self <""+Saga+"Step">);
34    autostart (printTransferCancel(BankAccount,BankAccount,int,int)
35       <source, dest, amount, num_receipt>
36       (num_receipt) {
37           name(self <""+Saga+"Comp">);
38           groupAdd(self <""+Saga+"Comp">);
39       });
40    begin {
41       alias (Comp <""+Saga+"Comp">);
42       dep(Saga ad self, self wd Saga, Comp bcd self); }
43    commit {
44       alias (Comp <""+Saga+"Comp">);
```

```
45        dep(CompPrev wcd Comp, Comp cmd Saga, Comp bad Saga); }
46  }
```

We have now seen how separate KALA modules are straightforwardly composed to form a complete specification of the usage of an ATMS by a given application. We have given an outline of the implementation of such a composition, and have show the result of composing the example modules created in 9.1 into one KALA program. Note that such composed KALA code is only intended as an intermediate representation, to be woven into the base code.

This composition of KALA code, which requires no programmer intervention, enables ATMS definitions to be easily modularized into different concerns. As a result, the advantages of separation of concerns also apply when developing ATMS. Consequently, in the remainder of this chapter we do not repeat such composed code, but write KALA code in different modules.

Given the description of Sagas as two KALA modules in 9.1, and the above merge into one KALA program, we now repeat the exercise for other ATMS, showing how they can be implemented in KALA.

9.3 Other ATMS Descriptions

To further illustrate the implementation of an ATMS, but abstracting from the application using this ATMS, we now show how we implemented two other ATMS: Nested Transactions and Relatively Consistent Schedules, using KALA.

To implement both these ATMS, we repeat the process of 9.1, where an ATMS is defined in different modules, each module corresponding to one concern of the ATMS. Regarding the concerns present in Nested Transactions and Relatively Consistent Schedules, we use the analysis of 4.1.2. This analysis revealed four concerns present in both ATMS. In addition to the structure of the ATMS and handling of rollbacks, both these ATMS also contain the concerns of view management and delegation of operations. We therefore implement both ATMS using four different KALA modules, one module per concern.

We do not provide example applications to which the KALA code is applied tor the code below. This is because as we solely wish to concentrate on the implementation of the ATMS, without considering how this ATMS is used by an application. We will use placeholder code, which is marked like this, when referring to base-level entities, such as method signatures. When these ATMS are used for a given application, this placeholder code needs to be replaced by the appropriate code for that application.

We start our illustration of ATMS implementations with describing the KALA code for Nested Transactions, and subsequently treat Relatively Consistent Schedules.

9.3.1 Nested Transactions

Nested Transactions is arguably the best-known ATMS, and we already provided an example implementation of this ATMS in KALA, in 8.1.3. Here we provide a more general implementation for this ATMS in KALA, based on a decomposition in concerns. Recall that in 4.1.2 we established that Nested Transactions is composed out of four different concerns: structure, handling of rollbacks, view management and delegation of operations. We now write KALA code for each of these concerns separately. We do not include the result of composing these different modules here, as it brings no significant contribution to this discussion.

Structure

The structure of Nested Transactions is not fixed statically as in Sagas, instead of this, at runtime a tree structure of transactions is built. Therefore there is no KALA code that statically imposes a structure, instead each transaction that forms a part of the tree structure is solely responsible for itself. This implies that each transaction has to perform termination on itself, to remove its dependencies from the ATPMos dependency model, which is performed in lines 5 and 6 of the code below.

Given such a tree structure, built at runtime, there is however one restriction to the structure: a parent may not commit before all its children have ended. Therefore a commit dependency needs to be placed between a parent and each of its children. This requires that each child obtains a reference to its parent, so as to be able to place this dependency, in line 4 of the code below. We achieve this using naming, by first letting each transaction name itself (line 2), so that it can be referred to by its children, and second letting each transaction obtain a reference to its parent by performing a lookup in line 3.

The above is achieved by the following KALA code, that must be defined for all transactional methods (recall that placeholder code is marked like this) :

```
1 packageName.className.methodName(parameterList) {
2    name(self <name expression>);
3    alias(parent <parent expression>);
4    begin {  dep(parent cd self); }
5    commit { terminate(self); }
6    abort {  terminate(self); }
7 }
```

Rollback Handling

When rolling back a transaction which is a part of a tree of nested transactions we need to ensure that if this transaction aborts, all its children have to abort also. This is implemented first by letting each child add itself to a group associated with the parent in line 4 of the code below, and second by letting each transaction terminate its children when aborting, in line 6 of he code below. Having each child add itself to the group associated with the parent, however, also implies that each parent needs to also clean up this group when committing, which is performed in line 5.

```
1 packageName.className.methodName(parameterList) {
2    name(self <name expression>);
3    alias(parent <parent expression>);
4    groupAdd(self <""+ parent + "Children">);
5    commit { groupTerminate(<""+ self + "Children">); }
6    abort {  groupTerminate(<""+ self + "Children">); }
7 }
```

Delegation

Upon commit of a child its work is delegated to the parent, which is performed in line 4 of the KALA code below. Again this requires a reference to the parent, which in turn requires that each transaction register itself.

```
1 packageName.className.methodName(parameterList) {
2    name(self <name expression>);
3    alias(parent <parent expression>);
4    commit {   del(self parent); }
5 }
```

View Management

Thirdly, in Nested Transactions, a child has a view on the intermediate results of its parent, which is achieved by setting the view at begin time in line 4 of the code below.

```
1 packageName.className.methodName(parameterList) {
2    name(self <name expression>);
3    alias(parent <parent expression>);
4    begin {   view(self parent); }
5 }
```

Conclusion

This concludes the general KALA implementation of Nested Transactions, divided in four modules, each module treating one separate concern. We see that in each concern the first two lines of the body are repeated. This is because each module requires a link to the parent transaction, which is manged by these `name` and `alias` statements. We could have chosen to define these two statements once, e.g. in the structure concern, and use these names in the code for the other concerns. We have however not performed this, as this tightly couples the implementation of the other modules to the code of the structure concern. Instead, in the implementation above no such coupling is present. The downside of this is that this does require us to repeat the `name` and `alias` statements. However, the repetition of these statements is not an issue while composing these modules, since name resolving can perform a straightforward merge of these names.

Having given a KALA implementation of Nested Transactions, we now move on to a more intricate ATMS, which is Relatively Consistent Schedules.

9.3.2 Relatively Consistent Schedules

Relatively Consistent Schedules (RCS) is an interesting case for defining KALA code, as on the one hand this ATMS is quite intricate, and on the other hand it shows how an ATMS can be built as the extension of another ATMS.

Indeed, as we have observed in 4.1.2, we can consider RCS as an extension of Sagas, onto which the concern of view management and delegation have been added. We illustrate this here by solely stating the KALA code for both these concerns, and reusing the code for the structure concern and the rollback concern of 9.1 (of which we suppose the concrete method names are replaced with placeholders). We only give the KALA code for the first and last step, and the top-level method here. This is because from the implementation of the first step the code for all subsequent steps, except the last, is easily deduced.

Note that we observed in 4.1.2 that both the concern of view management and delegation are required to form the RCS ATMS, as their interaction is tightly coupled. To emphasize this tight coupling, we could chose to put the code of both these concerns in one KALA definition, but we treat them separate here for clarity.

Delegation

Delegation in an RCS is required to ensure that the work of the RCS is not committed to the database as a step finishes, but only committed as the RCS commits. To achieve this, each step of the RCS will, at commit time delegate its work to the top level, implemented in lines 7 and 12 of the code below. This requires that each step first

obtain a reference to the top level, which is performed in lines 5 and 10. Furthermore, as each step of the RCS requires access to the intermediate results of the previous steps, at the beginning of the step all operations contained within the RCS are delegated to the step, which is performed in lines 6 and 11.

```
1  packageName.className.methodName(parameterList) {
2     name(self <Thread.currentThread()>);
3  }
4  packageName1.className1.methodName1(parameterList1) {
5     alias (RCS <Thread.currentThread()>);
6     begin {    del(RCS self); }
7     commit {   del(self RCS); }
8  }
9  packageNameN.classNameN.methodNameN(parameterListN){
10    alias (RCS <Thread.currentThread()>);
11    begin {    del(RCS self); }
12    commit {   del(self RCS); }
13 }
```

View Management

If we consider the ATMS created by adding the above delegation concern to the Sagas ATMS, we essentially return to one long-lived transaction, which runs contrary to the goals of this ATMS. Therefore we use view management to relax isolation at given moments, to allow other running transactions to access the intermediate data of this RCS.

In RCS, isolation is relaxed in between the different steps of the RCS by allowing a certain number of other transactions to interleave at those points. We can implement this by, after a step completes, setting the view of the RCS to include a group of transactions, as is performed in line 14 below. This implies, however, that the RCS will still have a view that was set by the previous step, which therefore first must be removed, and this is implemented in line 9. Note that for the first step of the RCS no such view will have been defined, but this does not cause the program to fail. This is because the view group to be removed will resolve to the empty group in line 8, the removal of which simply does nothing.

```
1  packageName.className.methodName(parameterList) {
2     name(self <Thread.currentThread()>);
3  }
4  packageName1.className1.methodName1(parameterList1) {
5     alias (RCS <Thread.currentThread()>);
```

```
6     groupAdd (self <group exp2>);
7     begin {
8        groupAlias(preView <""+RCS+"View">);
9        view(RCS -preView); }
10    commit {
11       groupName(<view group exp2> <""+RCS+"View">);
12       groupAlias(newView  <""+RCS+"View">);
13       view(RCS newView); }
14 }
15 packageNameN.classNameN.methodNameN(parameterListN){
16    alias (RCS <Thread.currentThread()>);
17    groupAdd (self <group expN>);
18    begin {
19       groupAlias(prevgroup <""+RCS+"View">);
20       view(RCS -prevgroup); }
21 }
```

Conclusion

Note that this specification, however, is slightly less powerful as the RCS model. In RCS we can specify that a given transaction type TX only interleaves if also another transaction type TY interleaves. Here we have no such control. Although this means we do not realize a complete implementation of RCS, we consider this difference to be negligible. In fact, we find the implementation to still be powerful enough to be useful, and we illustrate this in 11.3.

This concludes the definition of KALA code for the RCS ATMS, which illustrates how a new ATMS can be built by simply adding the implementation of extra concerns to an existing ATMS. We have shown this here by adding an implementation of delegation and view management to the Sagas code of 9.1.

Having given the definition of a number of published ATMS, we now show how we can use KALA to define a new ATMS by creating one: Cooperating Nested Transactions.

9.4 Building a New ATMS: Cooperating Nested Transactions

In this section we show how we can use KALA to define a new ATMS to fit a given application or class of applications. The goal is to achieve an ATMS in which the transactional properties better align with the transactional properties of the (class of) application(s). We show this by creating a new ATMS, which we call Cooperating Nested Transactions which aims to achieve the highest possible performance for

computations that are hierarchically structured.

One of the advantages of using the multi-tiered architecture in a large-scale distributed system is the ability of this architecture to provide a faster response time of the middle layer through load balancing. For load-balancing, we can deploy one component of the system multiple times, on multiple servers so that the workload is distributed over different servers. One component can also call other components to perform sub-computations in parallel, on different servers, which will speed up its response time.

We can use this parallelization on different servers to perform sub-computations of a given algorithm in parallel, but we want the entire computation to be performed as a single transaction to prevent data inconsistency. As the computation is hierarchically structured, we can consider the sub-computations as nested sub-transactions of the main algorithm, and distribute sub-transactions over different servers, to be performed in parallel.

We can consider using Nested Transactions as an ATMS for this application: as sub-computations are sub-transactions they will preserve data consistency, and can access the data of the parent. Also, a failure in the sub-computation will not necessarily imply that the entire computation is lost. This allows graceful recovery of errors in the computation, without needlessly losing work.

Having sub-computations performed in parallel, however, may entail that each of these sub-computations need to be able to access the other computations' intermediate results, as they are supposed to cooperate, in parallel, to achieve the overall goal. This is not possible when using nested transactions and therefore, we have adapted the Nested Transactions ATMS to allow sharing between different sub-transactions, yielding a new ATMS: *Cooperating Nested Transactions* (CNT).

In CNT, all siblings of a parent transaction have access to each other's intermediate results though a view relationship. This entails, as a consequence of also viewing the parent, that they also have access to the intermediate results of the siblings of the parent, up to the root transaction. This, however, has an impact when aborting a child transaction. The siblings which have seen the inconsistent data of this child and have not committed are also considered to be inconsistent and should abort.

Note that the rollback of all siblings only applies to the sibling transactions that run simultaneously, in parallel, with the aborting sub-transaction. Siblings that have committed before the aborter are not aborted, and siblings that start after the abortion need not abort. This limits the lost work in such cases to only include sibling sub-transactions which run at the same time as the aborting sub-transactions, and not all siblings. This is an advantage of using CNT over running the entire computation in one transaction. If we would do this and a sub-computation aborts, automatically all of the work of the entire computation is lost. With CNT, only the work of the sub-computations simultaneously running, in parallel, is lost.

We have implemented CNT in KALA by taking the implementation of Nested Transactions, reusing the structure and delegation concern and modifying the concerns of view management and rollback handling. This illustrates one of the benefits of applying separation of concerns at the level of the ATMS definition, easing modification of an ATMS as only the code for the changing concerns needs to be considered, as we show next.

View Management

In CNT, all children of a given transaction can see each other's intermediate results, as they are supposed to cooperate in parallel to achieve the overall computation. To implement this, in the code below, views are set from this transaction to all siblings, and the reverse, in line 6. This, however, requires each child of a transaction to add itself to the group of children of the parent transaction, performed in line 4, and a reference to be obtained to this group, in line 5.

Also, when a transaction ends, the group of children of this transaction has to be removed by the system, which is performed in lines 7 and 8 of the code below.

```
1 packageName.className.methodName(parameterList) {
2    name(self <name expression>);
3    alias(parent <extends expression>);
4    groupAdd(self <""+parent+"Children">);
5    alias(siblings <""+parent+"Children">);
6    begin {   view(self parent, self siblings, siblings self); }
7    commit {   groupTerminate(<""+ self + "Children">); }
8    abort {     groupTerminate(<""+ self + "Children">); }
9 }
```

Rollback Handling

As a consequence of allowing siblings to see each other's intermediate results, we must consider the consequence of a rollback of such a transaction. As the siblings have seen the intermediate state of the aborting transaction, which is now considered erroneous, these siblings should also abort. We cannot modify transactions which have already committed, but we can abort all currently running siblings of the aborting transaction, which is performed by the `groupTerminate` statement in line 8.

```
1 packageName.className.methodName(parameterList) {
2    name(self <name expression>);
3    alias(parent <parent expression>);
4    groupAdd(self <""+ parent +"Children">);
```

```
5     commit {   groupTerminate(<""+ self + "Children">); }
6     abort {
7        groupTerminate(<""+ self + "Children">);
8        groupTerminate(<""+ parent +"Children">); }
9  }
```

Conclusion

In this section we have created a new ATMS, called Cooperating Nested Transactions, to better fit the transactional requirements of applications with a parallelizable tree-structured algorithm. We performed this by adapting an existing ATMS: Nested Transactions, to also allow viewing between all children of a given transaction.

The separation of the Nested Transactions ATMS into different modules has enabled us to easily perform these modifications, by only needing to change the modules corresponding to the changed concerns. We did not need to consider he code for the structure concern and the delegation concern. This illustrates the advantage of separation of concerns at the level of an ATMS definition.

Having now provided different examples of KALA code, for multiple ATMS, we now take a step back, and consider how a programmer creating a middleware application will employ KALA to use an ATMS in the application being created.

9.5 Programing in KALA: The Problem Statement Revisited

In this chapter we have shown that KALA is well-suited for defining an ATMS by showing the KALA implementations of existing ATMS, and by building a new ATMS through modification of an existing ATMS. The question which remains, however, is how an application programmer will use such a definition to employ an ATMS in the application code, and we address this question here.

We have seen in 9.1 that to use an existing ATMS, in this case Sagas, the programmer has to write a set of KALA declarations, which apply to a number of methods within the application. We have seen that, to define a saga 39 lines of KALA code were needed, but a significant amount of this code is redundant. For example, the code for the begin block of the second and third step are identical, and there are only a few differences between the commit blocks of these steps. Overall, we see that in a saga definition a lot of code is repeated. Now imagine that in the same application multiple sagas need to be defined. This will entail writing this entire KALA code again for each instance of a saga. As each instance of a saga will behave identically with regard to the transactional properties, this means that, in essence, the code of 9.1 will be duplicated. The only exceptions will be the elements referring to the actual methods and compensating methods of the saga. We already hinted at this in 9.3,

where we used placeholders which have to be filled in when applying the ATMS to an application.

The underlying cause for this code duplication is that KALA code is too low-level and does not abstract over the part of the application which is using this ATMS. KALA code defines an ATMS by combining the fundamental ACTA building blocks, and linking these combinations to a part of the application being developed. This results in a definition of the transactional behavior of that part of the application. As a consequence, everywhere this ATMS is used the construction and linking process is repeated. This leads to the code duplication identified above, and it would be better if this is avoided. Furthermore, this process implies that the application programmer is required to know the complex technical implementation of the ATMS, because the sub-concerns of dependencies, views and delegation still have to be treated. This requires intricate work, registering transactions and adding them to groups when required, and performing clean-up operations when possible.

We should avoid the need for an application programmer to write low-level code, as this programmer, simply using an ATMS, should not be exposed to the internals of this ATMS. Doing so, as in KALA, is not necessary as the application programmer will not want to modify the ATMS. More importantly, this is dangerous as it needlessly exposes the internals of the ATMS, breaking encapsulation and more easily allowing errors to be written[1].

The upside of KALA is that it is a generally applicable programming language for declaring the transactional properties of code for an ATMS. Conceptually, we have moved up from low-level assembly-like constructs for demarcation code in 7.3.1, to a higher-level programming language in 9.1, but which is still general-purpose. Therefore KALA can be used as a common base, i.e. a kernel, for other programming languages, which further simplify specifying the transactional properties of code for a given ATMS, trading-off generality.

Such further simplification of the usage of an ATMS is possible by using an extra level of abstraction. We can abstract from the internals of the ATMS and instead let the application programmer reason about the concepts present in that ATMS. To use an ATMS, this programmer then no longer needs to construct an ATMS and link this construction to the application being developed. Instead an existing construct is reused that solely needs to be linked to the application.

For example, considering the saga code in 9.1, the application programmer would no longer need to specify all the dependencies, but rather state the sequence of steps and of compensating steps, as below. All dependencies can be deduced from this sequence. The code below, in which the links to the application being developed is emphasized can be translated to equivalent KALA code, which was given in 9.2.1.

[1] Following the simple rule that the more code is written, the higher chance for errors in that code.

```
1  Saga Cashier.moneyTransfer(BankAccount, BankAccount, int) {
2     step Cashier.transfer(BankAccount, BankAccount, int)
3        compensate transfer(BankAccount, BankAccount, int)
4           params <to, from, amount>;
5     step Cashier.printReceipt(BankAccount, BankAccount, int)
6        compensate printTransferCancel(BankAccount, BankAccount, int)
7           params <from, to, amount, num_receipt>
8           wrap(num_receipt);
9     step Cashier.logTransfer(BankAccount, BankAccount, int);
10 }
```

Reusing the Sagas ATMS within the same application, or for other applications is easy given the ability to reuse the above construct. The emphasized code is simply replaced by the methods and parameters to which the Sagas ATMS applies. This linking process is all that is required to reuse the Sagas ATMS.

In this dissertation this extra abstraction, from general-purpose KALA code to a definition specific for the ATMS, is made through the use of domain-specific languages. We discuss this abstraction in the next chapter.

9.6 Conclusion

As stated in the previous chapter, one of the goals of KALA is to enable modular specifications of ATMS. Instead of having to write tangled aspect code, in KALA it is possible to apply separation of concerns to the process of defining an ATMS, and to write each concern in a separate module.

We have shown in this chapter how KALA allows such modular specification by defining each module in a separate KALA file. We illustrated this by giving KALA code for four ATMS, where each ATMS was modularized in the different concerns of structure, rollback handling, view management and delegation. Of these four ATMS, three were published ATMS: Sagas, Nested Transactions and Relatively Consistent Schedules. Of these three, the last ATMS illustrates one of the benefits of modularization as it is defined by adding two modules to the first ATMS.

We also included one new ATMS: Cooperating Nested Transactions. This ATMS was created to fit a specific class of applications: computations that are hierarchically structured and that achieve higher performance through parallelization. The definition of this ATMS shows how we can easily create a new ATMS by modifying a number of modules from an existing ATMS, in this case Nested Transactions. This again shows a benefit of modularization, i.e. applying separation of concerns when defining an ATMS.

Modularity, however, requires a composition mechanism that combines the different modules into one full specification. In this chapter we also introduced composition of different KALA modules into one full specification, a process which requires no extra programmer input. We have given a sketch of the composition algorithm, which basically amounts to a straightforward merge operation with name resolving, and have given an example result of such a merge.

We ended this chapter by considering how an application programmer, wishing to use an existing ATMS defined in KALA, can apply this ATMS to the application being built. We have seen that while KALA is a good programming language to build such ATMS, the application programmer should not be exposed to this KALA code as it is too low-level. If simply using an ATMS, and not building a new ATMS, the application programmer wishes to abstract from the technical implementation of the ATMS. Instead of the implementation, the concepts present in the ATMS should be exposed to this programmer. The following chapter introduces how we enable this through the use of domain-specific languages.

Chapter 10

Domain-Specific Aspect Languages for ATMS

All language designers are arrogant. Goes with the territory ...
— **Larry Wall**

The goal for this dissertation is to allow an application programmer to use an ATMS for concurrency management in the middle tiers of a multi-tier distributed system. In the previous chapters, we have seen how, using KALA, an ATMS can be implemented which best fits the design of the application being built. However, as we have seen when evaluating KALA in 9.5, the resulting KALA code is extensive, quite complex and leads to a high level of code duplication. This makes the use of an ATMS with KALA a difficult and error-prone task.

Consider the perspective of an average application developer who is not an expert in ATMS, but still wishes to use an ATMS in the application. This developer does not need to be exposed to the internal implementation concepts of the ATMS as specified in KALA, as he will never modify such an implementation. Instead of reasoning about dependencies, views, delegation, et cetera, this application developer will want to reason about the concepts present in an ATMS, e.g. the concept of a compensating transaction. In other words, the application programmer will want to use a higher level of abstraction when using an ATMS. This makes the usage of an ATMS easier and less error-prone.

We achieve a higher level of abstraction for an application programmer in this chapter by using Domain-Specific languages. We first detail the general approach:

going from KALA code to a Domain-Specific Language for each ATMS, before introducing the different languages we created.

10.1 From KALA to Domain-Specific Aspect Languages for ATMS

KALA, which we introduced in the previous two chapters, was created to allow for a separate and modular specification of the usage of an ATMS in a given application. This allows for an ATMS to be implemented which best fits the design of the application. As KALA is a general aspect language for ATMS, it also allows for a new ATMS to be created, if needed. Creating a new ATMS is eased by the ability to specify the different concerns of an ATMS in different KALA modules. This allows a new ATMS to be created by changing one module, or combining different modules to form a new ATMS. However the general-purpose nature of KALA comes at a price, as we have seen in 9.5. KALA code is extensive, quite complex, and can lead to a high amount of code duplication. As a result, use of an ATMS with KALA is difficult and error-prone, and this would better be avoided.

The reason why the KALA code is extensive and complex is because it considers implementation details of the ATMS, instead of sticking to the concepts present in that ATMS. For example, if an ATMS proposes to use compensating transactions for rollback, such as Sagas (discussed in 2.3), the KALA programmer has to specify an autostart, give that autostart a name, place multiple dependencies, and terminate the autostart when no longer needed. KALA needs to work with these underlying concepts since it is a general-purpose ATMS aspect language that needs to cover multiple ATMS, but this requires a trade-off regarding ease of programming. It is clear that it would be easier for the programmer to specify just one ATMS-related concept: the compensating transaction, than to concern himself with four different implementation-related concepts.

We can add an extra layer of abstraction to ease programming an ATMS, sacrificing generality for a higher ease of use. If we drop the requirement to be able to build a new ATMS when needed, we can shield the programmer from the implementation concepts of an ATMS. This is because these implementation concepts are only relevant when building an ATMS and not when simply using an existing ATMS. Programmers that simply use an existing ATMS can work at a higher level of abstraction, using the ATMS as a black box component. In the example above, users of the Sagas ATMS would simply declare a compensating transaction, instead of specifying four different implementation concepts. This raises the abstraction level of the program to the level of the concepts present in the ATMS.

Domain-Specific Languages

In order to be able to program the concern of advanced transaction management in terms of the concepts present in the ATMS being used, significantly easing programming, we propose to add a second layer above KALA, using Domain-Specific Languages. *Domain-specific languages* (DSLs) are little languages, usually declarative, which are specifically designed to provide expressive power for a particular domain [vDKV00]. Because the language constructs of a DSL reflect the concepts of a domain, they will hide non-domain-specific technical issues from the programmer. This allows the DSL creator to fully shield the programmer from these issues, preventing their incorrect usage. Furthermore, hiding these technical issues allows the programs in the DSL to be as concise as possible, containing only the concepts that need to be expressed. This results, amongst others, in higher productivity and maintainability [vDKV00] of the application.

Because a DSL is focussed on addressing a specific problem domain, the code in the DSL is less complex and more concise than in a general-purpose language. We can therefore use DSLs to address the issue we identified with KALA code in 9.5, which is that it is too low-level. By defining a DSL for an ATMS, we can expose the concepts within the ATMS, instead of the underlying implementation concerns. Each DSL is then focused towards one specific ATMS, making the aspect code for that ATMS as concise and natural as possible, as it reflects the concepts of that model.

What we propose is to create a family of DSLs, each language specific to one ATMS, i.e. a model-specific aspect language. The advanced transaction management concern is programmed in such a DSL, and at compile time the DSL code is first translated to KALA code, before it is woven into the base concern, as illustrated in figure 10.1. This solution is mainly inspired from the Oz programming language and its use of a kernel language. The Oz programming language [MMR95] is based on a concurrent constraint kernel language, on top of which a number of languages have been defined for multiple programming paradigms. At compile-time, code in these languages is translated to the kernel language and this resulting code is compiled.

Contrary to using multiple KALA files for one ATMS, as in in the previous chapter, the DSL translators for these DSLs only output one file. Conceptually these translators themselves also implement a merging mechanism, which allows them to perform merging of names in an optimal way. As we are no longer concerned with developing new ATMS, but solely with using existing ATMS, modularity of the generated KALA code is no longer required. Therefore, in this chapter we show KALA code not separated in different modules, but as one, merged, file, containing tangled aspect code.

Note that also in the different languages we define here, the sub-concerns of each ATMS are not separated in different modules of the DSL. None of the DSLs provide for

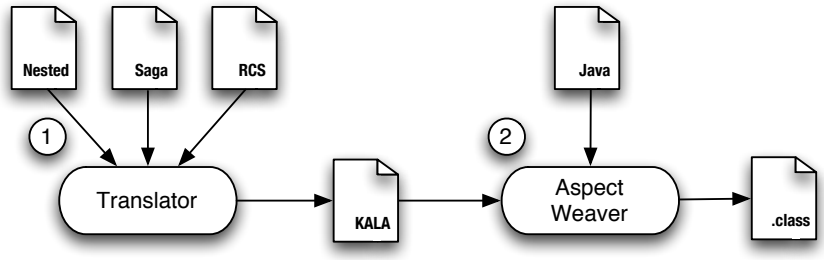

Figure 10.1: At compile time first code in the model-specific languages is translated to equivalent KALA code, which second is woven into the Java code.

a modularity mechanism. As a result, the code in the DSL again contains cross-cutting concerns. However, we explicitly allow this, as we want the DSL code to be as concise as possible, solely reflecting the concepts of the ATMS. Adding modularization would imply that we need to define how different modules are combined. This would require more information to be specified in the DSL programs, making them less concise. Therefore, we have chosen to omit modularization of the cross-cutting concerns within the DSL code.

A Family of Languages

We consider the DSLs we have built for this purpose to be a family of languages because they share and reuse as much syntax and semantics as possible. For each (semantical) concept of an ATMS we identified, we created one syntactical representation of that concept, which is used in all the DSLs that use that concept. For example, the concept of a compensating transaction, which we mentioned above, is always programmed using the `compensate` keyword. This aids the programmer because the skill set acquired by using one DSL can be transferred to other DSLs. It is easier to switch between different languages, as the same concepts are written down the same way, and this also aids in learning new languages, as concepts which are already known need not be re-learned.

We have created five DSLs for our language family, one for classical transactions and one for each transaction model we implemented in KALA in the previous chapter. The first model for which we have created a DSL is the classical transaction model. Second and third we created DSLs for the most well-known ATMS: nested transactions and Sagas. Fourth we chose to create a DSL for relatively consistent schedules (RCS), last is a DSL for the ATMS we created in the previous chapter: Cooperating Nested

Transactions (CNT). Note that we have chosen not to devise names for the different DSLs we create, instead referring to these languages as the transactions DSL, the nested transactions DSL, the Saga DSL, the RCS DSL, and the CNT DSL.

To show that we have a family of DSLs, in which ATMS concepts are reused where possible, we now detail the five different DSLs we created, in the sequence given above. For each DSL we first give a brief recap of the ATMS it addresses, and discuss the different concepts present in that model. We then show how these concepts are addressed in the DSL through an example transactional declaration. Last, we introduce how this example program is translated to the equivalent KALA code, giving a semi-formal definition of this translation step. We have an implementation of each DSL translator, an overview of which we presented in [FC05]. These translators also reuse the partial implementation of each ATMS concept, where possible. We do not discuss the implementation of the DSL translators here, instead we refer to [FC05].

For clarity in this text, we have chosen to work based on an example declaration and not through a more formal specification. The example declarations, however are extensive enough to cover all cases and therefore, combined with the implementation of the DSL transformers, can easily be used to deduce a more formal transformational specification.

We now introduce the DSLs we created, starting with classical transactions.

10.2 Classical Transactions

The first language we introduce here is the DSL for classical transactions. Although this dissertation focusses on advanced transaction models, we include the basic form of transaction management, for completeness.

There is only one concept present in this model, which is the indication that a method is a transaction. Therefore the transactions DSL is simple: programs consist of a list of declarations declaring a method to be transactional. Such a declaration takes the form of a `trans` keyword, followed by the full signature of a method, which is specified as a KALA compile-time method name, i.e. the full class name and method signature, separated by a dot.

For example, consider the method with name `methodName` with parameter list `parameterList`, of the class `className`, contained in the package `packageName`. In order to declare this method transactional, the following line of code is required in the transactions DSL:

```
1 trans packageName.className.methodName(parameterList);
```

Translating such a program to the equivalent KALA code is straightforward, as can be seen below. The only implementation concern which is added to the KALA code

is the termination of the transaction, which is required to remove the representation of this transaction from the dependency model of ATPMos, as we discussed in 8.1.3.

