# A Family of Domain-Specific Aspect Languages on Top of KALA

Johan Fabry
Vrije Universiteit Brussel, Programming Technology Lab
Pleinlaan 2,
1050 Brussel, Belgium
johan.fabry@vub.ac.be

Theo D'Hondt
Vrije Universiteit Brussel, Programming Technology Lab
Pleinlaan 2,
1050 Brussel, Belgium
tjdhondt@vub.ac.be

## 1. INTRODUCTION

Transactions are a long-standing example of a cross-cutting concern, and therefore it should come as no surprise that previous work has aspectized this concern [7, 9, 10]. We can, however, consider transactions as just one instance of the much broader field of *advanced transaction management* (ATMS) [2, 6]. In this field a multitude of advanced models have been defined. Each model has its own specific properties, and is typically designed to overcome one known shortcoming of classical transactions. For example, the poor performance of long-lived transactions is addressed by the Sagas model [5], which we discuss later.

Our work addresses the field of advanced transaction management, and aims for aspectizing a wide variety of models. To achieve this, we have built an aspect language and weaver for advanced transaction management, called *KALA* [4]. KALA is a *domain-specific aspect language* for ATMS, based on a formalism for advanced transaction management models. This formalism is called *ACTA* [1], and is well-known within the ATMS research community. KALA reifies the ACTA constructs. This allows for the aspectisation of a wide variety of advanced transaction management models.

In this paper we discuss how the use of KALA has lead us to build a family of domain-specific languages (DSLs) on top of KALA. Experience in using KALA has shown us that, indeed, KALA aids greatly in the specification of the use of advanced transaction management models. However, we see that for the use of specific models, we can further simplify the work of the transaction programmer by providing a DSL specific to that ATMS. Instead of covering the field of ATMS, such a model-specific language reifies the concepts of one model, which yields a higher level of abstraction. Furthermore, as different ATMS share common concepts, we can also share these concepts in the different model-specific languages. This sharing of concepts yields a family of domain-specific aspect languages.

*OAL Workshop At AOSD06* March 20, 2006, Bonn
.

## 2. EXAMPLE ATMS

We provide here a short description of two of the best-known ATMS: Nested Transactions and Sagas, as we shall describe the domain-specific languages for these models later in the text.

*Nested transactions* [8] is one of the oldest and easily the most well-known ATMS. It allows for a running transaction $T$ to have a number of child transactions $Tc$. Each $Tc$ is given access to the data used by $T$. This is in contrast to classical transactions, where the data of $T$ is not shared with other transactions. $Tc$ may itself also have a number of children, forming a tree of transactions. When a child transaction $Tc$ commits its data, this data is not written to the database, but instead delegated to its parent $T$, where it becomes part of the data of $T$. If a transaction $Tx$ is the root of a transaction tree, i.e. it has no parent, $Tx$'s data will be committed to the database when $T$ commits. Lastly, if a child transaction $Tc$ aborts, the parent $T$ is unaffected, i.e. $T$ is not required to also abort.

*Sagas* [5] is, next to Nested Transactions, one of the oldest ATMS and also arguably one of the most referenced ATMS in the community. Sagas is tailored towards long-lived transactions. Instead of one long transaction $T$, a saga $S$ splits $T$ into a sequence of sub-transactions $T1$ to $Tn$. Each sub-transaction is a normal classical transaction, and this sequence has to be executed completely before the saga can commit. To abort, or rollback, a running saga $S$, the currently running sub-transaction $Ti$ is aborted, and the work of already committed transactions $T1$ to $Ti-1$ has to be undone, as their results have already been committed to the database. To allow this, the application programmer has to define for each sub-transaction $Ti$ a compensating transaction $Ci$ that performs a semantical compensation action. To undo the work of $T1$ to $Ti-1$, $C1$ to $Ci-1$ are ran by the runtime transaction monitor in inverse sequence, i.e. starting with $Ci-1$.

## 3. KALA

The domain-specific aspect language for ATMS we created is called KALA, which stands for **K**ernel **A**spect **L**anguage for **A**TMS. Code in KALA implements the ATMS aspect for Java applications that use the Enterprise JavaBeans middleware architecture. KALA was designed to reify the concepts of the ACTA formal model for ATMS. In ACTA [1], a transaction can be given a number of different properties, of three possible categories: *dependencies*, *views* and *delegation*. The

main purpose of KALA code is to declare such dependencies, views and delegation for a Java method that corresponds to a transaction. These properties can be declared to apply at begin, commit or abort time of the transaction.

An advanced transaction, e.g. a Saga, usually is built up from multiple classical transactions that are related in some way. In the example, the Saga $S$ comprises the sub-transactions $T1$ to $Tn$. In KALA the basic ACTA building blocks of dependencies, views and delegation, combined with extra elements: *naming*, *grouping* and *termination*, are used to declare the transactional behavior of the different methods (i.e. transactions) of the advanced transaction.

As an example, we show here the KALA code for one step of a saga, corresponding to a Java method with name `methodName` and parameter list `parameterList`, of the class `className`, contained in the package `packageName`. (Code which is emphasized is discussed later)

```
packageName1.className1.methodName1(paramList1) {
   alias (Saga <Thread.currentThread()>);
   alias (CompPrev <""+Saga+"Comp">);
   groupAdd (self <""+Saga+"Step">);
   autostart (methodName1c(paramList1c)
      <actualslist1> (wraplist1) {
         name(self <""+Saga+"Comp">);
         groupAdd(self <""+Saga+"Comp">);
      });