```
1 packageName.className.methodName(parameterList) {
2    commit {terminate(self); }
3    abort {terminate(self); }
4 }
```

Using the transactions DSL here frees us from one implementation concern, which is termination of transactions. Although this is a minor advantage, such freedom of implementation concerns will return and become more significant in the remainder of this chapter, starting with the next model, nested transactions.

10.3 Nested Transactions

Nested transactions is one of the oldest and easily the most well-known ATMS, and therefore it was virtually essential that we construct a DSL for this language, and we discuss it here.

As said in 2.2, Nested transactions allow for hierarchically structured transactions, where a child transaction also has access to the data used by its parent transaction, and this recursively to the root transaction. Furthermore, when a child transaction commits its data, this is not committed to the database, but instead to its parent, which is now responsible for committing these data.

We built a DSL for nested transactions, which supports two ways to structure the transaction hierarchy. First we discuss a specific form of nested transactions, where the transaction hierarchy is equal to the method call hierarchy. Second we treat the general form, without such equality, which corresponds to the KALA definition we have seen in 9.3.1.

Hierarchy equal to caller hierarchy

The first form of structuring the hierarchy of nested transactions which is supported by the nested transactions DSL is where the hierarchy of transactions is equal to the caller hierarchy. In other words, every method which is transactional will be a child transaction of its caller transactional method. Note that the parent transaction need not be the immediate calling method, as this caller can be non-transactional. In this case we must conceptually go back up the call chain to locate the nearest transactional method.

We chose to explicitly support this form of nested transactions in the DSL, as this is a quite popular interpretation of the nested transactions ATMS. This leads as far

as the fact that some databases which initially claim nested transaction support turn out to only provide support for this form [Her]. Possibly this is because this form of nested transactions can easily be emulated using nested savepoints [GR93].

The new concept apparent in this ATMS, over classical transactions, is that this method is a sub-transaction of the (possibly indirectly) calling transaction. This is declared by reusing the naming from the classical transactions DSL, and extending it with the `extends caller` statement. If no transactional method can be found in the call chain, this implies that the transaction is a root of a nested transaction hierarchy. Root transactions can be stated explicitly in the DSL as well, an as they do not extend any other transaction, they are therefore specified by omitting the `extends caller` statement.

Specifying a root transaction is, in other words, identical to specifying a transaction in the classical transaction DSL. This is logical, as no new concept is required, and all the concepts from classical transactions are reused. This shows that indeed we have a family of languages, where concepts are reused over different DSLs and are declared the same way in these DSLs.

Both kinds of transactions can be seen in the example below, which uses analogous names for package, class, method and argument list:

```
1 trans packageNameR.classNameR.methodNameR(parameterListR);
2 trans packageNameC.classNameC.methodNameC(parameterListC)
3    extends caller;
```

The equivalent KALA code, given below, uses the `Thread.CurrentThread()` naming technique which we have already seen in 8.1.1 to identify the calling transactional method. The root transaction will register itself using `Thread.CurrentThread()`, and child transactions look up the parent using `Thread.CurrentThread()` as key. These transactions, such as declared for `methodNameC`, save the parent identifier locally and register themselves as `Thread.CurrentThread()`, to enable their children to obtain a reference to them. At the end of the transaction, the identity of the parent is restored using the saved identifier.

```
1 packageNameR.classNameR.methodNameR(parameterListR) {
2    name(self <Thread.CurrentThread()>);
3    commit {terminate(self); }
4    abort {terminate(self); }
5 }
6
7 packageNameC.classNameC.methodNameC(parameterListC) {
8    alias(parent <Thread.CurrentThread()>);
```

```
9     name(self <Thread.CurrentThread()>);
10    begin {
11       dep(self wd parent, parent cd self);
12       view(self parent);}
13    commit {
14       del(self parent);
15       name(parent <Thread.CurrentThread()>);
16       terminate(self); }
17    abort {
18       name(parent <Thread.CurrentThread()>);
19       terminate(self); }
20 }
```

Because of the semantics of naming, dependencies, views and delegation (defined in 8.1.1 and 8.1.2), when looking up a parent which is non-existent, i.e. this child transaction is in fact a root, no dependency, view and delegation will be performed. In other words, if `methodNameC` does turn out to be a root transaction, the effect of the KALA code will be identical to the code of `methodNameR`.

We see that the use of the nested transactions DSL has freed us from four implementation details: we do not need to concern us with naming, setting the views, performing delegation and termination of transactions. As a result of such concise code, we can conclude that the use of the nested transactions DSL here will indeed ease implementation, as the programmer needs to take less implementation details into account. Also, it is clear that the code in the DSL can not be made more concise, as it only contains the minimum of information required to construct a hierarchy of nested transactions.

Arbitrary Hierarchy

A second case of nested transactions we need to support is where the hierarchy of method callers does not concur with the hierarchy of nested transactions. In other words, a child transaction can declare any named transaction as its parent, and any transaction should be able to give itself a global name.

Conceptually, we consider this as an extension of the case above, where now each transaction declares the name of the parent, and can give itself a name. The former is performed through an `extends` statement, taking as argument a Java expression, which will resolve to the name of the parent. This expression has access to all actual parameters of the method, and, of course, to all globals, and is executed before the body of the method is run. The latter, giving a name to a transaction, is performed through a `name` statement, also taking as argument a Java expression, and which is as

the `extends` statement. Both the `extends` and the `name` statement are optional, as a root transaction does not extend any other transaction, and a child transaction may opt not to give itself a name.

The code below shows an example nested transaction declaration:

```
1  trans packageName.className.methodName(parameterList)
2     extends <extends expression>
3     name <name expression>;
```

The KALA code for this example is similar to the KALA code for previous case of nested transactions. The difference is concentrated solely in naming. Instead of using `Thread.CurrentThread()` to look up the parent, the expression for the parent name is used, and this parent is not set at the end of the method. Also the transaction uses the name given in the `name` statement to name itself. Note that this code is, as should be expected, the same as the merged form of the KALA code for Nested transactions we introduced in 9.3.1.

```
1  packageName.className.methodName(parameterList) {
2     alias(parent <extends expression>);
3     name(self <name expression>);
4     begin {
5        dep(self wd parent, parent cd self);
6        view(self parent);}
7     commit {
8        del(self parent);
9        terminate(self); }
10    abort {
11       terminate(self); }
12 }
```

Absence of `extends` and the `name` statements, which are optional, is treated as follows: If no name for this transaction is given in the DSL through a `name` statement, the KALA code will not include the naming of this transaction. If no `extends` statement is given, no `alias`, `dep`, `view` and `del` statement will be included in the KALA code.

Again, the use of the nested transaction DSL has freed us of the implementation details of setting the views, performing delegation and termination of transactions. The sole concepts which are present in the DSL are those of the model: identifying a method as transactional, naming of the parent transaction, and giving a transaction a name, so it can be referred to. . This implies that the programmer, when using

the DSL, does not need to take these implementation details into account, we can conclude that again the use of the DSL will ease implementation as a consequence. Furthermore, the code which has to be written is as concise as possible, since leaving out any more information entails leaving out a concept of the ATMS.

Conclusion

We have now fully introduced the DSL we created for specifying the use of nested transactions as a separate module. We have seen two usages of the nested transaction DSL, each targeting a different way of building the hierarchy of nested transactions.

We have shown that this DSL only reifies the concepts present in the nested transactions DSL and does not include any of the implementation details required when using KALA. Furthermore, the nested transactions DSL reuses concepts from classical transactions and therefore the classical transactions DSL.

We conclude that the DSL allows the programmer to work with the concepts present in the nested transactions ATMS and frees the application programmer from the implementation details of setting the views, performing delegation and termination of transactions. As a result, the code is as concise as possible and a higher level of abstraction is achieved. This leads to easier implementation of the concern of using nested transactions.

Having talked about the nested transactions DSL, we now proceed with the second ATMS for which we developed a DSL, which is Sagas.

10.4 Sagas

Sagas is, next to Nested Transactions, one of the oldest ATMS and also arguably one of the most referenced ATMS in the community. This has given us reason to define a DSL for this ATMS, while additionally providing for extra flexibility in the ordering of steps. We introduce the Sagas DSL here.

Sagas is tailored towards long-lived transactions, which are split into a sequence of atomic sub-transactions that should either be executed completely or not at all. In order to rollback the Saga compensating actions are used to undo the effects of already committed sub-transactions. Hence, the application programmer has to define a compensating transaction for each sub-transaction, which performs a semantical compensation action. To perform a rollback the TP Monitor aborts the currently running sub-transaction, and subsequently runs all required compensating transactions in reverse order.

We have, in fact, already given an example of the Saga DSL in 9.5, applied to a banking example of which the merged KALA code was given in 9.2.1. The example code we show here is not applied to an example, but is more generic, and we also

show some of the KALA code as it is generated by the DSL transformer. We will not go into details here, but solely focus on the changes with regard to the hand-written KALA code.

Considering the concepts present in the model, we find more concepts present than in the previous ATMS. We identify the concepts of the saga, the steps and their compensating steps. As we introduced in 4.1.1, the saga, steps and compensating steps coincide with methods in the base code. Therefore they can be identified with a method signature as in the transactions and nested transactions DSL. To run compensating steps, the equivalent method has to be invoked, which leads us to two more concepts present: the parameters for invoking the compensating step, and which of these are shared through a wrapper, as introduced in 8.1.4.

The code below shows an example declaration of a saga as a number of steps, each with their compensating step, which takes a list of parameters and an optional declaration of wrapped parameters. We show only two steps in this example, but conceptually allowing for a varying number of steps, as indicated by `[...]` in line 10.

```
1  saga packageName.className.methodName(parameterList) {
2     step packageName1.className1.methodName1(parameterList1)
3        compensate methodName1c(parameterList1c)
4        params <actualslist1>
5        wrap (wraplist1);
6     step packageName2.className2.methodName2(parameterList2)
7        compensate methodName2c(parameterList2c)
8        params <actualslist2>
9        wrap (wraplist2);
10 [...]
11    step packageNameN.classNameN.methodNameN(parameterListN);
12 }
```

The above code contains five different concepts, required for the Sagas ATMS: the saga, the steps, their compensating steps, the parameters for these steps and which of these parameters are wrapped. As this directly corresponds to the concepts of the sagas ATMS, leaving out one of these concepts in the code implies that this code does not implement usage of Sagas, but of another ATMS. As a result, we state that code in the Sagas DSL is as concise as possible.

Regarding the KALA code generated from the above DSL program, we do not include the full code here, as it can be easily inferred from the code given in 9.2.1. We only show the code for one step of the saga, to illustrate that the code for the different steps (except the last step) has become equivalent :

```
1  packageName1.className1.methodName1(parameterList1) {
```

```
2     alias (Saga <Thread.currentThread()>);
3     alias (CompPrev <""+Saga+"Comp">); //may be null tx
4     groupAdd (self <""+Saga+"Step">);
5     autostart (methodName1c(parameterList1c)
6        <actualslist1> (wraplist1) {
7           name(self <""+Saga+"Comp">);
8           groupAdd(self <""+Saga+"Comp">);
9        });
10    begin {
11       alias (Comp <""+Saga+"Comp">);
12       dep(Saga ad self, self wd Saga, Comp bcd self);
13    }
14    commit {
15       alias (Comp <""+Saga+"Comp">);
16       dep(CompPrev wcd Comp, Comp cmd Saga, Comp bad Saga);
17    }
18 }
```

Above, the code for all steps is equivalent, whereas in 9.2.1, this was not the case. In 9.2.1, the second step needed to look up the name of the compensating step of the first step, in order to fix the sequence of compensating steps. Obviously, the first step does not perform this, as it has no previous compensating transaction. However, because of the way naming and dependencies are defined in KALA, we can also add this code to the first step: as the lookup in line three will fail, no dependency will be placed. This, combined with the generic name for the compensating transaction has as a result that step sequence for all but the last step is determined at runtime, instead of statically. In other words, the sequence of steps is not fixed, except for the last step. This flexibility allows steps to be skipped even or performed multiple times, as the core application concern sees fit.

Considering the large size and obvious complexity of the KALA code in 9.2.1, treating many different implementation details: naming, grouping, placing of dependencies, starting secondary transactions, and termination of transactions, it is easy to conclude that the DSL version of the same specification is preferable. Using the Saga DSL, the application programmer is kept unaware of these implementation details, and is solely confronted with the concepts present in the ATMS. We state that the code in the Sagas DSL is as concise as possible and at a higher level of abstraction, which therefore eases implementation.

This brings us to the last ATMS for which we have defined a DSL; the most complex ATMS we discussed in chapter 2, which is relatively consistent schedules.

10.5 Relatively Consistent Schedules

The last ATMS for which we have created a DSL is relatively consistent schedules (RCS). When considering the different ATMS detailed in chapter 2, we see that RCS requires the most information to be specified by the programmer. Also it has a different modus operandi than the previous ATMS treated here, and therefore a different application domain. This makes it a good candidate to further show the power of using a DSL to program a complex ATMS, as we do here. Moreover, the RCS DSL also illustrates the reuse of concepts in the DSL family, as it is specified as an extension to the Sagas DSL.

RCS, as Sagas, was defined to tackle the issues of long-lived transactions, but realizes this differently. In RCS, a transaction is composed of different steps, as in Sagas, but these steps do not commit to the database, contrary to Sagas. Instead, in between each step, the intermediate results of a transaction are made visible to a set of other transactions currently running in the system. How to rollback an RCS is not explicitly stated in the published paper [AFTO89], except for the recommendation to use compensation, as in Sagas.

Considering the concepts present in the model, we note the similarities to Sagas: a top-level transaction which consists of different steps, each with their compensating step, which implies method arguments for the compensating step and the possibility to wrap these arguments. This allows us to reuse all these concepts in the RCS DSL, and we now only need to focus on the concepts new to this model. New here is selectively exposing the intermediate results to a limited group of transactions, and conversely specifying to which group a given transaction belongs. These concepts are present in the RCS DSL through `view` and `group` statements. The `view` statement declares what group of transactions can see the intermediate results after a step has completed, and before the next step begins. The `group` statement determines to which group the step belongs, allowing for this to be different for each step. Both `view` and `group` statements take as argument a Java expression that identifies the group, and has access to the actual parameters of the method.

Example code defining the use of the RCS ATMS is given below, again allowing for a varying number of steps, as indicated by `[...]` in line fourteen.

```
1  rcs packageName.className.methodName(parameterList) {
2     step packageName1.className1.methodName1(parameterList1)
3        compensate methodName1c(parameterList1c)
4        params <actualslist1>
5        wrap (wraplist1)
6        view <view group exp1>
7        group <group exp1>;
```

```
8     step packageName2.className2.methodName2(parameterList2)
9        compensate methodName2c(parameterList2c)
10       params <actualslist2>
11       wrap (wraplist2);
12       view <view group exp2>
13       group <group exp2>;
14 [...]
15    step packageNameN.classNameN.methodNameN(parameterListN)
16       group <group expN>;
17 }
```

As can be expected, the KALA code equivalent for the above RCS DSL code is similar to the code for Sagas. The extra concepts present in the RCS ATMS are translated to extra KALA statements, which are inserted into the declarations for the different steps. These extra statements maintain the views to the intermediate results and ensure the different steps of the RCS work with these results. We do not include the generated KALA code here, as it essentially boils down to the merged form of the RCS code we have already given in 9.3.2.

To illustrate the power of the RCS DSL compared to the equivalent KALA code, we mention that the equivalent KALA program for the example above is eighty-one lines of complex code. Using the DSL frees the programmer from writing such long and complex code where naming and grouping is used, dependencies are set, views are added and removed, and delegation is frequently performed. The DSL allows the programmer to code using the concepts provided by the RCS ATMS, as concise as possible and in a natural form, with only seventeen lines of code. This higher level of abstraction significantly eases implementation.

In the RCS DSL we reused all of the concepts present in the Sagas DSL, as these are also present in the RCS ATMS. This shows that we can achieve a family of DSLs where concepts are reused as much as possible, making the skill set of a programmer transferable over different languages. For example, a programmer knowledgeable of the Saga DSL now only needs to learn the new concepts of the RCS model, as we have demonstrated by only discussing these new concerns in this section.

10.6 Cooperating Nested Transactions

In 9.4, we have created our own ATMS: Cooperating Nested Transactions (CNT). As in the other ATMS we discuss in this dissertation, writing a DSL for this ATMS will ease the use of CNT by an application programmer. Furthermore, the CNT DSL illustrates why the DSLs here are as concise as possible and how a family of DSLs

allows a programmer to transfer his skill set from one ATMS to another. Therefore, we include a description of the CNT DSL here.

Recall that the CNT ATMS, which we defined in 9.4, was created to achieve the highest possible performance for computations that are hierarchically structured. This was implemented by modifying Nested transactions such that all the children of one parent can also view each others' intermediate results. As a consequence, rollback was also modified: if one sibling aborts, all the concurrently running siblings also abort.

The concepts which are present in CNT, in fact, are the same as the concepts of nested transactions. A transaction can be a parent of a number of other transactions and that transaction can be a child of another transaction. As sibling sharing and rollback are defined with respect to these two concepts, no new concept has to be introduced here. This has an important impact on the CNT DSL, as we see next.

As a result of having a family of DSLs, we reuse as much syntax as possible between DSLs. Therefore, as no extra concepts are present in the CNT DSL, the CNT DSL has identical syntax to the Nested Transactions DSL with arbitrary caller hierarchy, as can be seen from the code below:

```
1 trans packageName.className.methodName(parameterList)
2    extends <extends expression>
3    name <name expression>;
```

The above implies that the skill set of an application programmer who knows how to use the Nested Transactions DSL with arbitrary hierarchy, is completely transferable to the CNT DSL. Furthermore, no new syntax needs to be learned, the programmer only needs to be made aware of the peculiarities of this new model.

Note that, although the DSL code has the same syntax, the DSL translator will not produce the same KALA code as for the Nested transactions ATMS, which we have shown above. Instead the translator will produce the code as given in 9.4, albeit in merged form. This illustrates the conciseness of the DSL specifications, as they reflect solely the concepts present in the model, regardless of the actual implementation of this model in ACTA.

To summarize, the CNT DSL illustrates the conciseness of the code using such a DSL by showing how a new model, which does not include any new concepts, implies that the DSL syntax does not change. Combined with the maximal reuse of syntax, this allows the skill set of the application programmer to be fully transferable to the new DSL, once the programmer is made aware of the peculiarities of this model.

10.7 Conclusion

In this chapter we have taken the second step towards easing implementation of the concern of advanced transaction management by making this code as concise as possible. This step consists of using domain-specific languages, one language for each ATMS, to express aspect programs for that particular ATMS. We introduced five different domain-specific aspect languages, for five different transaction models, and have shown how programs in these languages are translated to their equivalent KALA code.

The use of DSLs allows us to isolate the aspect programmer from the implementation details of naming and groups, dependencies, views, delegation, and termination of transaction. Instead, the programmer now focuses solely on the concepts defined in the ATMS being used, as we have shown for all five DSLs. This allowed us to make the aspect programs as concise as possible, while having a natural form. As a result, we have eased implementation of this code, with respect to the equivalent KALA code.

Furthermore, we have built a family of DSLs which reuse as many concepts as possible between them. The same concept in an ATMS will be written down in the same way in the different DSLs. This ensures that the skill set of a programmer in one DSL is reusable as much as possible over the different languages. As a result, we have shown that switching between different languages is eased, as is learning to use a new model.

This completes the introduction of our approach to modularizing ATMS, where we first extracted the concern of ATMS from the base code by defining a general aspect language, and second defined a number of model-specific aspect languages on top of the general aspect language. The first step is beneficial because it effectively modularizes the concern of advanced transaction management. This not only eases implementation, but also enables an ATMS to be tailored to the design of the application. The second step adds the extra benefit of making the aspect code as concise and natural as possible, which aids in implementing and maintaining the concern of advanced transaction management.

Chapter 11

Using Domain-Specific Languages for ATMS

In theory there is no difference between theory and practice. In practice there is.
—**Yogi Berra**

We have now introduced our approach that enables usage of an ATMS the design of which best fits the application being built. This approach first separates out the concern of transaction management, at the level of the code, by using a special-purpose aspect language, called KALA, which we discussed in chapters 8 and 9. As as second step, we have built a number of Domain-Specific Languages (DSLs) in the previous chapter. Each of these is specific for one ATMS and adds an extra level of abstraction on top of KALA. Each language reifies the concepts present in the respective ATMS, making programs written in these languages as concise as possible, while having a natural form.

In this chapter we illustrate how this approach can be applied to applications which are multi-tiered distributed systems. We discuss three example applications, showing the wide varieties of applications to be found in the domain of multi-tier distributed systems. Each application has different requirements for the properties of transaction management, and we select the most appropriate ATMS for each application. For each of these examples we provide an implementation, both for the core concern, in Java, and for the transaction management concern in the appropriate DSL. We show that we can easily use the ATMS of which the design best fits the design of the application, thanks to the good modularization of ATMS demarcation code and the conciseness of the transactional specifications.

We start the illustration of our approach by considering the example application we have used at various points of this dissertation, which is the banking application, and more specifically we focus on the bank transfer operation.

11.1 Using Sagas

In this dissertation, we have used the example of a banking application in our discussion to illustrate the nature of demarcation code for ATMS, show how KALA code is composed, and argue for the need of a DSL to further abstract KALA code to the level of the concepts in the Sagas ATMS. Here we repeat the example to show the applicability of the Sagas ATMS to the bank transfer operation and how this can be achieved by using the Sagas DSL. We first review the bank transfer operation and its need for Sagas, second provide the full implementation of the bank transfer operation using the Sagas DSL, and end with a discussion.

The Case

An example application we have used at various points in this dissertation is a variation on the classical bank transfer transaction. We have positioned this transaction as a part of a multi-tier banking application, where cashiers use this bank transfer operation to perform a transfer for a customer at the bank till. We focussed solely on the bank transfer itself, and ignored other parts of the banking application.

The bank transfer operation is conceptually composed of three actions: first performing the actual bank transfer, second printing a receipt and third updating the logs of the bank. The first action is the transfer itself, which updates both accounts involved in the bank transfer with their new values. The second action points out a receipt which the cashier gives to the customer as a proof of the transfer operation. The third and last step updates the logs of the bank to reflect that a transfer has been performed and a receipt has been given to the customer.

Conceptually, this bank transfer operation is one atomic event, and needs to be made transactional because of the bank account manipulations performed in the first action. Note that this first action is even considered as a classical case for the need of transactions. Implementing the entire bank transfer operation as one transaction, however, is problematic. The cause for this is the printing out of a receipt. To make a printout, even on a fast printer, takes a few seconds, which turns the transfer operation into a long-lived transaction. As we have discussed in 2.3, long-lived transactions have a disastrous impact on application performance as they severely constrict concurrency.

To avoid the bank transfer operation becoming a long-lived transaction, instead of using classical transactions we have used the Sagas ATMS. This ATMS better suits the bank transfer operation as it turns the long-lived transaction into a series

of steps. Each action of the transfer maps to a step in the saga, and for each step a compensating step is possible. The compensating step of the transfer itself is a transfer in the opposite direction. The printout of the receipt can be compensated by printing out a transfer cancellation notice. The last step of the saga has no compensating step, as the Sagas ATMS does not require this.

Implementation

We have already discussed some variations of implementation of the bank transfer example in various parts of this thesis. We have, until now, shown excerpts of the bank transfer code and referred to appendix B for the full implementation. Here we provide a complete implementation using the Sagas DSL. We first detail the Java code for the bank transfer operation, and then give the code for the transaction management concern in the Sagas DSL.

The top level of the Saga is a method which calls the different steps of the saga in sequence, where each step of the saga corresponds to its own method:

```
1  public void moneyTransfer
2     (BankAccount from, BankAccount to, int amount)
3  {
4     this.transfer(from, to, amount);
5
6     this.printReceipt(from, to, amount);
7
8     this.logTransfer(from, to, amount);
9  }
```

This code is quite straightforward, receiving as parameters the EJB's for the respective `BankAccounts` , and the amount of money to be transfered between these accounts. It first invokes the transfer action, second has the receipt printed, and third calls the log action. We now show the code for these three actions in the above sequence, starting with the transfer operation:

```
10 private void transfer
11    (BankAccount from, BankAccount to, int amount)
12 {
13    int from_amount = from.getAmount();
14    int to_amount = to.getAmount();
15    to.setAmount(to_amount + amount);
16    from.setAmount(from_amount - amount);
17 }
```

Again, this code is straightforward, increasing the amount in the destination account and decreasing the amount in the source account.

The second step of the bank transfer is as follows:

```
18 private void printReceipt
19    (BankAccount from, BankAccount to, int amount)
20 {
21    ReceiptCounter counter = ReceiptCounterStore.getCurrent();
22    int num_receipt = counter.getCount();
23    counter.setCount(num_receipt + 1);
24    Printer.getCurrent().
25       printTransferReceipt(from, to, amount, num_receipt);
26 }
```

Here, a unique receipt number is first obtained, to be printed on the receipt. Printing the receipt itself is not implemented in this method, this is instead delegated to a `Printer` object.

This leads us to the last step in the bank transfer operation, which is updating the global log of the bank. Again the code is straightforward, and requires no further comment, as can be seen below:

```
27 private void logTransfer
28    (BankAccount from, BankAccount to, int amount)
29 {
30    OperationsLog log = OperationsLogStore.getCurrent();
31    int num_log = log.getCount();
32    log.setCount(num_log + 1);
33    log.addTransfer(from, to, amount, num_log);
34 }
```

Given the above code for the bank transfer operation, we can let this operation use the Sagas ATMS by providing the following specification in the Sagas DSL:

```
35 Saga Cashier.moneyTransfer(BankAccount, BankAccount, int) {
36    step Cashier.transfer(BankAccount, BankAccount, int)
37       compensate transfer(BankAccount, BankAccount, int)
38          params <to, from, amount>;
39    step Cashier.printReceipt(BankAccount, BankAccount, int)
40       compensate printTransferCancel(BankAccount, BankAccount, int)
41          params <from, to, amount, num_receipt>
42          wrap(num_receipt);
```

```
43     step Cashier.logTransfer(BankAccount, BankAccount, int);
44 }
```

The code above specifies the `moneyTransfer` operation as the top level of a Saga, which will call a number of steps. The first step is the `transfer` method, with as compensating step also the `transfer` method, but with source and destination accounts reversed. The second step is the `printReceipt` method, which is compensated by the `printTransferCancel` method, the code of which is included below, for completeness. In this code the local variable `num_receipt` of the `printReceipt` method is shared between this step and the compensating step (as specified on line 42), so that the compensating step can include this number in the cancellation notice. The last step of the saga is the `logTransfer` method, which does not specify a compensating step as this is not required.

```
45 private void printTransferCancel
46     (BankAccount from, BankAccount to, int amount, int num_receipt)
47 {
48     Printer.getCurrent().
49         printTransferCancel
50             (from_orig, to_orig, amount, num_receipt);
51 }
```

This ends the implementation of the bank transfer operation using the Sagas DSL. We briefly discuss this implementation next.

Discussion

We discuss the above implementation of the bank transfer operation, first focussing on the conciseness of this implementation, and second considering separation of concerns.

In the code above, the reduction in size with regard to the original version of the transfer operation is striking, illustrating the conciseness of using the Sagas DSL. The original version of the transfer operation, using manually inserted demarcation code, shown in appendix B is 267 lines of code. In contrast, above we have 51 lines of code, i.e. less than one fifth the amount of lines of code. This implementation also compares quite favorably to an implementation using KALA instead of the Sagas DSL. The equivalent KALA code for the specification given above is given in 9.2.1, which takes 46 lines of code instead of 10. This clearly shows the conciseness of the DSL specification, which is overwhelmingly better than using manual demarcation code, and also much more concise than the equivalent KALA specification.

Considering separation of concerns, we see that we have achieved a complete separation of concerns at the level of the code of the application. None of the Java methods

above contain code for transaction management. The transaction management concern is completely separated out in the separate transactional specification, in the Sagas DSL. The Java code for the bank transfer operation is written totally oblivious of the transaction management concern. This code simply needs to follow the design which does take transaction management using Sagas into account. This not only eases implementation of the bank transfer example, but also significantly enhances maintainability and evolvability. For example, consider what has to be done if the bank transfer operation would have to be extended with an extra step, say updating a local log. The programmer first implements this step, and the counterstep, not considering the transactional nature of these operations. Second, the top-level `moneyTransfer` method is modified to call the extra step. Third and last, to include this step in the saga, the corresponding `step` declaration is added to the program in the Sagas DSL. In general, due to the full separation of concerns, any method called from the top-level `moneyTransfer` method can be included in the saga, as an extra step, simply by adding the corresponding `step` declaration to the program in the Sagas DSL.

The combination of a full separation of concerns and concise DSL code for using this ATMS highly ease implementation of this application, making it almost seem trivial to use such an ATMS for a given application. This nicely illustrates the power and applicability of our solution, which enables an application programmer to easily use an ATMS the design of which is best suited to the application being built.

Conclusion

We have now given a full implementation of the bank transfer operation, an example which we used at various points in this dissertation. This transfer operation consists out of three separate steps: first the transfer itself, second printing out a receipt for the customer, and third updating the logs of the bank.

We have argued that it is better to have this transfer operation as a Saga than as a classical transaction, as the latter would be a long-lived transaction due to the time needed to print the receipt. We have identified the steps of the saga and their counter-steps and proceeded with giving the full implementation of the transfer operation.

The implementation of the transfer operation, in comparison to the implementation with manual demarcation code in appendix B, and to the implementation using KALA in 9.2.1, clearly shows the advantages of our solution. We now have concise code with a full separation of concerns, in contrast to verbose and tangled code. In fact the code for the bank transfer operation is completely oblivious of the transactional nature of the transfer. This illustrates how easily our solution enables the use of an ATMS the design of which best fits the application being built.

As a next example of the use of an ATMS DSL, we consider how nested transactions can be used for a web-based shopping aid application.

11.2 Using Nested Transactions

In nested transactions [Mos81], a tree is built of hierarchically structured transactions. In this tree, a child transaction has access also to the data used by its parent transaction. When a child transaction commits its data, this is committed to the parent, which is now responsible for committing this data. Aborting a child does not imply that its parents are aborted, which gives us a more fine-grained mechanism for handling rollbacks.

We conceived an example component of a multi-tier distributed system: a shopping aid for a web-based classic car parts store which is implemented as a set of rules. We first outline this example case, before introducing the implementation which shows how the nested transactions DSL is used and discussing this implementation.

The Case

A specialty market for which many on-line web shops can be found is the market of spare parts for old-timer cars. These shops sell a very extensive variety of parts, some shops virtually allowing the enthusiast to build a car from scratch, just by buying all the parts. These on-line web shops are typical cases of multi-tier distributed systems, and therefore are applicable to our research. The example application we conceived to show the usage of the nested transactions DSL is an automated shopping aid for these kinds of shops, and we introduce it next.

The shopping aid is a component of the web shop the purpose of which is to help the shopper obtain the parts required for a given repair. Based on a description of the task to be performed, the system will fill a shopping basket with spare parts required for this task. However, not all parts are created equal: some parts will be specialty parts which can only be obtained from this shop, while other parts are considered common, and can be bought at any auto parts store. For example: if a ball bearing needs to be replaced, a specialty part is the new ball bearing, while if the brakes need to be bled, new brake fluid is required[1], but this can easily be obtained from any auto parts store and is therefore a common part. A second distinction of parts is if they are from the original manufacturer or if they are replica parts. Classic car purists require original parts, and will only settle for replicas if no original part is available.

The shopping aid needs to take into account the distinction between specialty parts and common parts, and the difference between original parts and replicas. If a common part can not be placed in the shopping basket because the store has run out, a warning will be issued. If a specialty part from the original manufacturer is not available, a replica part should be substituted. If no specialty part at all can be obtained, however, the shopping aid will not place any part in the basket at all, since

[1] Brake fluid may never be reused [HS76].

the repair can not be performed. Note that before checkout, the customer can of course elect to add or remove items from the basket, but managing the contents of the basket is not relevant for this discourse, so we will not discuss this further.

The shopping aid is designed as a set of rules, where each rule corresponds to one task, e.g. bleeding the brakes, and is responsible for giving a list of items to be put in the basket. Rules may call other rules, as tasks can be split into sub-tasks, some of which may be quite common. For example, bleeding the brakes is a sub-task which is required for the majority of work on the brakes, such as when changing the disks or the calipers. Therefore the brake bleeding rule will be called from the brake disk changing rule and from the brake caliper changing rule. This results, when the shopping aid is executed, in a tree structure of rules that are run.

Each of these rules is implemented as a method, and will have to run within a transaction. This is because they may remove items from the parts inventory, to be placed in the shopping basket. If, within a rule, a part can not be obtained for whatever reason the transaction will abort, but this does not necessarily mean that the shopping aid as a whole should abort. The shopping aid should only abort if a specialty part is unavailable. If a common part can not be placed in the shopping basket, the shopping aid must continue. In other words, we need a fine-grained control of transaction aborts, as only in some cases aborting a rule implies aborting the entire transaction.

The shopping aid is designed as a set of rules, which at run-time form a tree structure. Furthermore, the computation needs to be enclosed in a transaction. Finally we require fine-grained control of rollbacks. This leads us to chose the nested transactions ATMS for this component. Each rule will run all the sub-rules it calls as separate sub-transactions, and also obtain parts from the inventory in a separate sub-transaction. For each of these sub-transactions, the rule can then decide what to to when a sub-transaction fails. The rule may itself abort or make amends for this failure. For example, the `bleedBrakes` rule, will call the `getPart` rule to obtain brake fluid from the inventory. If this fails, the `bleedBrakes` rule will not fail but return a dummy part instead of the brake fluid part. This part will simply display an appropriate warning when listed in the customers' shopping basket. Similarly, each rule will first try to obtain original parts, and if this fails look for replica parts, if available.

Implementation

Given the description of the shopping aid component above, we now show the implementation of this component, with regard to the rules and their transactional behavior. We do not detail all the possible rules of the shopping aid here, instead we focus on one specific task, which is the replacing of a ball bearing of a rear wheel, as given by the procedure laid out in [HS76]. We show next the code for all the rules that

are triggered for this procedure, all members of the `SARuleSet` class, followed by the transactional specification of this code in the nested transactions DSL. We start with the first rule which is triggered, the code for which is below:

```
1  public PartCollection replaceBallBearingAft(){
2     PartCollection pc = new PartCollection();
3
4     int bbnumber = partInventory.getPartNumber("BallBearingAft");
5     pc.addPart(getPart(bbnumber, 1));
6
7     pc.addParts(removeRearAxleBlockingComponents());
8     pc.addParts(dissassembleRearAxleHalf());
9
10    return pc;
11 }
```

The above code, as all rules, collects all the parts it requires in a `PartCollection`, which is returned to the caller. To get a specific part, its part number is obtained from the inventory using the `getPartNumber` method of the inventory, which takes a descriptive name of the part, and returns the number. We do not discuss this method in detail, as it is of no particular significance here. With the resulting part number, parts can be taken out of the inventory, using the `getPart` method, which we discuss in more detail later.

To replace a ball bearing aft, the rear axle must be freed from all blocking components, such as the brake assembly, after which it can be disassembled in place. These sub-tasks are implemented in the rules `removeRearAxleBlockingComponents` and `dissassembleRearAxleHalf`, and we will discuss these after detailing the `getPart` method, which is next:

```
12 private Part getPart(int partnumber, int req_amount) throws TxException {
13    PartStore store = partInventory.getParts(partnumber);
14    int stored_amount = store.getAmount();
15    if (stored_amount < req_amount)
16       throw new TxException("Not enough stock");
17    store.setAmount(stored_amount - req_amount);
18
19    return new Part(store.getDescription(),partnumber, req_amount);
20 }
```

The `getPart` method obtains a `PartStore` EJB, which represents the section of the inventory which contains the parts with the given part number. The `PartStore`

records the number of these parts in inventory, a descriptive text for these parts, amongst others. The `getPart` method verifies if there are sufficient parts in inventory, and if so removes these from the inventory, to be returned to the caller. If there aren't enough parts, the `getPart` method terminates immediately by throwing an exception.

Having discussed `getPart`, we now continue with the rule `removeRearAxleBlockingComponents`, the code for which is below:

```
21 public PartCollection removeRearAxleBlockingComponents(){
22    return bleedBrakes();
23 }
```

Removing the brake assembly and other blocking components does not directly require any new parts to be added to the shopping basket. The procedure however does require the brakes to be bled, which is represented by the `bleedBrakes` rule, which we discuss next:

```
24 public PartCollection bleedBrakes() {
25    PartCollection pc = new PartCollection();
26
27    int bfnumber = partInventory.getPartNumber("BrakeFluid");
28    try {
29       pc.addPart(getPart(bfnumber,1));
30    }
31    catch (TxException ex) {
32       pc.addPart(new WarningPart("Obtain brake fluid"));
33    }
34
35    return pc;
36 }
```

As we mentioned above, the `bleedBrakes` rule calls the `getPart` rule to obtain brake fluid from the inventory. If this fails, the `bleedBrakes` rule will not fail but return, instead of the brake fluid part, a dummy `WarningPart`, which will display an appropriate warning when listed in the shopping basket. This leaves us to discuss the `dissassembleRearAxleHalf` rule, which is as follows:

```
37 public PartCollection dissassembleRearAxleHalf() {
38    PartCollection pc = new PartCollection();
39
40    int blrnumber = partInventory.getPartNumber("RearAxleBLRing");
41    pc.addPart(getPart(blrnumber, 1));
```

```
42
43      return pc;
44  }
```

Disassembling the rear axle implies removing the rear axle ball bearing locating ring, which may not be reused [HS76], and therefore a new ring will be added to the shopping basket. This is the last of the rules which are invoked for replacing a ball bearing aft. Note that we did not include any rule here that looks for replica parts, it suffices to say that such rules will be implemented similarly to `bleedBrakes`, looking for a replica part if getting the original part fails, instead of creating a dummy `WarningPart`.

Having seen all the code for the core concern of the application, we can proceed with declaring the transactional properties of these methods in the nested transaction DSL, which is as follows:

```
45  trans SARuleSet.replaceBallBearingAft() extends caller;
46  trans SARuleSet.getPart() extends caller;
47  trans SARuleSet.bleedBrakes() extends caller;
48  trans SARuleSet.dissassembleRearAxleHalf() extends caller;
```

The above code is all which is required to have the declared methods as nested transactions, each a child transaction of the transactional method which called it. Note that `removeRearAxleBlockingComponents` is not declared a nested transaction, as no data access is performed in that method, the `bleedBrakes` method will therefore be a nested transaction of `replaceBallBearingAft`.

This ends our treatment of the code for the shopping and. We have first introduced the Java code for different rules of the shopping aid, and concluded by giving the DSL code which lets this application use the nested transactions ATMS.

Discussion

Given the above code for the shopping aid application, we now discuss the conciseness of the code and the degree of separation of concerns within this code.

We can safely state that the above code for transaction management is short, concise and simple, as it treats the transactional aspect in terms of the concepts exposed by the model: transactions and parent transactions. This makes it easy to write, very readable and also very easy to modify. If, for example, `removeRearAxleBlockingComponents` is modified, to now also include the application of axle grease, this method can straightforwardly be made transactional by adding an extra line to the transactional properties.

Some enhancements could be made to the nested transaction DSL, however. Currently we must explicitly declare all methods to be made transactional, and if the rule set becomes large, this will result in a long list of declarations. It would be better to be able to use some form of wildcarding, as is possible in AspectJ [Asp05], and our previous work [Fab04b], for example as below:

```
1 trans SARuleSet.*() extends caller;
2 trans SARuleSet.removeRearAxleBlockingComponents() exclude;
```

This will automatically apply to all the methods in the `SARuleSet`, and exclude `removeRearAxleBlockingComponents` from being made transactional. We have not yet included this feature in the nested transactions DSL, as it has only limited impact the core of this work: modularizing advanced transaction management using a concise specification, which is already achieved with the current version of this language.

In `getPart` and `bleedBrakes` the line between the transaction management aspect and the core concern of the application has become somewhat blurred, because the core concern uses transaction failures to indicate a failure: obtaining a part from the inventory. We consider, however, that this transaction management code belongs to the core concern, as the triggering of the rollback in `getPart` is the result of the application logic, as is the use of a `WarningPart` in `bleedBrakes`. Therefore, we state that this code should remain in the method, and not be placed in the aspect. This provides adequate ground for not adding rollback triggering and handling code in the nested transaction DSL.

What we have here is that the originally non-functional concern of rollback handling has been turned into a functional concern. Rollbacks of transactions are actively used in the base functionality of the application. Therefore, due to an explicit design choice of the application, we do not achieve total obliviousness of the transaction management concern, this in contrast to the bank transfer operation we discussed previously.

Further considering separation of concerns, we can conclude that using the nested transactions DSL does indeed separate out transaction demarcation code in a separate module to a very high degree. As we indicated above, it is not preferable to completely separate out all transaction related code, as in nested transactions part of the core concern might be tightly coupled with the transaction management concern.

Conclusion

In this section we have shown how to apply the nested transactions DSL to the building of a component of a web-shop for classic car parts. The component we investigated was a shopping assistant, a rule-based system that fills a shopping basket with spare

parts based on a given repair to be performed. In this component, each rule triggers sub-rules as sub-transactions, and may make amends if a sub-transaction fails.

We have seen the code for different rules, in Java, and the code for making these rules transactional in the nested transactions DSL. We can conclude that we have separated out transaction management as much as possible, as the only transaction demarcation code to be found in the code for the core concern is tightly coupled with the core concern. Furthermore, the code in the nested transactions DSL is indeed clear and concise, exposing the transactional concepts present in the model. Also, transactional modifications to the code are easy to make, as making a rule a nested transaction or not is done by adding or removing one straightforward line of code.

The next ATMS for which we show DSL usage is for a less typical multi-tier distributed system: calculating the bill and management of inventory in a supermarket. The ATMS which will be used here is relatively consistent schedules.

11.3 Using Relatively Consistent Schedules

The second ATMS we use to show how DSLs for ATMS are used is the relatively consistent schedules ATMS, the most sophisticated ATMS we have discussed. We introduced this ATMS in 2.4 as an ATMS which uses semantic information to allow more concurrency when using long-lived transactions. This is achieved by splitting up the transaction into different steps, and defining how each step may be interleaved with steps of other transactions. Through interleaving of such steps different transactions can share data without affecting consistency.

In order to show how the RCS DSL is used to employ the RCS ATMS on an application, we have created a second application: a supermarket checkout system. This system is responsible for calculating the bills for the customers as products are scanned at the checkout counters, and for managing the inventory of the shop. We first give an overview of the system, before detailing its implementation, which is then discussed.

The Case

Consider the checkout process in a common supermarket: the customer places his goods on the counter, and the checkout clerk uses a bar code scanner to scan all products, adding the cost to the bill, and decreasing stock count for the scanned product. The computer system calculating the bills and managing stocks can also be implemented as a multi-tier distributed system. The cash registers are thin clients, since they are embedded systems with few computational resources and are low on memory. To perform stock management, the cash registers communicate with a central server, which is the middle tier, and that server uses a database as the lowest tier.

Conceptually, the checkout action is one transaction, which starts when the first product is scanned and ends when the customer has paid the bill. But, as we all have experienced while waiting in line at the checkout counter, this transaction can be long-lived. The consequences of this are grave: when, for example, a bottle of Duvel beer is scanned, the transaction keeps an exclusive lock on the count of Duvel bottles in stock, making it impossible for other checkout counters to decrease the stock count. The other counters now have to wait until the bill is paid before their checkout process can continue, which is clearly unacceptable.

Furthermore, this is but a simple case: people can be inattentive, and for example, before paying the bill, realize that they do not need as many bottles of beer, and return one. These purchases have to be 'un-scan'ed, and stock count has to be increased.

Note that we could, of course, provide an implementation of the supermarket application using the classical transaction model. However, this solution would tend to be an application-specific solution which is not as conceptually clean, and which therefore suffers from drawbacks if we consider maintainability of the application. For this small example, this is not a big issue, but its importance increases significantly when the application becomes larger.

Let us consider the checkout transaction in an ATMS context. We can easily see that we can split the long-term transaction in different steps: each part is a 'scan' or an 'un-scan', and the last part is a 'payment' action. Using this analysis, we see that relatively consistent schedules is an applicable ATMS. The compensating step of a 'scan' is an 'un-scan', and vice-versa. There is only one transaction type: the 'Checkout' type, and both the 'scan' and 'un-scan' steps belong to this type. After every step, all steps of the Checkout type may interleave, whereas other transactions, which are of other types, may not.

Using the relatively consistent schedules ATMS here keeps the implementation conceptually clean: the checkout remains one transaction, as we will see next.

Implementation

The checkout process is implemented in the middle tier, i.e. the central server of the supermarket, as the cash registers do not have enough memory to keep a list of products and their prices. Each checkout process is modeled as an instance of the `Checkout` class, of which the `doCheckout` method contains the code for performing the checkout, which can be seen below:

```
1 void doCheckout(){
2    int qty;
3    while((qty = getInput()) != 0){
4       if(qty > 0)
```

```
5           scan(getInput(), qty);
6        else
7           unScan(getInput(), -qty);
8     }
9
10    showTotal();
11 }
```

The checkout process is a loop, repeatedly scanning products until all products are scanned. The scanning process starts by obtaining the quantity of products being scanned, which may be zero to indicate the end of the scanning process. Multiple items of the same product can be scanned at once, as some products may be bundled in groups, such as a six-pack of beer, or the cashier may manually input a negative amount to indicate a return of goods. Based on the sign of the quantity, either the `scan` or `unScan` method is called to either add to or remove the products from the bill. Both methods are given the product identifier, obtained from the input device, and the number of products scanned. At the end of the scanning process, the cash register is instructed to show the amount to be paid.

We do not detail the implementation of `getInput` and `showTotal` here, as the implementation of these methods is not relevant for this discussion. We focus, instead on the `scan` and `unScan` methods, and start with the code for the `scan` method, below:

```
12 void scan(int Pid, int Qty){
13    ProductBean product = getProduct(Pid);
14    StockBean stock = getStock(Pid);
15
16    stock.setQty(stock.getQty() - Qty);
17    if(stock.nearEmpty()) addOrder(stock);
18
19    addPrice(product.getName(), product.getPrice());
20 }
```

Based on the product identifier and the quantity of products, the `scan` method adds the product name and price to the bill, and removes this amount of products from the stock of the shop. `scan` first obtains a `ProductBean` entity bean, which represents the product, followed by a `StockBean` entity bean, which represents the part of the shop's stock that contains that product. The stock is decreased, and if it is nearly empty, an order is placed to the supplier. The method ends by having the cash register print the name and the price of the product on the bill, using the `addPrice` method, which also updates a running total within the cash register. We

do not discuss the `nearEmpty`, `addOrder` and `addPrice` methods here, as they are not transactional.

The inverse of scanning a product, is returning a product at checkout time, and this is processed through the `unScan` method, the code for which is below:

```
21 void unScan(int Pid, int Qty){
22    ProductBean product = getProduct(Pid);
23    StockBean stock = getStock(Pid);
24
25    stock.setQty(stock.getQty() + Qty);
26    notifyPickup(product);
27
28    removePrice(product.getName(), product.getPrice());
29 }
```

In this method the items are put back into stock, a notification is sent out to pick up the goods at the cashier's desk, to be restocked, the items are subtracted from the bill, and the running total is decreased. As the code for `unStock` is quite straightforward, we do not discuss it further, and neither do we discuss the `notifyPickup` and `removePrice` methods, as they are not relevant here.

Note that `scan` and `unScan` are not exactly each other's inverse: if an order has been placed for some products as the result of a `scan`, the undo of this scan, i.e. `unScan` does not anull this order. This is because it can be expected that this product will run out soon, and having the stock management application placing an order slightly early is not a problem. Similarly, a notification to restock goods as a result of an `unScan` is not rescinded by the `scan` method. This may lead to spurious pick-up notifications, but again this is allowed.

We have now shown the code for the core concern of the application, which is calculating the bill for customers and updating the stock inventory as items are checked out. What remains is to specify the concern of transaction management, stating that the `doCheckout` method is a transaction, using the relatively consistent schedules ATMS. This is performed by the following program, written in the RCS DSL:

```
30 rcs Checkout.doCheckout() {
31    step Checkout.scan(int, int)
32       compensate unScan(int, int)
33       params <Pid, Qty>
34       view <"Checkout">
35       group <"Checkout">;
36    step Checkout.unScan(int, int)
```

```
37        compensate scan(int, int)
38        params <Pid, Qty>
39        view <"Checkout">
40        group <"Checkout">;
41     step Checkout.showTotal()
42        group <"Checkout">;
43  }
```

In this program, we declare that the `doCheckout` method is a transaction, with three steps: the `scan` and `unScan` methods, and as a last step the `showTotal` method. All `doCheckout` transactions belong to the `Checkout` group, and between scans every `doCheckout` transaction allows all other `doCheckout` transactions to interleave. This allows all `doCheckout` long-lived transactions to execute in parallel, while maintaining consistency, as in the `scan`, `unScan` and `showTotal` methods, no data is shared.

Note that because the KALA code generated by the RCS DSL builds the sequence of steps dynamically, as we have seen in 10.5, there is no constraint on the number of sequence of steps for `scan` and `unScan`. The sequence of these steps is of no importance, and each step may be repeated any number of times, i.e. any number of products may be scanned, and unscanned, in any sequence. The only fixed step is the end of the checkout process, showing the total amount due.

We have now shown the implementation of the supermarket checkout system, first introducing the Java code responsible for stock management, and second detailing the code in the RCS DSL, which describes the ATMS concern for this application.

Discussion

We now discuss the code given above, focusing on the separation of concerns achieved, and considering the conciseness and complexity of the DSL code.

Using the RCS DSL we have completely separated transaction demarcation code from the core concern of the application. The entire transaction concern is contained within the RCS DSL program, and the Java code contains no transaction code whatsoever. In other words, again the Java code is totally oblivious to the use of an ATMS, simply following a design which fits the RCS ATMS. Having used the relatively consistent schedules ATMS, the implementation of the checkout process is kept conceptually clean: the checkout is one transaction, and because we have used the RCS DSL, this did not come at a significant price considering coding work. Furthermore, because we define the `doCheckout` method as a transaction, we get the ability to undo the checkout process simply by roll-backing the transaction. This will trigger the compensating methods, effectively annulling the checkout.

The code in the RCS DSL is more complex than the code we have shown for the

nested transactions example above, but this is unavoidable as the relatively consistent schedules ATMS requires more semantic information to be stated by the programmer. Nevertheless the code is concise as it directly expresses the concepts used by the ATMS (the transaction, the steps, their countersteps, visibility between steps, the type of each step) and the relationships between these concepts. As a result, this code is still quite straightforward to write, understand and maintain. For example, consider that a new type of step is introduced, say processing of coupons. These are also scanned, have an impact on the bill, and are tallied in the stock. First the code for scanning the coupon is written, together with the counterstep, which represents handing the coupon back to the customer. In order to include this in the `doCheckout` transaction, this method is simply added as a new step in the RCS DSL.

Conclusions

In this section we have shown the use of the RCS DSL to render a checkout process in a supermarket transactional, using the relatively consistent schedules ATMS. Every checkout is designed as a transaction, containing two possible steps: adding items to the bill which removes them from the stock count, or the inverse.

We have seen the code for the core concern of the checkout, which is calculating the bill and updating stock count, and have shown the code for the transactional aspect of this process, in the RCS DSL. Although the code in the RCS DSL is more complex than the code in the nested transactions DSL, we can conclude that this code is still concise as it directly reifies the concepts of the ATMS used, and their relationships. Furthermore, the Java code for the checkout process is completely oblivious of the transactional nature of this process. As a result, this code is easily written and maintained.

11.4 Conclusion

In this chapter we have shown three examples of how our approach enables application-specific advanced transaction management. The three examples applications we have shown are a bank transfer operation, a shopping assistant, and a checkout process of a supermarket. We now summarize each of these examples and give an overall conclusion.

The bank transfer operation is the example we have used in different parts of this dissertation. The bank transfer operation is part of a larger banking application, which is used by cashiers to serve customers in a bank office. For this application we solely focused on a money transfer operation between two accounts. This operation consists of three distinct actions: first transfering the money between the accounts, second, printing out a receipt to be handed to the customer, and third updating the

logs of the bank. Because printing the receipt turns the transaction into a long-lived transaction, and the transfer operation consists of three different actions, the Sagas ATMS is best suited for this operation. Each action corresponds to a step of the saga, and counter-steps are both available for the transfer operation and for the printing of the receipt. We have given a full implementation of the bank transfer operation, as Java code, and used the Sagas DSL to let this operation be a Saga.

The second example is a shopping assistant which is part of a web-based shop for spare parts of old-timer cars. Customers use the assistant to fill their shopping basket with spare parts, based on a general description of the repair work they wish to perform on their cars. Typically such work is hierarchically structured in different phases, and a given phase may be used in different repair activities. The shopping assistant is therefore structured as a rule-based system, each rule corresponding to a phase of the repair. Each rule is responsible for filling the shopping basket with the required items for that phase, and may call other rules to handle a sub-phase of the repair activity. We found the Nested Transactions ATMS to be suited for this application. Each rule of the application corresponds to a transaction, and will be a child transaction of the calling rule. We have given some of the rules of the application, written as Java code. Finally, we have shown how we can straightforwardly use the nested transactions ATMS for these rules by writing a small amount of code in the nested transactions DSL.

The last example we have treated in this chapter is the checkout process of a supermarket. Each cash register scans products so as to compute the bill. As the products are scanned, the shop inventory is updated and orders are placed to the distributors if stock is low. Conceptually, the entire checkout process, from the first product being scanned until the bill is paid, is one single transaction. Implementing this as one single transaction would be disastrous, however, as the checkout is a long-lived transaction. As a result it would be hard, if not impossible, to have many checkout counters to operate in parallel. A counter would spend most of its time waiting for other counters to finish their transaction before it can perform stock manipulations, and the probabilities for deadlocks would be high. The RCS ATMS is more suitable than classical transactions for this application. We can have each product scan as a step in the RCS, which can be undone by an un-scan. Both these steps are member of the same transaction type, and this type may interleave after each step. whereas other types may not. This allows the cashiers to operate in parallel, as a cashier now only needs to wait until another cashier has finished scanning a product, at the most. We have shown the implementation of this checkout process. We have given the Java code for the different scan and un-scan methods, as for the overall loop that drives the checkout process. We also provided the code in the RCS DSL which lets the application employ the RCS DSL.

In the three examples above, we have seen that the code concerning transaction

management is very concise while remaining readable. In the bank transfer example, we have seen that using the Sagas DSL the code size is one fifth the size of when using manual transaction demarcation code. Furthermore, in the shopping aid and checkout example, we have argued that the code can not be made more concise as it solely contains the concepts required for the ATMS being used.

Regarding separation of concerns, in the second example the transaction management concern was not fully separated out of the main application. This is due to the fact that the application logic uses the fine-grained rollback structure of nested transactions leading to a strong coupling between these two concerns. The code for the transaction management concern is, however, separated out as much as possible, which does aid implementation and maintenance of the application. Regarding the two other examples, however, we achieve a full separation of concerns. Neither the bank transfer nor the checkout example core application logic include any transaction management code at all. In both examples, all transaction management is programmed separately, in the DSL. Furthermore, the DSL code does not include any application logic whatsoever. This hugely benefits implementation and maintenance, allowing the transaction management concern to be implemented separately from the core application logic.

We conclude that we have successfully illustrated that our approach enables the use of an ATMS which best fits the design of the application being built. This is thanks to the good modularization of ATMS demarcation code. This modularization not only separates out the transaction management concern from the core application, but also lets this code be as concise as possible. As a result, using an ATMS in an application seems almost trivial, solely requiring a brief specification in the appropriate DSL.

Chapter 12

Conclusions and Further Research

"We apologize for the inconvenience"
— **Douglas Adams, God's Final Message to His Creation in "So Long, and Thanks for all the Fish"**

In this dissertation, we presented the first approach that enables the use of advanced transaction management in a multi-tier distributed system. We achieved this by applying the software engineering techniques of Aspect-Oriented Programming and Domain-Specific Languages. Combining these two techniques allowed us to create an approach that is widely applicable while ensuring that the code for using such an advanced transaction model is as concise as possible.

In this chapter we first provide a short summary of the context of the dissertation before elaborating on our contributions.

12.1 Research Context

This dissertation is situated in the context of multi-tier distributed systems. Multi-tier is the preferred architecture of large-scaled distributed systems which contain a large amount of clients, and we discussed this architecture in chapter 5. Typical examples of such systems are web banking applications and Internet shops. In general, a large part of the applications accessible through a web browser are multi-tier systems. Building such systems is aided by the large availability and the presence of a number of standards for middleware. Middleware is server software which facilitates the creation of such systems. This is performed by implementing support for common features of

these systems, such as network communications and database access. In multi-tier distributed systems the de facto standard for concurrency management is *transaction management*, and it is supported as a standard service by a built-in TP Monitor in the majority, if not all, of middleware. This has as a consequence that the concern of transaction management is no longer pertinent to an elite cadre of programmers. Instead all multi-tier application programmers need to take transaction management into account.

12.1.1 Issues with Transaction Management

The fact that transaction management is a concern for a large number of programmers working on a large variety of distributed system raises some issues. The core problem here is that transaction management was originally designed for one *limited form of data access*, which is using simple, isolated units of work, taking a short time to complete, that lock few data items. If this is ignored and applications use long units of work, or units of work with a complex structure, or sharing needs to be performed between different units of work, the transaction concept starts to break down. As a result, application performance takes a significant hit, there is no mapping between the structure of the unit of work and the corresponding transaction, and data sharing can not be cleanly implemented. Multi-tier distributed systems are already faced with these shortcomings, a testimonial to which is the fact that workarounds are present in the community to handle this issue. These workarounds address the mismatch between the concurrency management properties as offered by transaction management and the concurrency management properties required by a particular application.

Instead of using workarounds to match the required concurrency management properties to the provided properties, it is better to provide a fundamental solution to address this mismatch. Such a fundamental solution already exists in the form of *advanced transaction models* (ATMS). Many of such models have been developed, mostly between 1981 and 1997, and we discussed these in chapter 2. In these chapters we noted that each of these models addresses one specific set of issues of transaction management (which we term classical transaction management). Using such an advanced model allows for a fundamental approach to address the mismatch in concurrency management properties, i.e. we can match the design of the model to the design of the application such that the concurrency management properties match. We find, however, that ATMS are not used to achieve such a match, an observation which has also been made by other researchers, as we remarked in chapter 2.

12.1.2 Demarcation code

An important hurdle to take in adapting ATMS is the *complexity of demarcation code*. Demarcation code is the code of the application that instructs the TP Monitor where transactions start and end, and what operations on the data are included in the transaction. A known problem with demarcation code for classical transactions is that it crosscuts the application structure, i.e. it cannot be separated out from the application and placed into one module using traditional software engineering techniques. The crosscutting nature of demarcation code runs contrary to the software engineering principle of *separation of concerns*, which we discussed in chapter 3. Not obeying this principle makes the code hard to implement and maintain. A prominent technique to address the issue of crosscutting code is *Aspect-Oriented Programming* (AOP), which we also discussed in chapter 3. AOP has already been used to modularize demarcation code for classical transactions and we reviewed this work in 3.2. This work has significantly eased the task of using classical transactions in an application, but does not address ATMS. Demarcation code for ATMS is much more complex than classical demarcation code, as it needs to provide more instructions to the TP Monitor. Therefore, ATMS demarcation code will benefit even more from such a modularization, highly easing implementation and maintenance of such code.

An alternative technique addressing the crosscutting nature of transaction demarcation code is the technique of declarative transaction management in the EJB middleware standard. We described the EJB standard and declarative transaction management in 5.3, and analyzed its claims concerning separation of concerns. We found that in this technique separation of concerns is only present in the most trivial cases. Instead, in a realistic setting, separation of concerns is only partially achieved. In contrast to AOP, however, research has been performed to provide support of ATMS in declarative transaction management, and we also discussed this work in 5.3. This work has some important downsides in that it inherits the separation of concerns issues of declarative transaction management, and only allows for a limited support of ATMS.

The large volume, complexity and cross-cutting nature of demarcation code for ATMS hinders development and maintenance of the application being built. This significantly impedes the adoption of ATMS in multi-tier distributed systems. We need to address this issue to enable use of an ATMS in multi-tier distributed systems. To make demarcation code for ATMS easier to write and maintain, a good modularization of this demarcation code is required. This modularization needs to be general enough to cover a wide variety of ATMS. Furthermore, this also needs to be straightforward to use. As a result it will be possible, given an application, for an average multi-tier distributed systems programmer, to use an ATMS of which the design best fits the design of the application.

12.2 Contributions

The goal of this dissertation is to allow an application programmer of a multi-tiered distributed system to use an ATMS the design of which is best suited to the design of the application. To the best of our knowledge, this is the first time that the software engineering techniques of AOP and the use of a DSL have been used to address the use of an ATMS by application programmers.

Allowing application programmers to effectively use ATMS in multi-tiered distributed systems has the potential to restart the research on ATMS. As we have noted in chapter 2, between 1997 and now no fundamental research has been performed in this area. Recently, some efforts have been made to apply this research in a more contemporary setting, but these fail to address the application developer's perspective. We state that as long as using an ATMS is difficult, history will simply repeat itself and this research area will again fall dormant. By significantly easing the use of ATMS, however, these advanced models can be used and will be applied in new settings, which will spawn the creation of new models, restarting the research on ATMS.

We have achieved the goal of allowing application programmers to use an ATMS through a good modularization of demarcation code for ATMS and a concise specification mechanism for this code. We have a good modularization of demarcation code for ATMS because:

- We achieve a maximum separation of concerns for the transaction management concern.
- We support a wide variety of ATMS.
- We allow new ATMS to be created when needed.

Furthermore, the specifications of the use of an ATMS in an application are as concise as possible. This is because such specifications reify the concepts present in the ATMS, instead of the complex technical implementation. We elaborate on both these contributions next.

12.2.1 AOP to Modularize ATMS

Using AOP, we achieved a good modularization of the concern of using an ATMS in the application at hand. This modularization completely separates out the ATMS concern as an aspect for the application, providing the application has been designed with the usage of an ATMS in mind.

Our modularization features total obliviousness of the transaction management concern by the code for the base application concern within the transactional method.

No code for transaction management can be found in the code for the base concern if the design of the application takes transaction management into account. If all persistent data is stored in the database and no irreversible side-effects are present when executing an advanced transaction, the code is oblivious of the transaction management concern.

The only interaction between the transaction management concern and the base application concern is present in the caller of a transactional method. If the transaction rollbacks, the caller will be informed of this through an exception that has to be handled. Note that in 3.2 we have discussed a way to mitigate this, for classical transactions, through a more powerful aspect. This aspect also allows for the handling of rollbacks in the aspect definition. As a result, this part of the transaction management concern is also modularized to a certain extent.

Not only does our modularization feature obliviousness, it is also effective for a wide variety of ATMS. This allows a possibly large number of ATMS to be defined as an aspect using our approach. We require the ability to capture a large variety of ATMS as many ATMS have been published, each of these addressing only one specific set of drawbacks of the classical transaction model. To have an ATMS that suits the application at hand therefore requires the ability to choose one from this wide variety, or even to create a new ATMS if needed.

Tackling Tangled Aspect Code

We enabled the separation of the code for the ATMS concern itself into different modules, to allow the creation of a new ATMS. This allows for easy modification of an existing ATMS or creation of a new ATMS, when needed. Such modularization of the ATMS aspect itself is required as an ATMS can not be considered as one monolithic concern, as we analyzed in chapter 4. We have analyzed several ATMS and have shown that these consist of a combination of several (sub-)concerns, such as treating rollbacks, managing views and performing delegation. We have also shown that current, popular aspect languages used to modularize transaction management as an aspect do not provide adequate support to decompose the ATMS aspect into different modules. Instead, these tools require the code for the ATMS aspect itself to treat different concerns in a tangled way, a phenomenon which we called *tangled aspect code* in chapter 4. We therefore conceived an aspect language and weaver which successfully tackle this issue.

To the best of our knowledge, this is the first case where an aspect itself has been established to be composed of different cross-cutting concerns. This shows that the technique of separating a cross-cutting concern into an aspect can be applied recursively. The aspect itself may contain different cross-cutting concerns that are best separated into higher-level aspects. These then need to be combined to form one

complete aspect definition. We note that this idea appears similar to the concept of aspects on aspects [VC05], but this is not the case. In aspects on aspects, multiple crosscutting concerns at the level of the *base code* are created as aspects. In tangled aspect code, multiple crosscutting concerns are present at the level of the *aspect code*.

KALA: Kernel Aspect Language for ATMS

We have designed an aspect language that does allow for the ATMS aspect itself to be modularized into the different concerns present. This language is called *KALA*, which stands for **K**ernel **A**spect **L**anguage for **A**TMS, and was introduced in chapter 8. KALA provides support for a wide variety of ATMS, because it is based on the ACTA formal model for ATMS. The ACTA model, which we discussed in chapter 6 is an existing formal model known within the ATMS community, that allows for a formal specification of a wide variety of ATMS.

We have shown that KALA allows for a *clean modularization* of the different concerns within an ATMS by implementing the use of three existing ATMS in a modular fashion and creating a new ATMS. Concretely, in chapter 9, we implemented the use of the Nested Transactions, Sagas and Relatively Consistent Schedules ATMS, each in different KALA modules. This has given a first illustration of the benefit of modularizing ATMS code as the last ATMS we implemented could be implemented by adding two modules to the Saga definition. Also, we created a brand new ATMS: Cooperating Nested Transactions, by modifying a subset of the modules of Nested Transactions, again showing the benefit of this modularization.

The ability to use a *wide variety of ATMS* or to create a new ATMS by either modifying an existing ATMS or building a new ATMS from scratch shows KALA to have a wide applicability. This then allows an ATMS to be chosen of which the properties best fit the properties of the application being built. However, given the choice of an ATMS, writing KALA code to use this ATMS turns out to be extensive and complex work. We have significantly eased this using domain-specific languages.

12.2.2 Engineering of DSLs for ATMS

KALA is an aspect language which covers the domain of ATMS, and this flexibility has as a consequence that KALA code treats the implementation concepts of an ATMS. Writing KALA code entails combining these implementation concepts in a specific way to build behavior which corresponds to the ATMS being used. This has as a cost that KALA definitions are very extensive and complex, and a large amount of KALA code duplication occurs when using an ATMS within an application.

Not all application programmers, however, want to define a new ATMS. Most simply use an existing ATMS for the application being built. In other words, few

application programmers need to be exposed to the implementation details of the ATMS that is being used. Instead they will want to work with the concepts being used by the ATMS, and ignore the implementation details. We make this possible through the use of domain-specific languages.

Model-specific Languages

We have designed a family of domain-specific languages (DSLs), each language addressing a specific ATMS. This family of DSLs was introduced in chapter 10. By designing each language specific to one ATMS, each of these languages solely reflects the concepts of that ATMS ensuring the code written in this DSL is as concise as possible. Furthermore, we have a *family of DSLs*, where for each ATMS concept one syntax is used. This allows for reuse of the skill-set of the application programmer: as the same concept is always denoted identically, knowledge of this concept in one ATMS is straightforwardly transferable to other ATMS.

By defining five DSLs in our language family, we have shown that writing ATMS code using the DSLs we created is as concise as possible and allows for reuse of a programmers' skill-set. We have implemented DSLs for classical transactions, Nested Transactions, Sagas, Relatively Consistent Schedules and Cooperating Nested Transactions. For each DSL we have argued why code in this language is *as concise as possible*. An illustration of this is the DSL for Cooperating Nested Transactions, which is the same as for Nested Transactions, as both are composed of the same concepts. These two DSLs also illustrate the benefit of having a language family: as both DSLs treat the same concepts, the syntax for both languages is the same. This allows an application programmer with knowledge of the one language to use the other language immediately.

The use of DSLs significantly eases the use of an ATMS, which results in a higher productivity and maintainability of the application. We illustrated the ease of implementation by showing the code for three example applications in chapter 11, each using a different ATMS.

Extensibility: Engineering of ATMS

The DSLs in our language family are built on top of KALA: programs written in such a DSL are translated to the equivalent KALA code when weaving. This allows the easy treatment of a new ATMS. First the ATMS is written as KALA code, which is eased by the modular specification of ATMS in KALA. Code for the new ATMS can reuse some of these modules, modifying them or adding new modules. Second, as we have the KALA code for the new ATMS, the DSL compiler does not need to perform any aspect weaving. Instead of weaving, the DSL compiler only needs to produce the

equivalent KALA code. Weaving is then taken care of by the KALA weaver. This makes the DSL compiler much easier to implement.

An illustration of the extensibility of our approach is the creation of a new ATMS: Cooperating Nested Transactions (CNT), and its DSL. CNT was first defined, in chapter 9 as KALA code. In this definition, the modularity of KALA specifications allowed us to straightforwardly reuse and modify parts of the Nested Transactions ATMS. Second, a DSL for CNT was created, in chapter 10, translating DSL code to the equivalent KALA code.

12.2.3 Technical Contributions

Two technical contributions have been made to achieve the above result.

First, so as to be able to use an ATMS in a multi-tier distributed system, the middleware needs to provide support to use this ATMS in its TP Monitor. We have enabled this by building a *general TP Monitor for ATMS*, called ATPMos, based on the ACTA formal model. We gave an overview of this TP Monitor in chapter 7, and detailed the interface it exports to the application. Having a single TP Monitor support a wide variety of ATMS instead of having one TP Monitor per ATMS allows an application to use multiple ATMS at the same time, where each ATMS best suited to one particular part of the application. Furthermore, advantages made to the implementation of ATPMos immediately benefit all ATMS, which is not the case if we have one TP Monitor per ATMS.

Second, an aspect language such as KALA has no use if programs in this language can not be woven into the base code by an aspect weaver. Therefore, KALA is accompanied by an *aspect weaver*, which proves that KALA specifications are translatable to the equivalent demarcation code. We give an outline of KALA specifications are translated to the equivalent Java code in chapter 8, which allows for other implementations of such a weaver to be built.

12.3 Future Work

As any research endeavor, this work opens avenues for future work, both conceptually and technically. We see five major avenues for such work, which we discuss in this section. First we consider a structured approach to choose an ATMS for a given application. Second, we extend the domain of this research to also include multi-database systems and workflow management applications. Third, we consider the phenomenon of tangled aspect code, looking for a general solution. Fourth, we envisage more support for writing KALA code. Fifth and last we consider further extending KALA.

12.3.1 A Structured Approach to Select an ATMS

In chapter 11 we illustrated how applications can be built using an ATMS the design of which best fits the design of this application. We have, however, not considered the decision process which leads to the choice of a particular ATMS for the application at hand. For each example, we have matched the ATMS to the application in an unstructured way. It would be better to have a more structured approach to match an ATMS to a given application. The aim would be to have some methodology which eases the choice of ATMS, so that the best ATMS for a given application is found. To allow this, we propose that all ATMS be analyzed to determine their transactional properties. Based on these properties, a *classification of the ATMS* can be made. For a given application, the required transactional features of the application can than be used to look up the ATMS in the classification. If no fitting ATMS can be found, this classification is useful to find a number of ATMS which at least exhibit some of the required transactional properties. These can then serve as a basis to develop a new ATMS, reusing the appropriate parts of the different ATMS.

12.3.2 ATMS for Multi-Database and Workflow Management

We noted in chapter 2 that ATMS are not solely restricted to single-database multi-tier distributed systems. A significant amount of research on ATMS has also been performed for multi-database systems and workflow management systems. Both these areas pose significant additional requirements over single-database systems, and therefore we did not address such systems here.

Our approach for modularization of ATMS demarcation code can, however, also be applied in these domains. We therefore give a brief introduction to these domains and speculate how our work is applicable here. We start with Multi-Database support before treating workflow management.

Multi-Database support

Multi-Database Systems (MDBS) are typically the result of an integration process: an application is created, that uses a number of, pre-existing, heterogeneous Local Databases (LDBS), to provide an integrated view of that data. In such systems, the integrated system does not replace all existing systems: local applications continue to run on the LDBS, and therefore these databases may not be unduly modified [KPE92]. As a result, the most important requirement for MDBS is that some form of local autonomy for these LDBS must be guaranteed.

A classical example of MDBS is a travel agency application [ELLR90, KPE92]: This application provides an integrated way in which a travel agent can book a trip,

which consists of airline bookings, hotel reservations and possibly a rental car reservation. For each of these three steps, the databases of multiple companies are queried for availability, and depending on the clients' choice, bookings are made.

The MDBS transaction management system will break down the global transaction into a number of local transactions, submitted to the corresponding LDBS. Control of these transactions should be distributed to avoid bottlenecks, minimizing the global transaction duration. Even so, the global transaction will last significantly longer than the local transactions [KPE92], making the global transaction a long-lived transaction. Mind, however, that simultaneously with the global transaction, the local applications will still be accessing the local database. Therefore it is important to avoid that locks on the LDBS are held for a long period. However, while releasing such locks early increases concurrency, it also introduces the risk that the integrity constraints of the global, virtual, database are broken.

A well-known ATMS for multi-database systems is *Flex Transactions*[ELLR90, KPE92]. In Flex, a global transaction is split in different steps, as in Sagas. Three semantic extensions are added to the Sagas model, however: flexible transactions, mixed transactions and execution time. Flexible transactions allow the user to specify a number of alternative subtransactions for a given task, to be executed on a different database. Mixed transactions contain subtransactions that can be compensated and subtransactions that cannot. The first group will be committed when finished, the second group will only be committed when the global transaction commits. Execution time attaches a timeframe when a transaction may run at a LDBS. If the timeframe has passed, the subtransaction will fail.

We could design an aspect language targeted to this specific domain (as KALA is targeted to single-database ATMS), using the Flex model as a guideline. This new language would, however, not be able to reuse all of the features of KALA. For example, to set views and perform delegation require the LDBS to support these features. Furthermore, performing delegation of operations performed on one LDBS to another LDBS will be hard, if not impossible, due to the local autonomy. So therefore, views and delegation should be disallowed. An exception might be made if both source and destination transaction run on the same LDBS which is known to support setting of views and performing delegation. To implement flexible transactions, KALA dependencies and autostarts could be reused, in an extended form. They would specify alternative transactions for a given transaction and specify how to choose which transaction (out of all the alternatives) to commit, and which to abort. Committing only one transaction of a group is not supported in KALA, so some extension will have to be made to allow this. Considering mixed transactions, for a given transaction, dependencies and autostarts currently are used to specify compensating transactions, implicitly declaring the transaction to be compensatable. It would be better to have an explicit specification in the language, so the MDBS can easily identify compensa-

table transactions and immediately commit them. Finally, attaching execution time to a transaction is not implemented in KALA. A specification mechanism for this will have to be designed and added to KALA.

Workflow Management

In contrast to MDBS, there is no straightforward definition given of *workflow management systems* (WFMS). The description we give here is compiled from [RSS97, WS97, AM97]. Workflows are activities that result from an organizational process, often involving human and automated tasks. The workflow comprises the coordinated execution of these tasks. A WFMS automates coordination and control of, and communication and coordination between the tasks to satisfy the requirements of the organizational process. Example application areas for WFMS are office automation, computer supported cooperative work, and the ever-elusive paperless office. Workflow management is currently of considerable importance in the research on web-services, as such workflows are used to compose different services into one application.

As a token to the ambiguity of the definition of WFMS, the example application used in [WR92] is the same travel agency application we used above. Only here, emphasis is made on the dependencies between the different transactions. For example, if a client books a flight with a given airline he will get a discount on a rental car of a certain agency, which can not be modeled in a MDBS.

There are some disagreements regarding the applicability of ATMS for WFMS [WS97]: On the one hand, various researchers of the database community view workflows as an extension of ATMS. On the other hand, there are researchers who maintain that the requirements of the organizational processes are too rich with regard to modeling, coordination and run-time requirements to be able to be modeled by ATMS. Furthermore, while recovery by use of roll-backs is admissible for most ATMS, WFMS are known to also use roll-forward for recovery.

We do not wish to take sides in this debate here. We note that research exists applying ATMS for WFMS, and we now outline a well-known ATMS: the *ConTracts* model [WR92]. ConTracts addresses control flow dependencies and other semantic dependencies between individual, short transactions, by providing for an execution script for these transactions. In other words: a ConTract executes predefined steps (usually in the form of transactions), according to the control flow description given in a script. The script programming language is similar to parallel programming languages [WR92]. It allows for parallel execution of steps, sequences of steps, branches and loops of steps. Lastly, to be able to undo steps, compensating steps for each step need to be defined, and each step may be preceded by an entry invariant that must be met before the step will be executed. Forward recoverability of ConTracts is explicitly addressed in [RSS97], using recoverability at the step level and at the script level. At

the step level, if active steps are a transaction, they are rolled back, if not, manual intervention by an administrator is required. At the script level, a ConTract is forward recoverable: the state of the script is recovered, and each active step is recovered. After this phase, the ConTract will simply resume its execution.

Having a separate script for the specification of dependencies between different transactions of the workflow already provides for a separation of concerns. The code for the application logic is not polluted with execution script code. However, declaring methods as transactional is still performed by traditional demarcation code, which results in poor separation of concerns overall. It would be interesting to combine the script feature of ConTracts with KALA. This would result in a better separation of concerns, and also add the abilities of KALA regarding views and delegation to the workflow. Considering dependencies, we are unsure of the combination of a script language and KALA dependencies. The script may remove the need for KALA dependencies, but this will have to be investigated further.

12.3.3 Tackling All Tangled Aspect Code

This research uncovered, to our knowledge, the first case of where an aspect is itself composed out of multiple cross-cutting concerns. We termed this phenomenon tangled aspect code. We speculate that the concern of ATMS is not the only concern in which tangled aspect code is present. Indeed, if an aspect itself is complex enough to be composed out of different concerns, the possibility of tangled aspect code is real.

We have been able to tackle tangled aspect code through the use of domain knowledge. KALA was specifically designed for the ATMS domain, and to provide for modularization of tangled aspect code for this domain. Important research questions now are:

- Can we identify other complex aspects that suffer from tangled aspect code?
- For these complex aspects, are there domain-specific solutions to modularize this code?
- Is there a general way in which tangled aspect code can be tackled?

The first two items will require extensive and in-depth research of known aspects and of aspect candidates. We will need to determine if these are also composed out of different sub-concerns that, when combined, form tangled aspect code. For each of these aspects, a domain-specific solution may be possible, but it would be better if we could find a more general solution that provides relief for multiple aspects. This would give us a general two-level aspect mechanism. At the first aspect level an aspect is separated from the base code, and at the second aspect level the concerns within these aspects would be separated from each other.

Furthermore, the modularization process need not stop at the second level of aspects. There is no guarantee that a concern at the second level is monolithic. This concern can again be composed out of different concerns, and again we cannot be sure that there is no tangled aspect code at this level. In other words, a possible third level of aspects cannot be ruled out, and theoretically these levels can continue at infinitum. An interesting question here is about the nature of the aspect weaver at the third (and superior) levels. Is the aspect weaver for the third level the same as the weaver for the second level? If not, we have a clear distinction between aspect levels. If the weaver is the same, the levels are not clearly delineated. We could argue that third and superior levels are equal to composition of aspects at the second level.

It would be worthwhile to examine the aspects that suffer from tangled aspect code identified above, to determine the presence of such superior levels of aspects. We can first search for domain-specific solutions to modularize third-level aspects, and if successful consider generalizing the approach. For example, for the ATMS concern, we have not yet identified third-level aspects. However, we suspect that such cross-cutting code can be modularized using KALA modules. Therefore we do not need a separate weaver for third level aspects, and such superior levels are equal to the second level.

12.3.4 Tool support for KALA

In order to develop a new ATMS, the properties of this ATMS will have to be written down as KALA code. KALA, however, currently does not provide much support for development. We propose three points of interest to aid in developing KALA code: first a better form of error-handling, second more support for debugging, and third verification of dependency specifications.

Runtime error-handling in KALA is non-existent at the moment: *KALA fails silently*. If a dependency, view, delegation or termination is specified with an inappropriate argument, this specification is simply skipped silently. This is because name lookup for a transaction or a group of transactions returns the null transaction or an empty group when the lookup fails. On the null transaction and an empty group, dependency, views, delegation and termination are defined to have no effect. We cannot be sure that performing such operations on the null transaction or on an empty group is an error, however. An example of the correct use of this feature is the RCS definition in chapter 9, where this is used to allow the sequence of compensating transactions to be determined at runtime. A second case is the rebinding of a name of a transaction: no error is produced if an existing name for a transaction is overwritten, yielding a new transaction identifier for that name. Again, this can be an error, or this can be intentional. Therefore, we can not signal an error when dependencies, views, delegation and termination are performed with the null transaction or an empty group as argument, or if a transaction name is overwritten. Instead, what should be possible is

to *log these events*. If the application behaves erroneously, the log can be used to help determine the cause of the errors.

Currently, support for debugging KALA code is non-existent. However, as KALA code is woven into the Java code by source-code transformations, it is possible to use a Java debugger to observe the effects of KALA specifications. This can be aided by providing extra contextual information in comments interspersed throughout the code generated by the weaver. Furthermore, we can consider *integrating the KALA weaver* into an extensible Java IDE, such as Eclipse, as is performed in AspectJ [Asp05]. This integration could then also modify the debugger to show the appropriate KALA code instead of the annotated Java code when applicable.

Specifying dependencies in KALA is arguably a difficult task, especially if the network of dependencies becomes complex. Such a complex web of dependencies is difficult to model mentally and might lead to errors such as making it impossible for transactions to commit or abort. We have found a means to provide an explicit model of dependencies when implementing ATPMos, our TP Monitor for ATMS. In ATPMos, we model the dependencies of transactions using Petri nets [Pet77], and we discuss this in appendix A. Using Petri nets has advantages not only for ATPMos but also for specifying dependencies. Petri nets are represented graphically and drawing such an *explicit representation* of the dependencies allows the KALA developer to obtain a better picture of this network. Furthermore, Petri nets can be simulated, showing how the network of dependencies and the state of transactions evolves over time, and tools are available which perform such simulations graphically. Finally, an interesting feature of Petri nets is the ability to perform reachability analysis. Reachability analysis takes a net as input, i.e. a group of transactions and a network of dependencies between them, and determines if it is possible for this net to reach a certain state, for example, for all transactions to either commit or abort. Therefore, to aid in specification of dependencies, a tool can be built that uses Petri nets as a model, giving a graphical simulation of dependencies, and determining if certain errors will occur by performing reachability analysis.

12.3.5 Extending KALA and ATPMos

At the end of our discussion of the ACTA formal model in chapter 6, we have analyzed which of the ACTA features were not essential for a first implementation of this model in ATPMos. As a result, we have chosen to simplify the implementation of ATPMos somewhat, and correspondingly do the same for KALA. It is worthwhile to consider extending both KALA and ATPMos with features that were omitted, which we discuss here. Extending KALA and ATPMos will broaden their scope, i.e. we will be able to support a wider variety of ATMS. As the possible list of ATMS is open-ended, we can however not assess the impact of adding these features.

View Traversal Ordering

The first extension we consider is adding a view traversal order. As we have mentioned in chapter 6, ACTA allows for the definition of the *ordering of the views* of a transaction. This allows for a transaction to explicitly prefer the view of a given (group of) transaction(s) over another view, i.e. it will first attempt to access data using the view on the former before considering the view of the latter. As we have remarked in chapter 6, none of the ATMS for which an ACTA specification is available uses the view traversal order and therefore we omitted it. This could however be added in KALA by the use of a `viewTraversal` statement, which takes a list of transactions and transaction groups, and sets the view traversal to the ordering given in this list. This ordering can then be observed by the locking algorithm of ATPMos when acquiring locks, after a few modifications to this algorithm have been made.

Delegation Operation Selection

A second extension concerns delegation. In ACTA not necessarily all the operations of the source transaction are delegated to the destination transaction. We have, however, chosen to delegate all the work so as to simplify the implementation of ATPMos and KALA. To enable KALA and ATPMos to only partially delegate the work from one transaction to another transaction, we first need the ability to determine what part of the work should be delegated. In other words, we need to be able to *tag operations*, so that later they can be identified as having been included in the delegation. In ATPMos this is easily performed by adding an extra tagging parameter to each read or write call, and to the delegation operation. Each read or write then tags the operation, and when delegating the tag of the operations to be delegated is given. In KALA, however, this is more intricate. We can easily add a tag specification to the delegation operation, but we have no means to tag individual operations of the method. To tag individual operations, we need to target specific data accesses within the body of the method, which is not supported by KALA. This is because the join-point model of KALA is too simple: we can only specify join points targeting the start and the end of the method, and not specific data accesses, i.e. statements, in the method. To enable this, we require *statement joinpoints*, for example as proposed by Kellens and Gybels [KG05]. Using these joinpoints we can then target specific operations within the body of the method and tag these for later delegation.

Finer-grained Timing of Specifications

The concept of statement joinpoints also opens other horizons. We have now only been able to specify dependencies, views, delegation and termination at three specific points in the life-cycle of a transaction: begin, commit and abort time. Using statement

joinpoints allows us to provide such specifications at *any point of the execution* of the body of the transaction. Specifying dependencies, views, delegation and termination is straightforward, as ATPMos already provides support for such specifications at any moment in the life cycle of a transaction. We could, however, also consider having dependencies themselves consider points in the execution of the transaction instead of only placing relationships between begin, commit and abort of two transactions. This will require more work, as the dependency mechanism of ATPMos currently only supports the definition of dependencies that consider begin, commit and abort time of the transaction. To realize this requires a redesign of the dependency mechanism of ATPMos, which will also impact how KALA code interacts with this dependency mechanism.

Adding Exception Handling

Finally, a last extension to KALA we envisage is adding the ability to state exception handlers for transactional exceptions in KALA. This is similar to our work, discussed in 3.2, where the aspect definition of a transactional method allows for the inclusion of exception handlers. The purpose of these exception handlers is to catch exceptions related to transaction management, for example the occurrence of a rollback.

This research was not performed in KALA, but was limited to classical transactions. Technically, including this work in KALA should be straightforward. Care should be taken, however to consider the conceptual interaction between handling a transactional exception and the global rollback strategy of the advanced transaction model.

More Powerful Crosscut Specifications

As we have noted in chapter 11, when discussing the use of Nested Transactions, our crosscut specifications can become needlessly extensive if the same transactional properties apply to multiple methods. As we have shown, this requires the same properties to be repeated multiple times, as we show in the example below, taken from 11.2:

```
1 trans SARuleSet.replaceBallBearingAft() extends caller;
2 trans SARuleSet.getPart() extends caller;
3 trans SARuleSet.bleedBrakes() extends caller;
4 trans SARuleSet.dissassembleRearAxleHalf() extends caller;
```

It would be better to be able to use some form of wildcarding, for example as is performed in our previous work [Fab04b], to make the above specification more concise. For example, we could specify the above as follows (where the second line indicates that `removeRearAxleBlockingComponents` should not be made transactional).

```
1 trans SARuleSet.*() extends caller;
2 trans SARuleSet.removeRearAxleBlockingComponents() exclude;
```

Appendix A

ATPMos Implementation

The TP Monitor we built is called *ATPMos* (**A**TMS **TP Mo**nitor), and its implementation is sketched in this appendix. The focus of ATPMos is to provide an interface for ATMS, inspired on ACTA, to client programs, i.e. the middle tiers in multi-tier systems.

The first version of ATPMos was built earlier and was intended as a TP Monitor for classical transactions, therefore it did not provide support for any ATMS. The implementation of ACTA concepts that were missing was realized as a later step. In view of this, we first discuss how we designed ATPMos to support classical transactions and the interface to the application programs. Subsequently we show how the implementation of ACTA concepts was added, and the extensions to the interface.

A.1 Classical Transaction Management

In this section, we introduce the first version of ATPMos, our TP Monitor which supports a wide variety of ATMS. This first version, however, was built as a classical TP Monitor, and does not provide any support for ATMS. This is because ATPMos was originally developed to perform research on a cleaner separation of concerns for transaction management in EJBs [Fab04b] and to investigate default failure handling mechanisms for transaction rollback [Fab04a].

ATPMos is designed as a TP Monitor for EJB Entity beans. We assume that state is contained within such objects and that state modification happens through getter and setter methods. As a consequence, concurrency management is performed when these methods are called: calling a getter or setter on an Entity Bean, when in a transaction, requires that first a call is made to ATPMos. These calls inform ATPMos that a read or write will be performed, and contain as parameters the current

transaction identifier, the object and the field on which the read or write occurs. Internally, ATPMos uses a locking strategy to determine if this operation is allowed, acquiring locks if necessary. If the operation may proceed, the call will simply return. If the operation is not allowed at the moment, i.e. the required lock could not be acquired, the call will either block, until the lock is acquired.

We will now discuss the design of ATPMos in more detail, starting with an overview of the significant classes within ATPMos, before detailing how these classes interact to perform lock-based transaction management.

A.1.1 Class Layout

We start our discussion of the design of ATPMos by describing the major players in the class structure, as shown in the UML class diagram in figure A.1.

In ATPMos, all transactions are uniquely identified by an Integer object. All calls to ATPMos must include this identifier to indicate what transaction is the target of this operation. ATPMos knows of one 'special' identifier, the Integer zero, which is used to indicate the *null transaction*. The null transaction will never run, and therefore never keeps locks, and will never commit nor abort. Calls with as transaction identifier zero, i.e. pertaining to the null transaction, will therefore have no effect.

TransactionManager is the main entry point into the system. It is a Singleton, and all clients access it to either perform transaction primitive operations or before reads and writes. Of note is the `begin` method that starts a new transaction and returns an `Integer` object which is a unique identifier for this transaction, to be used in all subsequent communications between ATPMos and the client.

TransactionContext contains all transactional information for a given transaction. First, it keeps all locks on `Resource`s held by the transaction. Second, it maintains a list of all other `TransactionContext` objects on which this context is waiting, which occurs if it wants to obtain a lock that first has to be released by them. Third, it keeps a record of all original values in the database that were modified by this transaction. This allows them to be restored in case of rollback.

ResourceTable reifies all the data that is currently being accessed by transactions. `Read` and `write` requests pass through here, as we will see in the next section, and the table decides if locks should be acquired. Failure to acquire a lock is handled differently, depending on which subclass of the table is instantiated in ATPMos. The `RestartingTable` throws an exception, signaling that the transaction has to restart from scratch, while the `BlockingTable` blocks until the lock can be acquired.

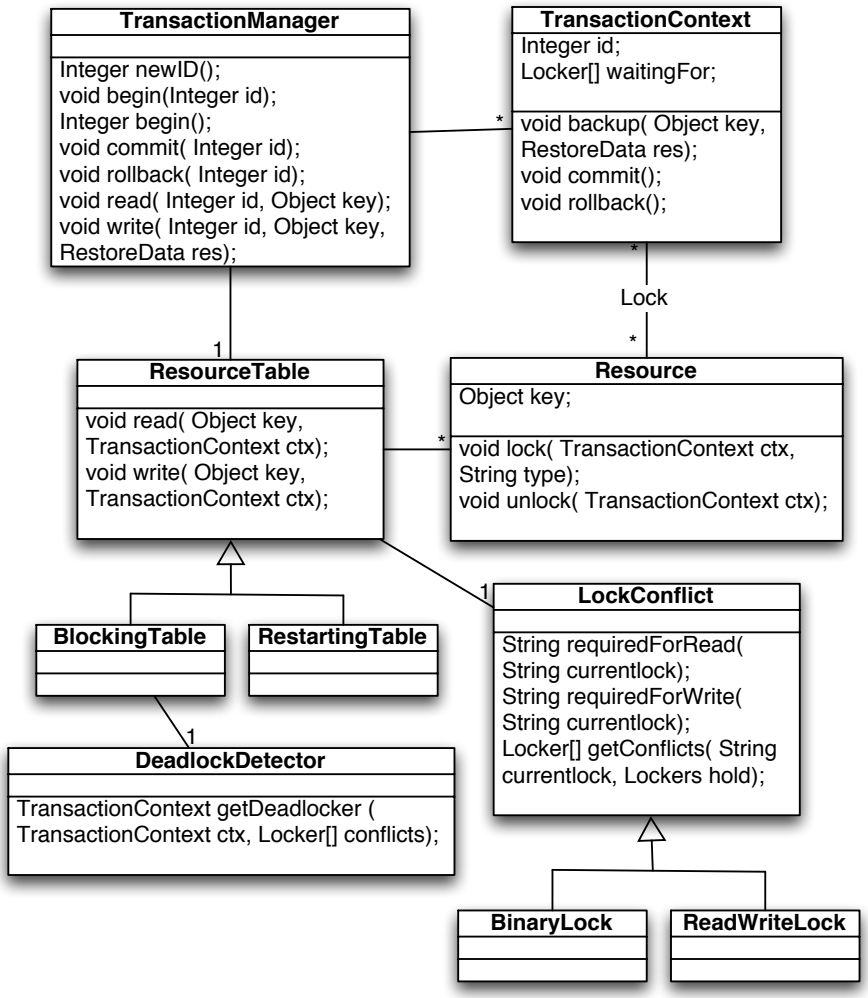

Figure A.1: UML class diagram of major classes in first version of ATPMos

DeadlockDetector implements a wait-for graph [CDK94] deadlock detection algorithm, to determine if a deadlock has occurred. If this is the case, a transaction involved in the deadlock will immediately be rolled back. This process is repeated until the deadlock is broken.

Resource represents one datum in the database, and keeps the lock information for that datum: which `TransactionContext`s keep a lock on that datum, and the nature of that lock.

LockConflict abstracts over different possible locking strategies that may be used by defining the required operations to allow this. The basic locking strategy is implemented in `BinaryLock`, while shared and exclusive locks are implemented in the `ReadWriteLock` class.

We did not include relatively minor elements, such as `RestoreData` and `Locker` here, as their purpose can be easily inferred from the context.

With the major players known, we can now proceed with discussing how they interact to provide transaction management.

A.1.2 Interactions

To complete our discussion of the design of ATPMos, we will now detail how the classes we introduced above interact. To show this interaction, we describe the control flow within the system for the beginning and ending of a transaction, as well as the read and write calls.

Beginning and Ending Transactions

To start a transaction, the client first needs to obtain a unique identifier for that transaction, by calling the `newID` method on `TransactionManager`, which will return an `Integer` object. Internally, this identifier is generated by a simple increasing counter. To start the transaction, the `begin` method is called, with as argument the identifier. A new `TransactionContext` is created with that identifier, and kept in a dictionary with as key the identifier. For convenience, a second `begin` method is included, which takes no arguments, to perform the above two steps in one go.

A transaction is ended either by a call to `commit` or to `rollback`, which we both discuss next. To commit a transaction the `TransactionManager` will first obtain the indicated `TransactionContext` and tell it to commit. This simply boils down to releasing all the transactions' locks and therefore the `TransactionContext` iterates over the `Resource`s it holds, and tells each to release the lock it holds. As a result of the locks being released, `TransactionContext`s waiting to obtain these locks will be notified of this release, and will attempt to acquire the lock, as we will discuss later.

To rollback a transaction, we cannot simply release the locks, as data may have been modified through writes, and these modifications have to be undone. Therefore, there is one change from the sequence above: the `TransactionContext` first uses the backup of old values it has kept, as we will see later, to restore the data to its original values before releasing all the locks.

Data Access

This section is the core of the concurrency strategy: while running inside a transaction, each call to a getter or a setter on the data is prepended by a call to the `TransactionManager`. This indicates a read or write is about to occur on a given datum, within a given transaction. Depending on the `Lockconflict` and `ResourceTable` subclasses instantiated in the system ATPMos will behave differently, achieving the intended concurrency strategy. We now first detail reads before we discuss writes.

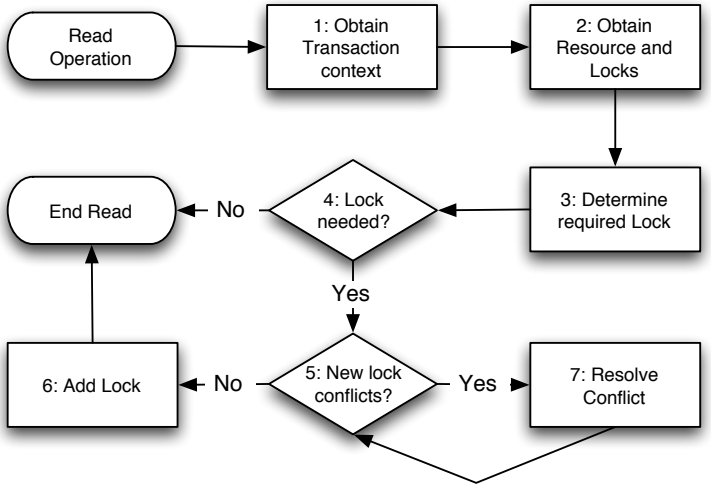

Figure A.2: Flow chart illustrating data access in ATPMos as result of a read operation

The different steps that occur during a read operation are illustrated in figure A.2 and proceed as follows:

1. The `TransactionManager` obtains the correct `TransactionContext`, and calls the `read` method on the `ResourceTable`, which performs the remainder of the work.

2. The `ResourceTable` obtains the indicated `Resource`, creating one if it is not yet

present in the system, and obtains the locks held by the `TransactionContext` on that `Resource`.

3. The `ResourceTable` queries the `LockConflict` to determine what type of lock should be acquired for a reading operation, given the locks currently held. Depending on the subclass of `LockConflict` instantiated, and the locks held by the `TransactionContext`, a type of lock is returned.

4. If no lock should be acquired, i.e. the `TransactionContext` already has the lock required for reading, the `read` method on the `ResourceTable` ends, ending the involvement of ATPMos.

5. The `ResourceTable` queries the `LockConflict` to determine if existing locks on the `Resource` (by other `TransactionContext`s) are in conflict with the lock to be acquired.

6. If no conflicts exist, the lock is added to the `Resource`, the `Resource` is added to the `TransactionContext` and the `read` method on the `ResourceTable` ends, ending the involvement of ATPMos.

7. Depending on the subclass of `ResourceTable` present in the system, this conflict is handled differently.

   With a `RestartingTable`, only one transaction can conflict with the requesting transaction, and the youngest conflicting transaction is immediately rolled back, with an exception thrown to the client of that transaction[1].

   With a `BlockingTable`, the transaction should wait until the conflicting locks are released before acquiring the lock. However, waiting may lead to deadlock, as we have seen in 2.1.3, and this needs to be addressed. This is performed by the `DeadlockDetector`, which implements a wait-for-graph deadlock detection, as we have shown in 2.1.3. The `BlockingTable` asks the `DeadlockDetector` for other `TransactionContext`s that would deadlock with this context, and selectively rollbacks these transactions until the deadlock is broken. If we are free from deadlock, and still unable to acquire the lock, the `TransactionContext` is updated with the list of lockers on which we wait, and the thread is suspended. When one of these conflicting locks is released, execution restarts at step 5.

8. No conflicts remain, so the lock is added to the `Resource`, the `Resource` is added to the `TransactionContext` and the `read` method on the `ResourceTable` ends, ending the involvement of ATPMos.

[1] Transaction age is easily determined using the transaction identifier. Restarting the youngest guarantees that no starvation occurs.

Write operations are treated similarly as read operations, therefore we do not include a full description here. Two differences are worth mentioning: in all the above steps 'reading' is replaced by 'writing', as is `read` by `write`, and as a last step, the `TransactionManager` uses the `backup` method on the `TransactionContext` to safeguard a copy of the original data for a later restore.

Having presented the control flow for both reads and writes, we have concluded the section on the interactions performed for data access management in ATPMos, ending our discussion on the design of ATPMos.

A.1.3 Conclusion

In this section, we detailed the design of the first version of ATPMos, our TP Monitor that supports a wide variety of ATMS. This first version does not support any ATMS, it was built purely as a classical TP Monitor, focussing on the concurrency management functionality of TP Monitors and the interface towards client applications.

ATPMos performs transaction management for data contained within Java objects, provided that data access occurs through getters and setters. Concurrency management is performed when these methods are called by first making a call to ATPMos. Also, to start and end transactions, calls must be made to ATPMos, as in traditional transaction demarcation.

We discussed the design of ATPMos in some detail, first giving an UML class diagram of the main players in the ATPMos design; the TransactionManager and TransactionContext, the ResourceTable, Resource and DeadlockDetector, and lastly the LockConflict class.

Name		Summary
Integer newID	()	Returns new identifier.
void begin	(Integer id)	Start transaction.
Integer begin	()	Start a new transaction, returns id.
void commit	(Integer id)	Commit transaction.
void rollback	(Integer id)	Rollback transaction.
void read	(Integer id, Object key)	Block until object can be read.
void write	(Integer id, Object key, RestoreData res)	Block until object can be written, keeps backup value.

Table A.1: ATPMos classical transaction interface.

With the responsibilities of these classes known, we proceeded with detailing how these classes interact to provide the required functionality. We have shown what happens when beginning and ending transactions, both by a commit and abort, and have given a detailed list of the operations performed when performing a read or write call, concluding our design discussion. An interface summary, containing of the methods in ATPMos to be called by clients, is given in table A.1.

Next, ATPMos is extended with ATMS support by adding implementation of the ACTA concepts that were missing from the first version. We discuss these ATPMos extensions that enable support by ATMS next. We begin with naming and grouping, proceed with views and delegation, and end with dependencies.

A.2 Naming and Grouping

ATPMos contains a naming and grouping service, inspired by the wide variety of name services available in distributed systems. The name service acts as a global dictionary, allowing transactions to bind their identifier to a key of their choosing, and to retrieve identifiers of other transactions based on such a key.

More concretely, the `bind` method of ATPMos takes an object and a transaction identifier, and inserts this association in a global dictionary, overwriting any previously stored association with that key. The `lookup` method takes as argument an object and will return the corresponding transaction identifier, or the null transaction if the dictionary does not contain that key. A consequence of this is that binding an object to the null transaction is equal to removing that binding from the name service. Associations can also be removed from the dictionary using the `unbind` method, which takes as argument the key to remove, or the `unbindAll` method, which takes as argument a transaction identifier and removes all bindings to that identifier.

ATPMos also provides support for groups of transactions, allowing a transaction to be added to a group using the `addToGroup` method. Adding to a non-existing group automatically creates it. Looking up a group, using the `lookupGroup` method, returns an array of transaction identifiers, which is empty if the group does not exist. A transaction can be removed from all the groups it is contained in by the `removeFromGroups` method. Lastly, the bindings to groups are removed using the `removeGroup` method.

Similar to a name service for transactions, ATPMos contains a name service for groups. This name service for groups is provided by ATPMos through the `bindGroup`, `lookupGroupBinding`, `unbindGroup` and `unbindGroupAll` methods. These behave analogously to the naming service methods we introduced above, save that they work for groups and that lookup of a non-existing group returns `null`. Note that group support does not operate with names registered by `bindGroup`, but only the original names as used in the grouping methods can be referred to. For example,

binding a new name `foo` to an existing group `bar` does not imply that the call `lookupGroup(foo)` returns the contents of the group `bar`. To achieve this, first the binding of `foo` must be obtained and then used as argument to `lookupGroup`, as follows: `lookupGroup(lookupGroupBinding(foo))`.

Name		Summary
void bind	(Object key, Integer id)	Register transaction identifier.
Integer lookup	(Object key)	Lookup transaction identifier.
void unbind	(Object key)	Unregister identifier key.
void unbindAll	(Integer id)	Unregister all identifier names.
void addToGroup	(Object key, Integer id)	Add identifier to group.
Integer[] lookupGroup	(Object key)	Lookup identifiers in group.
void removeFromGroup	(Object key, Integer id)	Remove identifier from group.
void removeGroup	(Object key)	Unregister group.
void bindGroup	(Object key, Object group)	Register group.
Object lookupGroupBinding	(Object key)	Lookup group.
void unbindGroup	(Object key)	Unregister group key.
void unbindGroupAll	(Object group)	Unregister all names for group.

Table A.2: ATPMos interface for naming and groups.

This concludes the introduction to the naming service of ATPMos. The naming service provides for a global registry of transactions and transaction groups, and its methods are summarized in table A.2.

A.3 Views

Views allow one transaction to see the intermediate results of another transaction. Views can take as target either another transaction or a group of transactions. We will first address views between two transactions, before detailing the use of groups to define a view.

To implement views between two specific transactions, the management of locks was modified in two steps: Firstly, `TransactionContext` was modified such that it also keeps a reference to all other transactions that are in the view, i.e. an extra association is kept to other `TransactionContext`s. Extra methods were added in

`TransactionManager`, `addView` and `removeView`, both having a source and destination transaction identifier as formal parameters. These then add or remove the destination `TransactionContext` from the sources' `TransactionContext` views. Secondly, these views are used when accessing data. We have shown in A.1.2 that, when attempting to read or write a datum in a transaction Ti, in step four `LockConflict` first decides what kind of lock should be acquired for an operation. We need to modify this step so that `LockConflict` also checks if another transaction Tj already has a lock of the required type and is in the transitive closure of the view of Ti. If this is the case, Ti does not need to acquire any lock and can simply proceed.

We need to implement it as above, and cannot just take a copy of the lock of Tj, i.e. acquire it by skipping step six, seven and eight. This is because we literally see what the other transaction sees every time the data is accessed; if in Tj the lock is modified or released, this change immediately applies to Ti. Therefore we need to verify this at every data access. Optimizations of this scheme through a caching mechanism are, of course, possible but we chose not to implement this, as this work does not fall within the scope of this thesis.

Also, this implementation of views has no impact on possible deadlocks: since Ti simply uses the lock already held by Tj, it does not wait for this lock to be released. Therefore, as no extra wait relationship is created in the wait-for graph, it can not cause a deadlock.

Using groups to define a view from one transaction to a set of other transactions is analogous to the implementation above. View groups are added and removed with the `addViewGroup` and `removeViewGroup` methods, which take the name of a group as destination. To implement this, `TransactionContext` also keeps a list of the names of groups that define a view for this transaction. When building the transitive closure of all transactions in the view of the current transaction, the `lookupGroup` method on the name service is called for all entries in this list. The resulting transactions are added to the closure, and used as roots for further building this closure. Again, optimizations of this scheme are certainly possible, but we chose to omit this as it does not fall within the scope of this thesis.

Note that our implementation of views is not fully compliant with the ACTA specification of views: if Ti uses a write lock on a datum x, held by Tj, and writes a new value to x, Tj will see that new value, which does not happen in ACTA. We have chosen not to remedy this, as it does not impact the interface towards the client application, and such an implementation would require a prohibitory large amount of work.

A.4 Delegation

Through delegation, a delegating transaction Ti delegates responsibility for committing or aborting data modifications to a delegatee Tj, which boils down to changing the ownership of all the locks from Ti to Tj.

This is implemented quite straightforwardly through an extra `delegate` method on `TransactionManager`, again having a source and destination transaction identifier as formal parameters. First the source and destination `TransactionContext`s are retrieved. Second all the locked resources that the source `TransactionContext` keeps are modified such that the association is changed to the destination `TransactionContext`.

Name		Summary
void addView	(Integer source, Integer dest)	Add a view from source to dest.
void removeView	(Integer source, Integer dest)	Remove the view from source to dest.
void addViewGroup	(Integer source, Object dest)	Add a view group to source.
void removeViewGroup	(Integer source, Object dest)	Remove the view group from source.
void delegate	(Integer source, Integer dest)	Delegate all resources from source to dest.

Table A.3: ATPMos interface for views and delegation.

This concludes our discussion on the implementation of delegation, easily the most straightforward extension of ATPMos. An interface summary for both views and delegation is given in table A.3. The next extension, dependencies, was more intricate, as we shall see next.

A.5 Dependencies

In this section we discuss how the dependency mechanism in ATPMos was implemented. We present this mechanism in detail, to illustrate how we enable an open-ended amount of dependencies, and how new dependencies can be created and added to the system. First we describe how we model dependencies, second we discuss adding and removing of dependencies, and third how enforcement of dependencies is performed, also detailing the client interface to the dependency system.

A.5.1 Modeling Dependencies Through Petri Nets

In general, there are a large variety of systems available for constraint checking, but we did not perform any systematic comparison on the different systems, as this falls beyond the scope of this thesis. We chose to use Petri nets [Pet77], which we will discuss next, because of a convincing case was made for the use of Petri nets in [VB03], and we had a Petri net evaluator, by the same author, at our disposal.

We now start with a brief introduction to Petri nets, before describing how we represent transactions as Petri-Nets and end with with a description of how dependencies are modeled.

Petri Nets in a Nutshell

A *Petri net* [Pet77] is a formal model of information flow, usually represented in a graphical form. The major use of Petri nets is modeling of concurrent systems where there are constraints on concurrence or precedence of these concurrences [Pet77]. Many varieties of Petri nets have been developed, but we restrict ourselves to the plainest form, which are simple Petri nets. In general, Petri nets have both a static and a dynamic side, and we first discuss the static side before talking about dynamic properties.

Statically, Petri nets are represented as graphs with two kinds of nodes: *places* and *transitions*, which are connected by directed arcs. A place represents a state of a (sub-)system, and is usually drawn as a circle. Transitions represent, as their name implies, a transition from one state to another, and are usually drawn as bars. The input places of a transition are all the places with an outgoing arc to that transition, and the output places of a transition are all the places to which the transition has an outgoing arc.

Dynamically, states are marked by *tokens*, usually drawn as black dots in places, which indicate that a (sub-)system is in the marked state. We show an example of a Petri net with marked states, called a *marked Petri net*, in figure A.3. Marked Petri nets are evaluated stepwise, each step consisting of the following rules:

1. A transition is enabled if first all its input places contain a token, and if second none of its output places contain a token unless that place is also an input place (such as, for example, the Tj-Committed place in figure A.3).

2. An enabled transition is fired. If, at a given time, different transitions are enabled, only one is fired. Note that there is no formal definition of which enabled transition is fired, so this selection can be considered as random.

3. Firing a transition happens atomically and consists of removing the token from each of the input places, and placing a token in each of the output places.

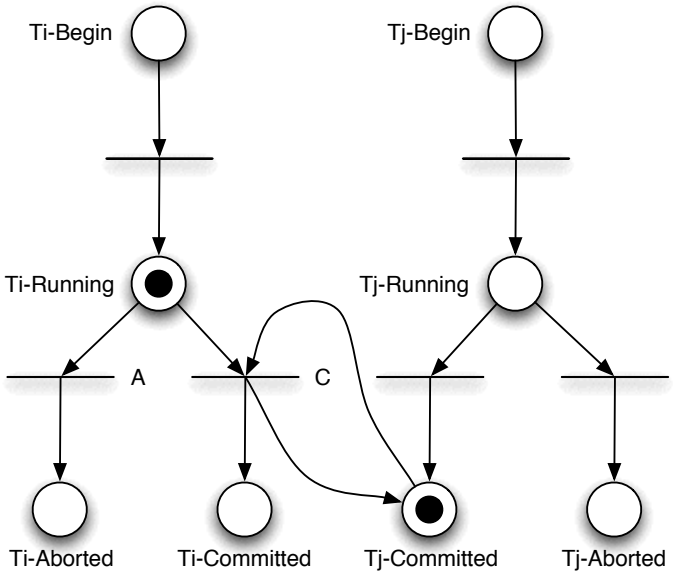

Figure A.3: An example marked Petri net. Both the transitions labeled A and C are enabled.

The evaluation of a Petri net causes the tokens to be re-distributed over the net every time an evaluation step finishes. The dynamic state of the net is defined by the distribution of these tokens, which is called the *marking*. Considering the marking in figure A.3, we see that both transitions labeled A and C are enabled. In the next step, the evaluator will, at random, choose one transition to fire, say C. In that case, both the token in Ti-Running and in Tj-Committed will be consumed, and a new token will be placed, both in in Ti-Committed as in Tj-Committed. If the marking would have been different, and Tj-Committed did not contain a token, only A would have been enabled. In that case, there is no choice: the token from Ti-Running would be consumed, and a new token would be placed in Ti-Aborted.

The above example should sound familiar: think of the places prepended with Ti (i.e. the left column) as the state of a transaction Ti, and the places prepended with Tj (i.e. the right column) as the state a transaction Tj is in. The above marking can be interpreted as follows: Ti is running and may choose to either commit or abort, since both the C(ommit) and A(bort) transitions are enabled. However, if Tj has not committed, i.e. Tj-Committed is not marked, Ti can only choose the A(bort) transition. This is very similar to a commit dependency between Tj and Ti.

The above example already is an indication of how we can use Petri nets to model transactional states and the dependencies between them. In the remainder of this section we first detail how we model transactional states, and then show how dependencies between transactions are modeled.

Transactions as Petri-Nets

In figure A.3 we have shown how different states of a transaction: beginning, running, committed and aborted can be modeled. We will now extend this representation somewhat, to take into account extra state information required for dependencies, and explicitly include the ACTA significant events begin, commit, and abort.

Figure A.4 shows the above representation side-by-side with the new representation. The changes we made are as follows:

Notation: The shape of places and transitions has been modified such that we could place the labels inside these nodes, instead of outside. This makes the graph easier to read when it has many arcs.

Beginning: Begin is not a state, but a significant event which transitions from non-existence to existence. To allow this, we created a place called #Pre, and made Begin a transition. Placing a token in #Pre allows the Begin transaction to be fired, i.e. represents the starting of a transaction.

Wait places: As we have stated in the beginning of this chapter, due to dependencies needing to be satisfied, transactions may be forced to wait when attempting to begin, commit or abort. This is modeled using Wait Begin, Wait Commit and Wait Abort places, with their respective OK transitions.

Commit or Abort?: The choice between taking the commit and abort transition can obviously not be made by the Petri net evaluator, as we must assume it will take a transition at random. The choice is made by the transaction body, firing the corresponding significant event, and this choice should be reflected in the model. This is done in analogy with the #Pre place, using #Com and #Abt places. Placing a token in one of these places represents, respectively, a commit or abort decision. With a token in Running and one in #Com, the evaluator will proceed with the Commit transition, and with a token in Running and one in #Abt, the evaluator will fire the Abort transition.

Note that this representation of transactions explicitly limits the list of significant events for a ATMS to begin, commit, and abort. As we have discussed in 6.3, a limited list of significant events suffices, as extra significant events can be mapped to these three, possibly taking effect on a different transaction.

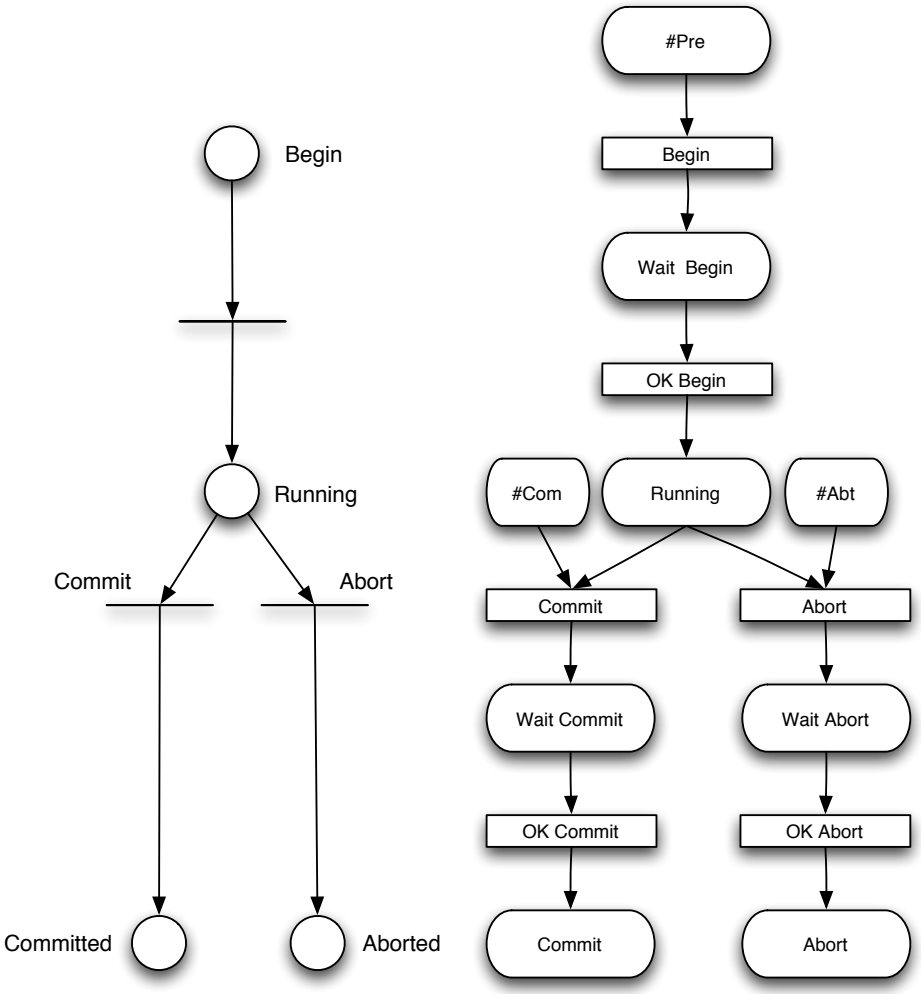

Figure A.4: Completing the transactional representation

Having shown the representation of transactions we use, we can now show how we model dependencies by adding extra arcs and transitions between different transactions.

Dependencies are Arcs and Transitions

A dependency between two transactions Ti and Tj places a constraint on the execution of significant events of transactions Ti and Tj, depending on the state of these transactions, as we discussed in 6.3. In other words, depending on the state of both transactions, certain transitions can no longer be enabled in the Petri net representation.

We achieve this by adding extra arcs and transitions to the graph, relating both transactions to each other by using input and output places from the different transactions. Consider, for example, figure A.5, which shows a $\mathcal{BCD}$ dependency placed between the transaction Tj on the left hand side of the figure, and the transaction Ti on the right hand side. Tj can not proceed beyond the Wait Begin place, until the Ti has reached either Commit or Abort. Furthermore, in case Ti commits, Tj will start, and in case Tj aborts, Ti is forced to immediately abort itself. This is, in effect, an enforcement of the Tj $\mathcal{BCD}$ Ti dependency.

A second example is figure A.6, showing a $\mathcal{CD}$ dependency: if the left hand side wants to commit, the right hand side has to have either committed or aborted, which satisfies the $\mathcal{CD}$ dependency. Beyond the two examples above, we have modeled six more dependencies of the twelve defined in ACTA. For brevity, and as the list of possible dependencies is open-ended, we do not include any more models here.

We have now shown how we formulate dependencies between transactions through Petri nets. We first gave a short introduction to Petri nets, second we have shown how we model transactions, and third we have demonstrated how dependencies are modeled. Next we will show how we can build a run-time model of the transactions in the system that reflects the state of these transactions and the dependencies between them.

A.5.2 Managing Dependencies

We have now seen how we model dependencies though the use of Petri nets, by letting each transaction have an associated Petri net, and having dependencies placed by adding arcs and transitions between these nets. We can now proceed with detailing how these models are constructed at run-time, and describe how the system can be extended with new kinds of dependencies when needed.

At run-time, we construct a representation of the running transactions and their dependencies as Petri nets. These nets are evaluated using a Petri net evaluator which

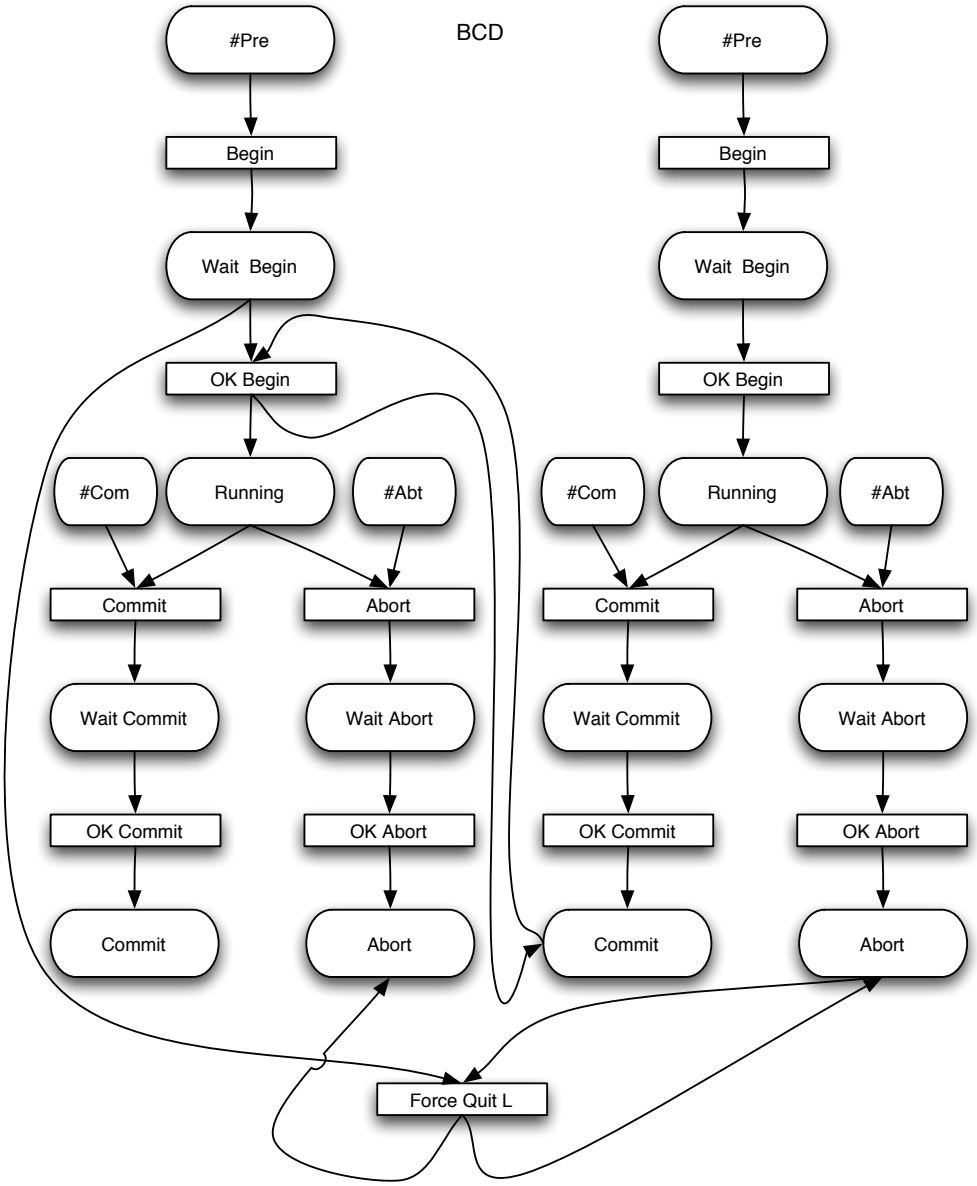

Figure A.5: $\mathcal{BCD}$ Dependency

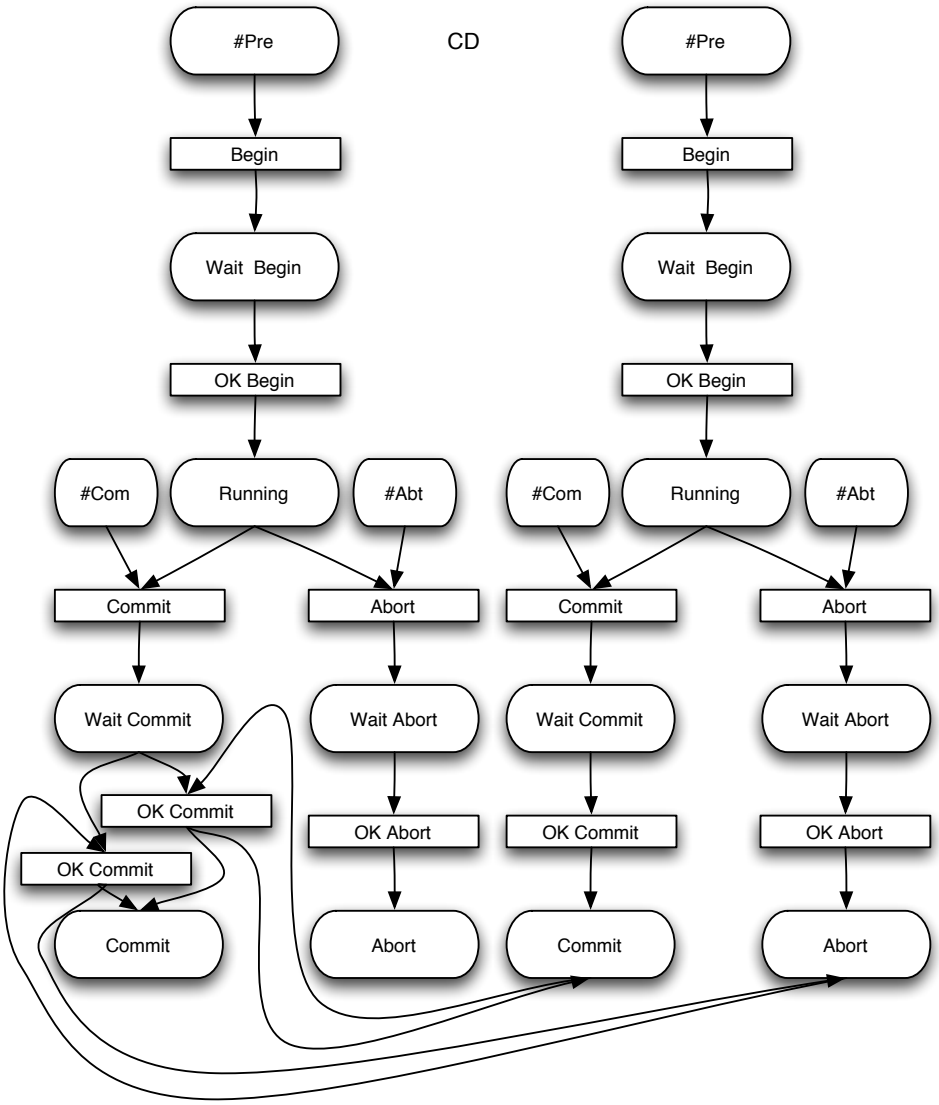

Figure A.6: $\mathcal{CD}$ Dependency

was put to our disposal, and of which we do not provide a full description here, as it is detailed in [VB03]. Important for this work is that the evaluator supports the use of dynamic Petri nets: nets where transitions, places and arcs can be added and removed at runtime. The representation of the Petri net that serves as input for the evaluator is not graphical, but a textual representation, which is detailed in [VB03]. Such a representation, as a data file, allows us to easily keep a generic form for transactions and dependencies in memory, and tailor these at runtime when adding them to the overall Petri net.

An important advantage of this system is that we can easily add more kinds of dependencies later on, without even needing to restart the evaluator. This is what enables us to have an extensible dependency evaluation mechanism.

A dependency is added to the system by calling the `addDependency` method on ATPMos, which takes as arguments the transaction identifiers of the affected transactions and the name of the dependency. Removal of transactions and their related dependencies is performed by calling the `removeTransaction` method on ATPMos. As removing transactions implies removing the related dependencies, for each transaction modeled in the net, we keep a list of dependencies that refer to that transaction. Removing a transaction entails first removing the dependency from the model, and subsequently removing the transaction itself.

We have now seen how we construct a model, at run-time, of the running transactions and their dependencies as Petri nets. We do this using generic representatios of transactions and dependencies, which are customized when added to the model. Removal of transactions automatically entails removing their associated dependencies. Having a model of the transactions and their dependencies at run-time, we can now discuss how this is used to enforce dependencies.

A.5.3 Enforcement of Dependencies

The run-time model of transactions and their dependencies we introduced above allows ATPMos to enforce dependencies, which we detail here.

As an illustration of how dependencies are enforced, we use figure A.7 which shows a Petri net at run-time. This net contains two transactions, labeled 41 and 42, with 41 $\mathcal{BCD}$ 42, 42 running, and 41 waiting to begin, which is indicated by the tokens in the respective places. Recall that we have already shown the $\mathcal{BCD}$ dependency in its generic form in figure A.5.

Let us first elaborate on the state of transaction 42 before discussing transaction 41. 42 is stuck in the running state because a token is missing from either #Com 42 or #Abt 42. A token placed in either of these places indicates the choice of the transaction to either commit or abort, respectively. As long as this choice remains unknown, the token will remain in the running state. Transaction 41 did not reach the running state,

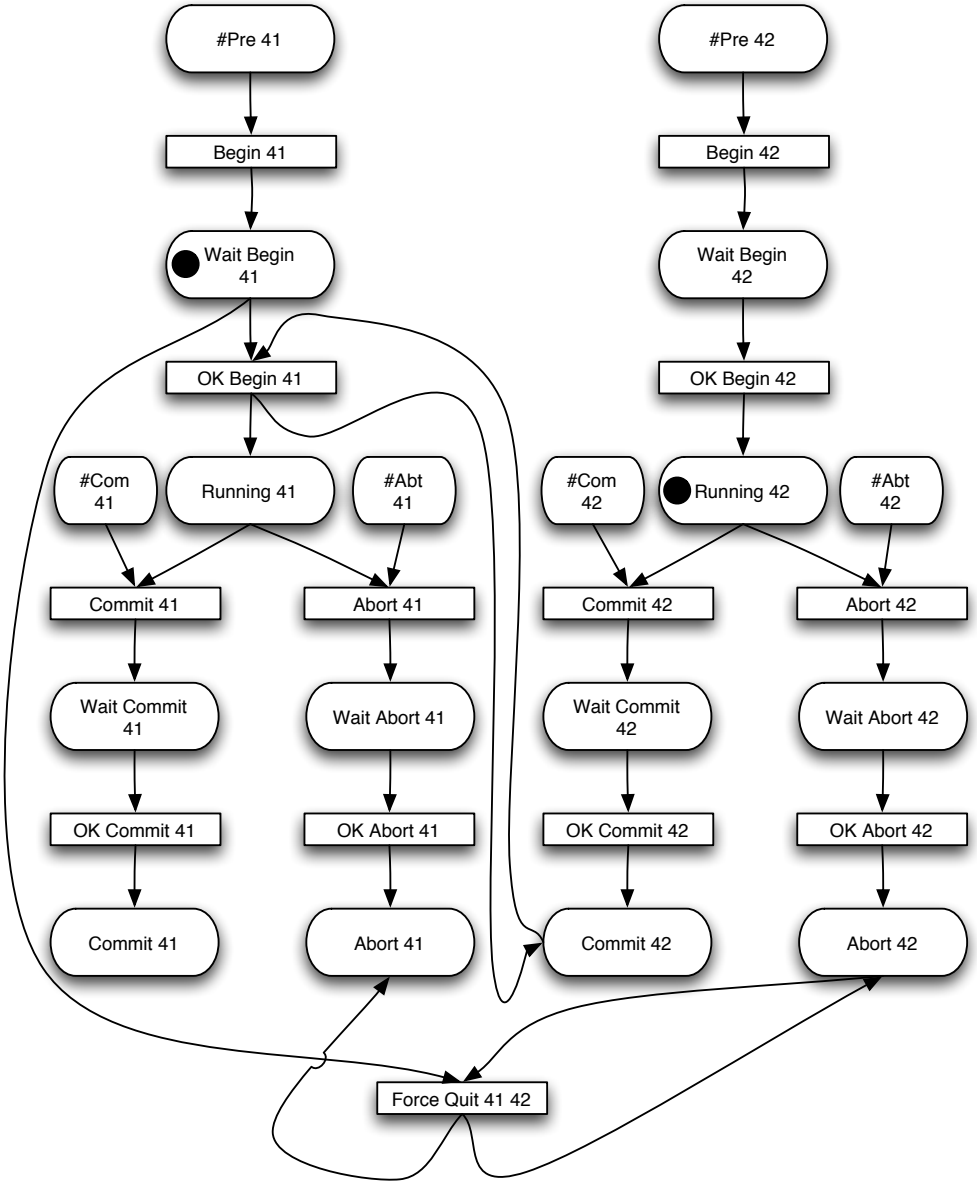

Figure A.7: Transactions 41 and 42, with 41 $\mathcal{BCD}$ 42. 42 is running, 41 is waiting to begin.

but remains waiting to begin, due to the $\mathcal{BCD}$ dependency. It will only be able to begin when allowed by the $\mathcal{BCD}$ dependency, i.e. when 42 has committed, which enables the OK Begin 41 transition. Furthermore, if 42 has aborted, 41 will not be able to begin, but will instead be forced to abort, as a result of the $\mathcal{BCD}$ dependency.

The model illustrated in figure A.7 can be used to verify and enforce dependencies at the level of the actual transactions, by maintaining a link between the model and the running transactions. This link is used at the significant events of the transaction, when a transaction wants to begin, commit or abort. It is at these points that dependencies must be checked, as these dependencies are expressed in relationship to the significant events of the transaction. Therefore, before a `begin`, `commit` or `abort` call is made to ATPMos, the transaction must issue a request to ATPMos, using the `mayBegin`, `mayCommit` and `mayAbort` methods, respectively. These request methods provide a bidirectional link between transaction and model: first they update the state of the model and second they provide for dependency verification and enforcement, as we show next.

The state of the model is updated by the request methods, which interface with the model by placing a token in the #Pre, #Com and #Abt places, respectively, which may allow some transitions to become enabled. The requester method, however will not immediately return, it will block until, respectively, the OK Begin, OK Commit and OK Abort transitions are fired, or an additional transition added by a dependency is fired. Blocking the requester method ensures that the transaction will not simply proceed, but will wait until an eventual dependency is satisfied.

When such a request method returns, it may return with or without a forcing indication. If a request method returns without a forcing indication, all dependencies are satisfied and the transaction may simply proceed. If the method returns with a forcing indication, this means that the model has followed a transition that was added by a dependency, i.e. was forced to proceed abnormally to satisfy a dependency. Therefore, to enforce this dependency, the transaction may not simply proceed, but must immediately go to the state indicated in the forcing indication.

In our example figure A.7, transaction 41 issued a `mayBegin` call, and that call is blocked, waiting for 42 to either commit or abort, due to the $\mathcal{BCD}$ dependency. If 42 decides to commit, it will issue the `mayCommit` call, placing a token in #Com 42. After a number of evaluation steps, the OK Commit 42 transition will be fired, allowing the `mayCommit` call to proceed and transaction 42 to commit. Firing the OK Commit 42 transition will enable the OK begin 41 transition, which will be fired. When OK begin 41 is fired, the `mayBegin` call issued by 41 will return without a forcing indication, satisfying the $\mathcal{BCD}$ dependency. Should 42 decide to abort instead of commit, a similar scenario is played out, with as major difference in outcome that the `maybegin` call issued by 41 will proceed with an indication forcing to abort. 41 must then immediately abort to satisfy the $\mathcal{BCD}$ dependency.

We shall not elaborate on the implementation of the constraint system in further detail. It suffices to say that the Petri net evaluator runs in its own thread, and the implementation of the waiting methods interact with the evaluator through message queues, blocking and unblocking themselves when required.

A.5.4 Conclusion

In this section, we have introduced the dependency system of ATPMos, which can later be extended with new dependencies when the need arises, as a result of using Petri nets to represent transactions and their dependencies. We repeat the different methods we added to ATPMos in the interface summary table A.4.

Name		Summary
void addTransaction	(Integer id)	Add transaction to the model.
void removeTransaction	(Integer id)	Remove transaction from the model.
void addDependency	(Integer left, String type, Integer right)	Add dependency of given type between two transactions.
Forcing mayBegin	(Integer id)	Block until transaction may begin.
Forcing mayCommit	(Integer id)	Block until transaction may commit.
Forcing mayAbort	(Integer id)	Block until transaction may abort.

Table A.4: ATPMos interface for views and delegation.

As a model for transactions and dependencies, we use Petri nets, which we briefly introduced. Transactions are modeled as a Petri nets, and dependencies are modeled using extra transitions and arcs placed between these nets.

An important feature for the dependency system of ATPMos is the ability to add new kinds of dependencies. This is achieved by adding the corresponding Petri net representations to the system, and can even be performed at run-time.

The Petri net model of the running transactions is grown at run-time, by adding transactions and dependencies. Removing of transactions, implies removing the dependencies that are associated with these transactions. Verification of dependencies is performed using this run-time Petri net model. We illustrated how verification and enforcement works within the model, and shown how this model is linked to the actual transactions through method calls, which are shown in the interface summary table A.4, resulting in dependency support for transactions.

A.6 Conclusion

Name		Summary
void bind	(Object key, Integer id)	Register transaction identifier.
Integer lookup	(Object key)	Lookup transaction identifier.
void unbind	(Object key)	Unregister identifier key.
void unbindAll	(Integer id)	Unregister all identifier names.
void addToGroup	(Object key, Integer id)	Add identifier to group.
Integer[] lookupGroup	(Object key)	Lookup identifiers in group.
void removeFromGroup	(Object key, Integer id)	Remove identifier from group.
void removeGroup	(Object key)	Unregister group.
void bindGroup	(Object key, Object group)	Register group.
Object lookupGroupBinding	(Object key)	Lookup group.
void unbindGroup	(Object key)	Unregister group key.
void unbindGroupAll	(Object group)	Unregister all names for group.

Table A.5: ATPMos interface overview for naming and groups.

In this appendix, we have introduced ATPMos, the TP Monitor we created for this dissertation. ATPMos was first built as a TP Monitor for classical transactions, and later extended to support ATMS by adding the required ACTA Primitives. We chose not to fully implement all these primitives, as this is not the scope of this thesis. We instead research how to program the software that interacts with these kinds of TP Monitors, and therefore we focused on the interface to ATPMos, summarized in tables A.5 and A.6. This interface does provide all the required features for the basic implementation we considered in 6.3.

The first version of ATPMos only supported classical transaction management for EJB Entity beans through the use of locks. Within a transaction, concurrency management is performed when getter and setter methods of the bean are called by first calling the `read` or `write` methods on ATPMos. These methods will block until that datum is locked for reading or writing, respectively, effectively performing concurrency management. Initiation and termination of transactions is performed using the `begin`, `commit` and `rollback` methods.

Of ATPMos, we described the class layout and the interactions of these classes before discussing the extensions of ATPMos into a TP Monitor for ATMS, following

Name		Summary
Integer newID	()	Returns new identifier.
void begin	(Integer id)	Start transaction.
Integer begin	()	Start a new transaction, returns id.
void commit	(Integer id)	Commit transaction.
void rollback	(Integer id)	Rollback transaction.
void read	(Integer id, Object key)	Block until object can be read.
void write	(Integer id, Object key, RestoreData res)	Block until object can be written, keeps backup value.
void addView	(Integer source, Integer dest)	Add a view from source to dest.
void removeView	(Integer source, Integer dest)	Remove the view from source to dest.
void addViewGroup	(Integer source, Object dest)	Add a view group to source.
void removeViewGroup	(Integer source, Object dest)	Remove the view group from source.
void delegate	(Integer source, Integer dest)	Delegate all resources from source to dest.
void addTransaction	(Integer id)	Add transaction to the model.
void removeTransaction	(Integer id)	Remove transaction from the model.
void addDependency	(Integer left, String type, Integer right)	Add dependency of given type between two transactions.
Forcing mayBegin	(Integer id)	Block until transaction may begin.
Forcing mayCommit	(Integer id)	Block until transaction may commit.
Forcing mayAbort	(Integer id)	Block until transaction may abort.

Table A.6: ATPMos interface overview for ATMS.

the ACTA model.

The first extension, Naming, was not a direct implementation of an ACTA concept, but rather a consequence of using these concepts. It is required for the code within a transaction to also be able to obtain a reference to other, running transactions, for example to place a dependency. The naming service we implemented in ATPMos allows a transaction to register itself under a certain key, using the `bind` method. This allows other transactions to obtain this reference, using the `lookup` method with that key. References are removed using the `unbind` and `unbindAll` methods. Furthermore, the naming service also supports groups of transaction identifiers, which can be added to using the `addToGroup` method. Lookup using the `lookupGroup` method returns an array of transaction identifiers. Transactions are removed from groups using the `removeFromGroup` methods and groups can be completely removed by the `removeGroup` method. Lastly, group support is also available in the naming service, through the `bindGroup`, `lookupGroupBinding`, `unbindGroup` and `unbindGroupAll` methods.

The next two extensions: views and delegation, are quite straightforward. Views are implemented through a change in the locking algorithm of ATPMos, now taking into account view relationships. These are set using the methods `addView` and `removeView` for single transactions and `addView` and `removeView` for groups. Delegation is implemented by one extra method `delegate`, which performs the required lock reassignment.

The last extension, dependency support, required more work due to the need for extensibility, i.e. the ability to add new kinds of dependencies when needed. Dependency support was implemented by using a Petri net representation of transactions and dependencies. New types of dependencies can be added when needed, thanks to the use of separate dependency descriptions. At run-time, dependencies are managed by adding transactions using the `addTransaction` method, and adding dependencies through the `addDependency` method. Removal of transactions from the run-time model is performed using the `removeTransaction` method, which also removes the associated dependencies. Concluding dependency support, we have shown how dependencies are enforced using the request methods `mayBegin`, `mayCommit` and `mayAbort`. These block until either the transaction may continue, or until it is forced to immediately proceed to another state.

Appendix B

Demarcation Code for ATMS

An Academic speculated whether a bather is beautiful if there is none in the forest to admire her. He hid in the bushes to find out, which vitiated his premise but made him happy. Moral: Empiricism is more fun than speculation.
— **Sam Webe**

This appendix contains all the Java code referenced from chapter 7. Each snippet of code is annotated with a discussion.

B.1 The Example Application

```
1  public void moneyTransfer
2     (BankAccount from, BankAccount to, int amount)
3  {
4     int from_amount = from.getAmount();
5     int to_amount = to.getAmount();
6     to.setAmount(to_amount + amount);
7     from.setAmount(from_amount - amount);
8
9     ReceiptCounter counter = ReceiptCounterStore.getCurrent();
10    int num_receipt = counter.getCount();
11    counter.setCount(num_receipt + 1);
12    Printer.getCurrent().
13       printTransferReceipt(from, to, amount, num_receipt);
14
```

```
15    OperationsLog log = OperationsLogStore.getCurrent();
16    int num_log = log.getCount();
17    log.setCount(num_log + 1);
18    log.addTransfer(from, to, amount, num_log);
19 }
```

This code has three distinct steps, the first step, from line 4 to 7 performs the actual bank transfer, the second step, from line 9 to 13, prints a receipt to be given to the customer, and the third step, from line 15 to 18, logs the transfer operation. Note that the sequence of these three steps has been explicitly fixed in the requirements of the application, which will be important later.

In the second step, `ReceiptCounter` (an EJB Entity Bean), is obtained from a repository, which returns a separate counter for each branch office. Each counter keeps the count of the receipts printed in this office, so that each receipt can be given a unique reference number when printed. Similarly, in the third step, the `OperationsLogStore`, keeps a global counter for all the operations performed by the bank. For brevity, we do not include the code for these objects, nor the code for `Printer`, as it is not relevant for the example.

B.2 Making the Transfer Operation Transactional

Making the code above transactional has resulted in the method code below, in which demarcation code is emphasized like this.

```
1  public void moneyTransfer
2     (BankAccount from, BankAccount to, int amount)
3     throws TxException
4  {
5     boolean printed = false;
6     int num_receipt;
7     TransactionManager txmgr = TransactionManager.getCurrent();
8     Integer tx_id = txmgr.begin();
9
10    try {
11       txmgr.read(tx_id, from);
12       int from_amount = from.getAmount();
13       txmgr.read(tx_id, to);
14       int to_amount = to.getAmount();
15       RestoreData tores = new RestoreData(to,"amount");
16       txmgr.write(tx_id, to, tores);
```

```
17          to.setAmount(to_amount + amount);
18          RestoreData fromres = new RestoreData(from,"amount");
19          txmgr.write(tx_id, to, tores);
20          from.setAmount(from_amount - amount);
21
22          ReceiptCounter counter = ReceiptCounterStore.getCurrent();
23          txmgr.read(tx_id, counter);
24          num_receipt = counter.getCount();
25          RestoreData countres = new RestoreData(counter,"count");
26          txmgr.write(tx_id, counter, countres);
27          counter.setCount(num_receipt + 1);
28          Printer.getCurrent().
29             printTransferReceipt(from, to, amount, num_receipt);
30          printed = true;
31
32          OperationsLog log = OperationsLogStore.getCurrent();
33          txmgr.read(tx_id, log);
34          int num_log = log.getCount();
35          RestoreData logres = new RestoreData(log,"count");
36          txmgr.write(tx_id, log, logres);
37          log.setCount(num_log + 1);
38          log.addTransfer(from, to, amount, num_log);
39
40          txmgr.commit(tx_id);
41       }
42       catch (TxException ex){
43          txmgr.rollback(tx_id);
44          if (printed)
45             Printer.getCurrent().
46                printTransferCancel(from, to, amount, num_receipt);
47          throw ex;
48       }
49    }
```

What is immediately obvious is that a lot of transaction demarcation code has been added, and that it is spread out all over the application code, cross-cutting the original concern. As a result, the size of the code has more than doubled: from 19 to 49 lines of code. This example illustrates how adding transaction demarcation code tangles an extra concern into the code and increases code size, which significantly decreases readability.

Let us first consider what has happened to the three steps of the previous version of the code. It is striking that in all steps the concern of transaction management interleaves the original application code. Also, as a result, the first step has grown to 10 lines, from 4, the second step to 9 lines, from 5, and the last step to 7 lines, from 5. A more detailed look reveals the origin of the interleaving of the two concerns. Each getter operation is preceded by a `read` call to ATPMos, and each setter operation is preceded by 2 lines: first a construction of a `RestoreData`, and second a `write` call to ATPMos. These interleaving additions are required, because both the `read` and the `write` call before a getter or setter are needed to allow ATPMos to perform concurrency management. Recall that a `RestoreData` is required in a write call to allow ATPMos to restore the original values of a datum when a rollback is performed.

Also, an additional significant inconvenience to the use of the `read` and the `write` calls is the fact that they are repetitive, yet slightly different. It is all too easy to perform copy-paste reuse here, and incorrectly modify the required parameters, which leads to errors. Similarly, when reading the code, the repetitive nature of these calls makes it all too easy to gloss over the details, again leading to errors, such as in line 19, where `to` and `tores`, should actually be `from` and `fromres`.

But let us assume this part of the demarcation code is correct, and continue with the remainder of the demarcation code. A reference to the `TransactionManager` is obtained in line 7 and used in line 8 to start the transaction. As a result of starting the transaction, a transaction identifier is returned, which is used in all subsequent calls to ATPMos. Considering code tangling, these lines do not pose as big a problem as the code treated above. However there is still some tangling, because this transaction code is present in the same module as the core logic of the application, i.e. the method definition.

As all calls to ATPMos may yield an exception, for example signaling that the transaction was aborted, the three steps of the transaction are included in a `try` block, that ends with the instruction to commit the transaction in line 40. The `catch` block treats exceptions by first performing a rollback of the transaction in line 44, and then possibly printing out a cancellation notice, again to be handed to the customer. Also, to notify the caller of the `moneyTransfer` method of the error when processing the transaction, the raised exception is re-thrown in line 47. Again code tangling is not as big an issue, but is still present.

The code for `printTransferCancel` is not relevant to this discussion, and therefore, we do not include it here. There is one catch to the use of `printTransferCancel`, however: it requires the receipt number of the printed transfer receipt, obtained in line 24. To be able to reference this variable here[1], its definition had to be moved out of line 24, in the `try` block, to line 6, which further tangles the code.

[1] In Java, the scope of a `catch` block does not include the body of the `try` block

This concludes the discussion of the transactional code for the `moneyTransfer` method. We first remarked on the cross-cutting nature of demarcation code and the increase in code size, which was more than doubled. Looking at the three steps of the application, we identified the tangledness of the two concerns and the repetitiveness in the `read` and the `write` calls as a likely source of errors. Furthermore, we identified the need for a new feature as a result of transactional exceptions, the printing of cancellation notices, and discussed how this is used.

A concluding remark we wish to make is with regard to the leap in complexity of the code. The original code for the `moneyTransfer` method we proposed in B.1 was quite simple, but lead to the tangled and complex code we have just discussed. Imagine the original code not so simplistic, already treating different concerns and closer to a real banking application in its complexity. The corresponding transactional code would be even more complex and therefore very hard to understand.

B.3 Simplification Through Wrappers

The use of a wrapper significantly simplifies and shortens the code, but does not do away completely with the burden imposed by transaction management. To illustrate this, we have rewritten the above code to make use of wrappers, included below, with code related to transaction demarcation emphasized:

```
1  public void moneyTransfer
2     (BankAccount from_orig, BankAccount to_orig, int amount)
3     throws TxException
4  {
5     boolean printed = false;
6     int num_receipt;
7     TransactionManager txmgr = TransactionManager.getCurrent();
8     Integer tx_id = txmgr.begin();
9
10    try {
11       BankAccountWrap from = new BankAccountWrap(from_orig);
12       BankAccountWrap to = new BankAccountWrap(to_orig);
13       int from_amount = from.getAmount(tx_id);
14       int to_amount = to.getAmount(tx_id);
15       to.setAmount(to_amount + amount, tx_id);
16       from.setAmount(from_amount - amount, tx_id);
17
18       ReceiptCounterWrap counter =
19          new ReceiptCounterWrap
```

```
20              (ReceiptCounterStore.getCurrent());
21          num_receipt = counter.getCount(tx_id);
22          counter.setCount(num_receipt + 1, tx_id);
23          Printer.getCurrent().printTransferReceipt
24             (from_orig, to_orig, amount, num_receipt);
25          printed = true;
26
27          OperationsLog log_orig = OperationsLogStore.getCurrent();
28          OperationsLogWrap log = new OperationsLogWrap(log_orig);
29          int num_log = log.getCount(tx_id);
30          log.setCount(num_log + 1,tx_id);
31          log_orig.addTransfer(from_orig, to_orig, amount, num_log);
32
33          txmgr.commit(tx_id);
34       }
35       catch (TxException ex){
36          txmgr.rollback(tx_id);
37          if (printed)
38             Printer.getCurrent().
39                printTransferCancel
40                   (from_orig, to_orig, amount, num_receipt);
41          throw ex;
42       }
43    }
```

At first sight transaction demarcation code does not seem as widely distributed over the original code, cross-cutting less the original concern. Also, the use of the wrapper has decreased the line count from 49 to 43, but we need to take into account that 4 lines were added due to formatting issues. Therefore, in total, the use of the wrapper has decreased the line count by 10 lines, or slightly over one fifth.

What is immediately obvious is that the three steps of the transfer operation have become much more legible, almost reverting back to their original form in B.1, and the role of the demarcation code is much less pronounced. The major differences are that we now create wrappers for each datum, and in the getter and setter calls we pass the transaction identifier as extra parameter.

Working with wrappers has a downside, however, because when handing out references to data, such as in `printTransferReceipt`, we should not pass out the wrappers, but pass out the original objects. Also, we have to take care that calls that are not transactional, such as the `addTransfer` call to the logging object, are not performed on the wrapper, but on the original object.

Looking at the remainder of the example, we see that it has not significantly changed from the previous version of `moneyTransfer`. It is obvious that this part of the demarcation code, including exception handling code using `printTransferCancel`, can not be simplified through a wrapper and is still tangled with the code for the core concern.

B.4 Transaction Management as an Aspect

Before discussing the transaction aspect, we first show the `moneyTransfer` code, below, with added demarcation code emphasized:

```
1  public void moneyTransfer
2     (BankAccount from, BankAccount to, int amount)
3  {
4     int from_amount = from.getAmount();
5     int to_amount = to.getAmount();
6     to.setAmount(to_amount + amount);
7     from.setAmount(from_amount - amount);
8
9     ReceiptCounter counter = ReceiptCounterStore.getCurrent();
10    num_receipt = counter.getCount();
11    counter.setCount(num_receipt + 1);
12    Printer.getCurrent().
13       printTransferReceipt(from, to, amount, num_receipt);
14    printed = true;
15
16    OperationsLog log = OperationsLogStore.getCurrent();
17    int num_log = log.getCount();
18    log.setCount(num_log + 1);
19    log.addTransfer(from, to, amount, num_log);
20 }
```

In the aspect code, written in the transaction aspect language, transaction properties are grouped per class. In each group, method signatures are listed and postfixed either with `new` or `none`, indicating whether a new transaction should be started or the method is not transactional. For the parameter list of method signature, the `*` wildcard may be used, indicating applicability regardless of parameter types. Also, default behavior for a bean can be set to be either `new` or `none`, using the `default` keyword instead of a method signature. We can also add instance variable declarations and define exception handlers for methods. Instance variable declarations are added

to the method using the `declare` keyword that declares instance variables that are accessible both in the body of the method as in the exception handler. Exception handlers are defined by appending a number of `catch` blocks, containing java code, to the transactional declaration of the method. Within this code, the transaction can be rolled back by simply calling a `txRollback()` method. If the transactional method throws an exception that is not caught by these handlers, or the handlers do not call the `txRollback()` method, the transaction commits when the method ends.

We include the description of the transactional properties of `moneyTransfer` below:

```
21 transactions Cashier
22 {
23    moneyTransfer(BankAccount, BankAccount, int) new
24       declare {
25          boolean printed = false;
26          int num_receipt;
27       }
28       catch (TxException ex) {
29          txRollback();
30          if (printed)
31             Printer.getCurrent().
32                printTransferCancel(from, to, amount, num_receipt);
33          throw ex; }
34 }
```

B.5 The Transfer Operation as a Saga

We have split up the `moneyTransfer` operation of B.1 into different methods. We have a top-level `moneyTransfer` method that subsequently calls three methods `transfer`, `printReceipt`, and `logTransfer`, each method corrsponding to a step of the saga. The new code for the `moneyTransfer` operation, without any transaction demarcation, is given below:

```
1 public void moneyTransfer
2    (BankAccount from, BankAccount to, int amount)
3 {
4    this.transfer(from, to, amount);
5
6    this.printReceipt(from, to, amount);
7
```

```
8     this.logTransfer(from, to, amount);
9  }
10
11 private void transfer
12    (BankAccount from, BankAccount to, int amount)
13 {
14    int from_amount = from.getAmount();
15    int to_amount = to.getAmount();
16    to.setAmount(to_amount + amount);
17    from.setAmount(from_amount - amount);
18 }
19
20 private void printReceipt
21    (BankAccount from, BankAccount to, int amount)
22 {
23    ReceiptCounter counter = ReceiptCounterStore.getCurrent();
24    int num_receipt = counter.getCount();
25    counter.setCount(num_receipt + 1);
26    Printer.getCurrent().
27       printTransferReceipt(from, to, amount, num_receipt);
28 }
29
30 private void logTransfer
31    (BankAccount from, BankAccount to, int amount)
32 {
33    OperationsLog log = OperationsLogStore.getCurrent();
34    int num_log = log.getCount();
35    log.setCount(num_log + 1);
36    log.addTransfer(from, to, amount, num_log);
37 }
```

Having seen the new `moneyTransfer` code, we can now proceed by adding demarcation code for the saga.

We have chosen to also use wrappers here, as in B.3, to simplify the code as much as possible, without using aspects.

We now tackle the different sections of the saga in increasing order of complexity. We start with describing the top-level method, `moneyTransfer`, before treating the steps of the saga. The last step, `logTransfer` is discussed first, as it is the easiest, then we discuss the first step, `transfer`, and end with the second step, `printReceipt`.

B.5.1 Saga Top-level

We now show the new code for the `moneyTransfer` method, with transaction demarcation code emphasized, and which is subsequently discussed in more detail:

```
1  public void moneyTransfer
2     (BankAccount from, BankAccount to, int amount)
3     throws TxException
4  {
5     TransactionManager txmgr = TransactionManager.getCurrent();
6     Integer tx_id = txmgr.newID();
7
8     txmgr.bind(Thread.currentThread(), tx_id);
9     txmgr.addTransaction(tx_id);
10
11    txmgr.mayBegin(tx_id); //disregard Forcing result
12    txmgr.begin(tx_id);
13    try {
14       this.transfer(from, to, amount);
15
16       this.printReceipt(from, to, amount);
17
18       this.logTransfer(from, to, amount);
19
20       Forcing cf = txmgr.mayCommit(tx_id);
21       if (cf != null)
22          throw new TxAbortedException();
23
24       txmgr.commit(tx_id);
25    }
26    catch(TxException ex){
27       txmgr.mayAbort(tx_id);//never fails
28       txmgr.rollback(tx_id);
29       throw ex;
30    }
31    finally {
32       Integer[] comps = txmgr.lookupGroup("Saga"+ tx_id + "Comp");
33       for(int i=0; i<comps.length; i++) {
34          txmgr.removeTransaction(comps[i]);
35       }
36
```

```
37          Integer[] steps = txmgr.lookupGroup("Saga"+ tx_id + "Step");
38          for(int i=0; i<steps.length; i++) {
39             txmgr.removeTransaction(steps[i]);
40          }
41
42          txmgr.removeGroup("Saga"+ tx_id + "Comp");
43          txmgr.removeGroup("Saga"+ tx_id + "Step");
44          txmgr.unbind("Saga"+ tx_id + "Comp");
45
46          txmgr.unbind(Thread.currentThread());
47       }
48    }
```

If we consider the code sequentially, we see that code is comprised of six parts, which can be compared to the demarcation code phases we discussed in 4.2: first the preliminaries, from line 5 to 9, then the begin phase, from line 11 to 12, followed by the running phase, from line 14 to 18, then either the commit phase, line 20 to 25 or the rollback phase, line 26 to 29, takes place, and the code ends with the cleanup phase, line 32 to 46.

In the preliminaries, a new transaction identifier is requested from ATPMos, in line 6, and the transaction is added to the dependencies model in line 9, which is the task of transaction initiation. In line 8, the saga top-level is registered in the naming service with as key the current thread, i.e. the task of name management. As the different steps within the saga run within the same thread, this enables them to obtain a reference to the saga, as we see later on.

The beginning phase consists solely of the transaction initiation task and is started with updating the transaction model of ATPMos in line 11. As we know that there are no dependencies placed on this transaction, we know that no `Forcing` will be returned, and we disregard it. The transaction is started in line 12 through the `begin` call.

The running phase is as before, since no data access is performed, and requires no further comment. The commit phase treats the transaction conclusion task. It starts with the `mayCommit` call, which serves two purposes: updating the transaction model of ATPMos, and verifying if we may commit or must rollback. This decision will be verified by the dependency mechanism, because the saga must rollback if a step rolls back. If we may commit, this is performed in line 24, otherwise an exception is raised, to be caught in the rollback phase.

Any transaction exceptions thrown by the different steps of the saga will indicate that the step has rolled back, which implies that the saga may not perform any more steps, and must be rolled back. In the rollback phase, which is also a part of the

transaction conclusion task, `mayAbort` updates the transaction model of ATPMos. This causes some dependencies to be satisfied, firing compensating transactions, as we discuss later. Again, no `Forcing` will be returned, and therefore, we continue with the rollback on line 28. The caller of the `moneyTransfer` method is informed of the saga rollback in line 19, by re-throwing the exception that caused the rollback of the saga.

The cleanup phase performs name management, group management and transaction termination. It is mainly responsible for updating the transaction model of ATPMos, cleaning the model of the different compensating transactions and steps. These have added themselves to the groups[2] referenced in line 32 and 37, and are removed from the model one by one in lines 34 and 39, which is transaction termination. Lastly, the groups themselves are removed in line 42 and 43, for the group management task. In line 44 a reference that was bound in the steps, which we see later, is removed, and the reference to the saga itself is removed in line 46, performing name management and ending the cleanup phase.

This concludes the description of the top-level saga code, of which the transaction demarcation code takes up the bulk. This code can be conceptually divided into five different low-level tasks, spread out through the demarcation code, with the sole goal to create and tear down a structure for the different steps of the saga to do their work in. Note that this code, by itself, is already larger than the full transfer operation in the classical model when using wrappers, as in B.3.

B.5.2 Last Step

We begin our discussion of the different steps in the saga at the end, with the last step. We have chosen to begin with the end step because the code is the easiest to understand, as it does not define any compensating transactions.

We first show the code, and detail it later:

```
49 private void logTransfer
50    (BankAccount from, BankAccount to, int amount)
51    throws TxException
52 {
53    TransactionManager txmgr = TransactionManager.getCurrent();
54    Integer tx_id = txmgr.newID();
55    txmgr.addTransaction(tx_id);
56    Integer saga = txmgr.lookup(Thread.currentThread());
57    txmgr.addToGroup("Saga"+ saga + "Step",tx_id);
```

[2]The group names are guaranteed to be unique, as they include the transaction identifier of the saga, which is guaranteed to be unique.

```
58
59      txmgr.addDependency(saga, "ad", tx_id);
60      txmgr.addDependency(tx_id, "wd" ,saga);
61      txmgr.addDependency(saga, "scd", tx_id);
62
63      Forcing bf = txmgr.mayBegin(tx_id);
64      if (bf == null)
65         txmgr.begin(tx_id);
66      else {
67         txmgr.rollback(tx_id);
68         return;
69      }
70
71      try {
72         OperationsLog log_orig = OperationsLogStore.getCurrent();
73         OperationsLogWrap log = new OperationsLogWrap(log_orig);
74         int num_log = log.getCount(tx_id);
75         log.setCount(num_log + 1,tx_id);
76         log_orig.addTransfer(from, to, amount, num_log);
77
78         Forcing cf = txmgr.mayCommit(tx_id);
79         if (cf != null)
80            throw new TxAbortedException();
81
82         txmgr.commit(tx_id);
83      }
84      catch (TxException ex){
85         txmgr.mayAbort(tx_id);//will always succeed
86         txmgr.rollback(tx_id);
87         throw ex;
88      }
89  }
```

Again, demarcation code, which is emphasized, takes up the bulk of the code, and again different tasks are present: transaction initiation, transaction conclusion, name management, group management, wrapper management, and dependency management. In the discussion of this code, we do not dwell on tasks we already talked about above, and solely mention the new tasks.

The same phases as in the code for the top-level return, save for the cleanup phase: preliminaries are from 53 to 57, the begin phase is from 59 to 69, the running phase

starts at 72 and ends at 86, followed by the commit phase, lines 78 to 82, and the code ends with the rollback phase: lines 85 to 87. We now detail these phases in the above sequence.

The preliminaries are straightforward, save for obtaining a reference to the saga in line 56, using the current thread as key, and registering this step in the corresponding group, in line 57.

The beginning phase starts with registering three dependencies between the saga and this transaction, which is the dependency management task. The need for these dependencies, as for all the dependencies in the rest of this example, has been formally stated in the Saga formal definition in [CR92]. The goal for placing these dependencies is to ensure correct rollback in case the saga is aborted. In general, in this code, we assume that the ATMS programmer knows what dependencies need to be placed where, and therefore we do not discuss the intricacies of these dependencies, and solely mention the desired result.

After dependencies have been placed, line 63 also performs dependency management: it verifies if the transaction may begin. This is not the case if the saga has already been marked for rollback. In this case, the method will immediately end, as no work may be performed. After this method ends, the top level of the saga, the `moneyTransfer` method, will initiate rollback of the saga, as we have seen above.

The running phase is changed from the code shown in B.5, in that we use wrappers for the data access. In fact, the code is identical to lines 26 through 30 of the wrappers example in B.3, therefore including the work to perform wrapper management.

The commit phase is identical to the commit phase of the top-level method, and therefore requires no further comment. The abort phase is also quite straightforward, as no dependency of the saga will ever disallow aborting a step.

This ends the discussion of the last step of the saga, which introduced the placing of dependencies in the beginning phase, and the disallowing of steps to begin in case of saga rollback. The main goal of the demarcation code was to ensure correct rollback of the saga if this step is aborted. The next step we discuss is the first step of the Saga, which shows further demarcation code for the rollback concern in the definition of a compensating transaction.

B.5.3 First Step

Each step of the saga, except for the last one, needs to define a compensating transaction that is run when the saga is roll-backed after that step has committed. For the first step in this example, the `transfer` step, we created an `undoTransfer` step, which performs the inverse of the transfer operation, as can be seen below:

```
90  private void undoTransfer
91     (BankAccount from_orig, BankAccount to_orig,
92     int amount, Integer tx_id)
93  {
94     TransactionManager txmgr = TransactionManager.getCurrent();
95
96     Forcing bf = txmgr.mayBegin(tx_id);
97     if (bf == null)
98        txmgr.begin(tx_id);
99     else {
100       txmgr.rollback(tx_id);
101       return;
102    }
103
104    try {
105       BankAccountWrap from = new BankAccountWrap(from_orig);
106       BankAccountWrap to = new BankAccountWrap(to_orig);
107       int from_amount = from.getAmount(tx_id);
108       int to_amount = to.getAmount(tx_id);
109       to.setAmount(to_amount - amount, tx_id);
110       from.setAmount(from_amount + amount, tx_id);
111
112       txmgr.mayCommit(tx_id); //always succeeds
113       txmgr.commit(tx_id);
114    }
115    catch (TxException ex){
116       //should never happen
117       ex.printStackTrace();
118    }
119 }
```

This method is transactional, with the demarcation code following the organization of previous transactional methods, with a preliminaries phase, beginning phase, running phase, and commit phase, which can be clearly deduced from the code.

In the preliminaries phase, no transaction identifier is requested from the `TransactionManager`, and this transaction is not added to the model. Because this is a secondary transaction, it will be subject to a number of dependencies, as we see later. Therefore, the adding has already been performed in the `transfer` method, and the transaction identifier to be used is passed as a parameter to the method.

Of key importance is the beginning phase, where we verify if the transaction may begin. The `mayBegin` call will block until dependencies are satisfied. This is used here to ensure that the compensating transaction will not immediately start when this method is called. Instead it will wait until it either may run, or it must immediately abort. The former will occur at a given point of rolling back the saga, and the latter will occur when the saga commits.

The running phase of `undoTransfer` is quite straightforward, using wrappers for transactional data access, and performing an inverse money transfer between the two accounts. As commitment of compensating transactions must always succeed, we disregard any forcing result. Transaction rollback should never occur, but this can not be guaranteed (e.g. in case of a forced rollback due to a deadlock, as we have discussed in 2.1.3), so if the abort phase is reached, the error is logged. The intention is that this error is later resolved manually by a database administrator, as is also performed in [KS03].

Having seen the code of the compensating transaction, we still have to address how it is linked from the `transfer` step, which we perform in following code:

```
120 private void transfer
121    (BankAccount from_orig, BankAccount to_orig, int amount)
122    throws TxException
123 {
124    TransactionManager txmgr = TransactionManager.getCurrent();
125    Integer tx_id = txmgr.newID();
126    txmgr.addTransaction(tx_id);
127    Integer saga = txmgr.lookup(Thread.currentThread());
128    txmgr.addToGroup("Saga"+ saga + "Step",tx_id);
129
130    final Integer comp_id = txmgr.newID(); //for compensation
131    txmgr.addTransaction(comp_id);
132    txmgr.addToGroup("Saga"+ comp_id+ "Comp",comp_id);
133    txmgr.bind("Saga"+ comp_id+ "Comp",comp_id);
134
135    final BankAccount compfrom = from_orig; //for inner class
136    final BankAccount compto = to_orig; //for inner class
137    final int compamount = amount; //for inner class
138
139    Runnable compensator = new Runnable()
140 {
141    public void run(){
142       undoTransfer(compfrom, compto, compamount, comp_id);
```

```
143     }
144 };
145 
146     txmgr.addDependency(saga, "ad", tx_id);
147     txmgr.addDependency(tx_id, "wd" ,saga);
148     txmgr.addDependency(comp_id, "bcd" ,tx_id);
149 
150     new Thread(compensator).run();
151 
152     Forcing bf = txmgr.mayBegin(tx_id);
153     if (bf == null)
154        txmgr.begin(tx_id);
155     else {
156        txmgr.rollback(tx_id);
157        return;
158     }
159 
160     try {
161        BankAccountWrap from = new BankAccountWrap(from_orig);
162        BankAccountWrap to = new BankAccountWrap(to_orig);
163        int from_amount = from.getAmount(tx_id);
164        int to_amount = to.getAmount(tx_id);
165        to.setAmount(to_amount + amount, tx_id);
166        from.setAmount(from_amount - amount, tx_id);
167 
168        Forcing cf = txmgr.mayCommit(tx_id);
169        if (cf != null)
170           throw new TxAbortedException();
171 
172        txmgr.addDependency(comp_id, "cmd" ,saga);
173        txmgr.addDependency(comp_id, "bad" ,saga);
174 
175        txmgr.commit(tx_id);
176     }
177     catch (TxException ex){
178        txmgr.mayAbort(tx_id);//will always succeed
179        txmgr.rollback(tx_id);
180        throw ex;
181     }
182 }
```

We do not detail the line numbers of all the phases here, as they can be readily inferred from the source code. Instead we focus on the parts that merit special attention due to a new task: management of secondary transactions (i.e. the compensating transaction), ignoring all other tasks.

The additions to the preliminaries phase, line 130 to 144, all are due to the compensating transaction. First the transaction identifier is created and it is added to the transaction model. Then the compensating transaction is added to the compensation group, and registered as the last created compensating transaction, which is used in the next step of the saga. Lines 135 though 137 are present due to the requirements of using an inner class in 139 to 144. The sole purpose of the `compensator` object built by this inner class is to call the `undoTransfer` method in a separate thread, and to pass it the source and destination accounts, along with the amount to be transfered back and the identifier of the transaction to be used.

In the beginning phase, after dependencies have been placed, the `compensator` object is given to a new thread and started, in line 150. Due to the fact that a $\mathcal{BCD}$ dependency is placed between the compensating transaction and this transaction in line 148, compensation will not start, as the `transfer` method must commit before `undoTransfer` may begin. As a result, the new thread will be blocked until all dependencies on its begin phase are satisfied, as we have stated when discussing the `undoTransfer` method. Also, further dependencies on the compensating transaction are placed in the commit phase of the code, which ensures that the compensating transaction will only run when the saga aborts.

This concludes the discussion of the first step of the saga, where we described the impact of using a secondary transaction, in this case a compensating transaction, for rollback of the saga. We have seen that adding this compensating transaction not only requires an extra method to be added, but also significantly increases the complexity of the code, to enable this method to be called in a separate thread.

In the last step of the saga, compensation also plays an important role, as it is treated slightly differently, which we see next.

B.5.4 Second Step

The second step declares a compensation step: the printing of a transfer cancellation notice. We assume that a method implementing this has been defined on the current printer, as we did in the first transactional bank transfer operation above. This has as a result that we do not need to implement an extra semantical undo method for this secondary transaction, and instead call the printer method directly from the inner class we create, as can be seen below:

```
183 private void printReceipt
184    (BankAccount from_orig, BankAccount to_orig, int amount)
185    throw TxException
186 {
187    TransactionManager txmgr = TransactionManager.getCurrent();
188    Integer tx_id = txmgr.newID();
189    txmgr.addTransaction(tx_id);
190    Integer saga = txmgr.lookup(Thread.currentThread());
191    txmgr.addToGroup("Saga"+ saga + "Step",tx_id);
192    Integer prev_comp_id = txmgr.lookup("Saga" + saga + "Comp");
193
194    final Integer comp_id = txmgr.newID(); //for compensation
195    txmgr.addTransaction(comp_id);
196    txmgr.addToGroup("Saga"+ comp_id+ "Comp",comp_id);
197
198    final BankAccount compfrom = from_orig; //for inner class
199    final BankAccount compto = to_orig; //for inner class
200    final int compamount = amount; //for inner class
201    final int[] num_receipt_holder = new int[1];
202
203    Runnable compensator = new Runnable()
204 {
205    public void run(){
206       TransactionManager txmgr = TransactionManager.getCurrent();
207
208       Forcing bf = txmgr.mayBegin(comp_id);
209       if (bf == null)
210          txmgr.begin(comp_id);
211       else {
212          txmgr.rollback(comp_id);
213          return;
214       }
215       try {
216          Printer.getCurrent().printTransferCancel
217    (compfrom, compto, compamount, num_receipt_holder[0]);
218          txmgr.mayCommit(comp_id); //always succeeds
219          txmgr.commit(comp_id);
220       }
221       catch (TxException ex){
222          //should never happen
```

```
223             ex.printStackTrace();
224          }
225       }
226    };
227
228       txmgr.addDependency(saga, "ad", tx_id);
229       txmgr.addDependency(tx_id, "wd" ,saga);
230       txmgr.addDependency(comp_id, "bcd" ,tx_id);
231
232       new Thread(compensator).run();
233
234       Forcing bf = txmgr.mayBegin(tx_id);
235       if (bf == null)
236          txmgr.begin(tx_id);
237       else {
238          txmgr.rollback(tx_id);
239          return;
240       }
241
242       try {
243          ReceiptCounterWrap counter =
244             new ReceiptCounterWrap
245                (ReceiptCounterStore.getCurrent());
246          int num_receipt = counter.getCount(tx_id);
247          num_receipt_holder[0] = num_receipt;
248          counter.setCount(num_receipt + 1, tx_id);
249          Printer.getCurrent().printTransferReceipt
250                (from_orig, to_orig, amount, num_receipt);
251
252          Forcing cf = txmgr.mayCommit(tx_id);
253          if (cf != null)
254             throw new TxAbortedException();
255
256          txmgr.addDependency(prev_comp_id, "wcd", comp_id);
257          txmgr.addDependency(comp_id, "cmd" ,saga);
258          txmgr.addDependency(comp_id, "bad" ,saga);
259
260          txmgr.commit(tx_id);
261       }
262       catch (TxException ex){
```

```
263         txmgr.mayAbort(tx_id);//will always succeed
264         txmgr.rollback(tx_id);
265         throw ex;
266     }
267 }
```

In terms of transaction demarcation code tasks, this method does not introduce any new tasks, so we do not fully discuss this code here, we only point out the remarkable parts .

The most noticeable impact is the growth of the inner class responsible for building the `compensator` object. This now makes its run method transactional in itself, as this transactional behavior should not be placed in the code of the printer. Furthermore, we need to perform a trick using an array in lines 201, 217 and 247: we need to pass the identifier of the receipt to the compensating method, however this is not yet known when the inner class is created. To bypass this, we use an array as a container, which is filled in at line 247, but already referenced in line 217. We know that, at run time, the compensating transaction will only run after the `printReceipt` transaction has committed, which implies that the container will have been filled in, and therefore this will never produce an error.

Also meriting our attention is the lookup of the previous compensating transaction in line 192 and the dependency placed in the commit phase, in line 256, between this compensator and the previous. This is done to ensure that compensation proceeds in the required order, first compensating this step, before compensating the first step of the saga.

Appendix C

KALA Grammar Specification

This appendix contains a formal specification of the KALA grammar and the tokens for this grammar. We first provide the grammar for the KALA language, before giving a scanner definition for this grammar.

C.1 KALA Grammar

We first show the grammar for the KALA language, which was defined using the SmaCC parser generator [Ref05]. SmaCC, which stands for Smalltalk Compiler-Compiler, is a parser generator for the Smalltalk programming language, the language in which our aspect weaver is implemented. SmaCC allows production rules for the grammar to be written in quite a straightforward way, including semantic actions for each rule. The code below shows all the rules for the grammar, of which we have removed the semantic actions for clarity. In our implementation, the semantic actions are used to build a parse tree of the KALA program, which the weaver then uses to weave the appropriate aspect code.

We provide an excerpt of the syntax for SmaCC production rules, taken from [Ref05] here, for convenience:

- Each production rule consists of a non-terminal symbol name followed by a `:` separator which is followed by a list of possible productions separated by vertical bar `|`, and finally terminated by a semicolon `;`.

- Each production consists of a sequence of non-terminal symbols, tokens, or keywords.

- Non-terminal symbols are valid Smalltalk variable names and must be defined somewhere in the parser definition. Forward references are valid.
- Tokens are enclosed in angle brackets as they are defined in the scanner e.g., `< IDENTIFIER >` (the scanner is discussed next), and keywords are enclosed in double-quotes, e.g. `"alias"`.
- `Symbol * or Symbol +` makes a repeating `Symbol`. The `*` repeats zero or more times, and the `+` repeats one or more times.
- `Symbol ?` makes `Symbol` optional.
- `( Productions )` groups the items in `Productions`.
- `[ Productions ]` is equivalent to `( Productions ) ?`.

The left-hand side of the first grammar rule is the start symbol, which in our case is `goal`, as can be seen below.

Grammar definition

```
goal :
   tx_declaration_list
   ;

tx_declaration:
   method_signature "{"
   [naming_grouping_autostarts]
   [tx_begin] [tx_commit] [tx_abort] "}"
   ;

naming_grouping_autostarts:
   naming_grouping_autostart_list ";"
   ;

tx_begin:
   "begin" "{" se_declaration_list [";"] "}"
   ;

tx_commit:
   "commit"  "{" se_declaration_list [";"] "}"
   ;
```

```
tx_abort:
   "abort"  "{" se_declaration_list [";"] "}"
   ;

naming_grouping_autostart:
   naming
   | grouping
   | autostart
   ;

naming:
   tx_alias
   | tx_name
   ;

grouping:
   groupAdd
   | groupName
   | groupAlias
   ;

autostart:
   "autostart" "("
   method_signature <UNPARSED_EXPRESSION>
   [wrapper_declaration] [tx_declaration_body]
    ")"
   ;

se_declaration:
   naming
   | grouping
   | deps_declaration
   | view_declaration
   | dele_declaration
   | terminate_declaration
   ;

method_signature:
   name  "(" type_list_opt ")"
```

```
   ;

wrapper_declaration:
   "(" simple_name_list ")"
   ;

tx_declaration_body:
   "{" [naming_grouping_autostarts]
   [tx_begin] [tx_commit] [tx_abort]
    "}"
   ;

tx_alias:
   "alias" "(" name_key_exp ")"
   ;

tx_name:
   "name" "(" name_key_exp ")"
   ;

groupAdd:
   "groupAdd" "(" name_key_exp ")"
   ;

groupName:
   "groupName" "(" name_key_exp ")"
   ;

groupAlias:
   "groupAlias" "(" name_key_exp ")"
   ;

name_key_exp:
   simple_name <UNPARSED_EXPRESSION>
   ;

deps_declaration:
   "dep" "(" dep_list ")"
   ;
```

```
dependency:
   simple_name simple_name simple_name
   ;

view_declaration:
   "view" "(" "-"? simple_name simple_name ")"
   ;

dele_declaration:
   "del" "(" simple_name simple_name ")"
   ;

terminate_declaration:
   tx_terminate
   | terminateGroup
   ;

tx_terminate:
   "terminate" "(" simple_name ")"
   ;

terminateGroup:
   "groupTerminate" "(" simple_name ")"
   ;

type_list_opt:

   | type_list
   ;

type:
   name
   | name dims
   ;

dims:
   "[" "]"
   | dims "[" "]"
   ;
```

```
name:
   simple_name
   | qualified_name
   ;

simple_name :
   <IDENTIFIER>
   ;

qualified_name :
   name  "." <IDENTIFIER>
   ;

tx_declaration_list:
   tx_declaration
   | tx_declaration_list ";" tx_declaration
   ;

se_declaration_list:
   se_declaration
   | se_declaration_list ";" se_declaration
   ;

naming_grouping_autostart_list:
   naming_grouping_autostart
   | naming_grouping_autostart_list ";" naming_grouping_autostart
   ;

type_list:
   type
   | type_list "," type
   ;

simple_name_list:
   simple_name
   | simple_name_list "," simple_name
   ;

dep_list:
   dependency
```

```
   |dep_list  "," dependency
   ;
```

C.2 KALA Tokens

The grammar defined above relies on a scanner to return tokens for symbols, such as `< IDENTIFIER >` . This scanner is also written using SmaCC [Ref05], which allows a scanner to be defined using a collection of token specifications, with each specification terminated by a semicolon `;` . Each token is specified by a pair of token name and a regular expression, separated by `:`, where the token name is a valid Smalltalk variable name that is surrounded by `<>`. We do not include the SmaCC syntax for regular expressions here, as it is extensive yet quite straightforward, instead we refer to the SmaCC documentation [Ref05].

Token definitions

```
<IDENTIFIER> :
   [a-zA-Z_] [a-zA-Z0-9_]*;

<UNPARSED_EXPRESSION> :
    \< ([^\\\>] | <ESCAPE_SEQUENCE> | \\ [\>\<] )* \> ;

<ESCAPE_SEQUENCE> :
   \\ ([btnfr\"\'\\] | ([0-3] [0-7]{0,2} | [4-7][0-7]?)) ;
```

Index

Bibliography

[AC95] E. Anwar and S. Chakravarthy. Realizing transaction models: an extensible approach using ECA rules. Technical report, Computer and Information Science and Engineering Department, University of Florida, 1995.

[ACM01] Deepak Alur, John Crupi, and Dan Malks. J2EE patterns. Technical report, Sun Java Center, 2001.

[AFTO89] Abdel Aziz Farrag and M. Tamer Özsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503 – 525, December 1989.

[AM97] Gustavo Alonso and C. Mohan. *Advanced Transaction Models and Architectures*, chapter Workflow Management: The Next Generation of Distributed Processing Tools. Kluwer, 1997.

[Asp05] The AspectJ project, 2005. http://eclipse.org/aspectj/.

[ASSR93] P. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In *Proceedings of the 19th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Dublin*, 1993.

[BDG+94] A. Biliris, S. Dar, N. Gehani, H. V. Jagadish, and K. Ramamritham. Asset: a system for supporting extended transactions. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 44–54. ACM Press, 1994.

[BFB+] Bill Burke, Marc Fleury, Adrian Brock, Claude Hussenet, Kabir Khan, and Marshall Culpepper. JBoss Aspect Oriented Programming. http://www.jboss.org/developers/projects/jboss/aop.

[Blo77] Arthur Bloch. *Murphy's law and other reasons why things go wrong!* Price/Stern/Sloan Publishers, 1977.

[BMDV02] Johan Brichau, Kim Mens, and Kris De Volder. Building composable aspect-specific languages. In *Proc. Int'l Conf. Generative Programming and Component Engineering*, pages 110–127. Springer Verlag, 2002.

[Bon04] Jonas Bonér. What are the key issues for commercial aop use: how does aspectwerkz address them? In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 5–6. ACM Press, 2004.

[BP96] Roger Barga and Calton Pu. Reflection on a legacy transaction processing monitor. In *Proceedings of Reflection '96*, 1996.

[CDK94] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems, Concepts and design*. Addison-Wesley, second edition, 1994.

[CR91] Panos K. Chrysanthis and Krithi Ramamritham. A formalism for extended transaction models. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 103–112, 1991.

[CR92] Panos K. Chrysanthis and Krithi Ramamritham. *Database Transaction Models For Advanced Applications*, chapter ACTA: The SAGA continues, pages 349–397. Morgan Kaufmann, 1992.

[DV99] Kris De Volder. Aspect-oriented logic meta programming. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *Lecture Notes in Computer Science*, pages 250–272. Springer Verlag, 1999.

[Edw99] Jeri Edwards. *3-Tier Client/Server At Work*. Wiley Computer Publishing, 1999.

[ELLR90] A. Elmagarmid, Y. Leu, W. Litwin, and Marek Rusinkiewicz. A multidatabase transaction model for interbase. In *Proceedings of the sixteenth international conference on Very large databases*, pages 507–518, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[Elm92] Ahmed K. Elmagarmid, editor. *Database Transaction Models For Advanced Applications*. Morgan Kaufmann, 1992.

[Fab04a] Johan Fabry. But what if things go wrong? Paper presented in CADS workshop at ECOOP04, 2004.

[Fab04b] Johan Fabry. Transaction management in EJBs: Better separation of concerns with AOP. Paper presented in The Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, 2004.

[FC05] Johan Fabry and Thomas Cleenewerck. Aspect-oriented domain-specific languages for advanced transaction management. In *Proceedings of 7th International Conference on Enterprise Information Systems*, 2005.

[FG04] Johan Fabry and Kris Gybels. Building software with logic and OO symbiosis: an experience report. Proceedings of MPOOL04 workshop at ECOOP 04, 2004.

[FM04] Johan Fabry and Tom Mens. Language-independent detection of object-oriented design patterns. *Elsevier Computer Languages, Systems and Structures*, 30(1-2):21–33, April-July 2004.

[GB03] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *2nd International Conference on Aspect-Oriented Software Development*, 2003.

[GHKM94] Dimitrios Georgakopoulos, Mark F. Hornick, Piotr Krychniak, and Frank Manola. Specification and management of extended transactions in a programmable transaction environment. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 462–473, Washington, DC, USA, 1994. IEEE Computer Society.

[GM83] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186 – 213, June 1983.

[GMDW00] Hector Garcia-Molina, Jeffrey D.Ullman, and Jennifer Widom. *Database System Implementation*. Prentice Hall, 2000.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the ACM SIGMOD Annual Conference on Management of data*, pages 249 – 259, 1987.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing, Concepts and Techniques*. Morgan Kaufmann, 1993.

[Han99] Per Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4), April 1999.

[Her] Alvaro Herrera. Personal communication.

[HS76] J.H Haynes and Adrian Sharp. *Fiat 124 sport automotive repair manual*. Haynes Publishing Group, 1976.

[HVL95] Walter L. Hürsh and Cristina Videira Lopes. Separation of concerns. Technical report, College of Computer Science, Northeastern University, 1995.

[I+97] John Irwin et al. Aspect-oriented programming of sparse matrix code, 1997. Xerox Palo Alto Research Center.

[JK97] Shushil Jajodia and Larry Kershberg, editors. *Advanced Transaction Models and Architectures*. Kluwer, 1997.

[KG02] Jörg Kienzle and Rachid Guerraoui. Aop: Does it make sense? - the case of concurrency and failures. In *Proceedings of ECOOP 2002*. Springer Verlag, 2002.

[KG05] A. Kellens and K. Gybels. Issues in performing and automating the "extract method calls" refactoring. In *Proceedings of Software Engineering Properties of Aspect Languages and Technologies (SPLAT) Workshop at AOSD '05, Chicago, USA*, 2005.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of ECOOP 1997*. Springer Verlag, 1997.

[KPE92] Eva Kühn, Franz Puntigam, and Ahmed K. Elmagaramid. *Advanced Transaction Models and Architectures*, chapter Multidatabase Transaction and Query Processing in Logic. Morgan Kaufmann, 1992.

[KS03] Randi Karlsen and Thomas Strandenæs. Trigger-based compensation in web service environments. In *International Conference on Enterprise Information Systems 2003*, pages 487–490, 2003.

[Lea99] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, second edition, 1999.

[MH01] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, third edition, 2001.

[Mic05] Microsoft Corporation. Microsoft .NET Home Page. http://www.microsoft.com/net, 2005.

[MKL97] Anurag Mendhekar, Gregor Kiczales, and John Lamping. RG: a case-study for aspect-oriented programming, 1997. Xerox Palo Alto Research Center.

[MMR95] Martin Müller, Tobias Müller, and Peter Van Roy. Multi-paradigm programming in Oz. In Donald Smith, Olivier Ridoux, and Peter Van Roy, editors, *Visions for the Future of Logic Programming: Laying the Foundations for a Modern successor of Prolog*, Portland, Oregon, 7 1995. A Workshop in Association with ILPS'95.

[MMW01] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. In *13th Int. Conf. Software Engineering and Knowledge Engineering, Buenos Aires*, pages 136–143. Knowledge Systems Institute, June 2001.

[Mos81] E. B. Moss. Nested transactions: An approach to reliable distributed computing. Technical report, Massachusetts institute of Technology, 1981.

[MRKN92] Peter Muth, Thomas C. Rakow, Wolfgang Klas, and Erich J. Neuhold. *Advanced Transaction Models and Architectures*, chapter A Transaction Model for an Open Publication Environment. Morgan Kaufmann, 1992.

[Obj04] Object Management Group, Inc. The common object request broker: Architecture and specification. http://www.omg.org/technology/documents/formal/corba_iiop.htm, 2004.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Pet77] James L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3), September 1977.

[PKH88] Calton Pu, Gail E. Kaiser, and Norman C. Hutchinson. Split-transactions for open-ended activities. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, pages 26–37, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[Pro01] Marek Prochazka. Advanced transactions in enterprise javabeans. In *EDO '00: Revised Papers from the Second International Workshop on Engineering Distributed Objects*, pages 215–230, London, UK, 2001. Springer-Verlag.

[RC96] Krithi Ramamritham and Panos K. Chrysanthis. *Executive Briefing: Advances in Concurrency Control and Transaction Processing*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[RC03] Awais Rashid and Ruzanna Chitchyan. Persistence as an aspect. In *2nd International Conference on Aspect-Oriented Software Development*. ACM, 2003.

[Ref05] Refactory, inc. The SmaCC parser. http://www.refactory.com/Software/SmaCC/, 2005.

[RSS97] Andreas Reuter, Kerstin Schneider, and Friedemann Schwenkreis. *Advanced Transaction Models and Architectures*, chapter Contracts Revisited. Kluwer, 1997.

[SGM94] Kenneth Salem and Hector Garcia-Molina. Altruistic locking. *ACM Transactions on Database Systems*, 19(1):117 – 165, March 1994.

[SLB02] S'ergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of OOPSLA 02*. ACM, November 2002.

[Sun02] Sun Microsystems. Java naming and directory interface (JNDI). http://java.sun.com/products/jndi/, 2002.

[Sun03a] Sun Microsystems. Java data objects (JDO). http://java.sun.com/products/jdo/, 2003.

[Sun03b] Sun Microsystems. Java remote method invocation (RMI). http://java.sun.com/products/jdk/rmi/, 2003.

[Sun05] Sun Microsystems. Enterprise JavaBeans architecture. http://java.sun.com/products/ejb/docs.html, 2005.

[SVJ03] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29. ACM Press, 2003.

[Szy98] Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[TCLL03] Bruce Tate, Mike Clark, Bob Lee, and Patrick Linskey. *Bitter EJB*. Manning, 2003.

[The] The PostgreSQL Global Development Group. The PostgreSQL database. http://www.postgresql.org/.

[TOHJ99] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.

[tpc01] Transaction processing performance council. http://www.tpc.org, 2001.

[tpc04] TPC benchmark C. Technical report, Transaction Processing Council, http://www.tpc.org/tpcc, 2004.

[VB03] Werner Van Belle. *Creation of an Intelligent Concurrency Adaptor in order to mediate the Differences between Conflicting Concurrency Interfaces*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, 2003.

[VC05] Bart Verheecke and Maria Augustina Cibran. Dynamic aspects in large scale distributed applications, an experience report. In *Proceedings of Software Engineering Properties of Aspect Languages and Technologies (SPLAT) Workshop at AOSD '05, Chicago, USA*, 2005.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[Vin97] Steve Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997.

[VLK97] Cristina Videira Lopes and Gregor Kiczales. D: a language framework for distributed programming, 1997. Xerox Palo Alto Research Center.

[WR92] Helmut Wächter and Andreas Reuter. *Advanced Transaction Models and Architectures*, chapter The ConTract Model. Morgan Kaufmann, 1992.

[WS92] Gerhard Weikum and Hans-J Schek. *Advanced Transaction Models and Architectures*, chapter Concepts and Applications of Multilevel Transactions and Open Nested Transactions. Morgan Kaufmann, 1992.

[WS97] Devashish Worah and Amit Sheth. *Advanced Transaction Models and Architectures*, chapter Transactions in Transactional Workflows. Kluwer, 1997.

[Wuy01] R. Wuyts. *A Logic Meta-Programming Approach to Support Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, January 2001.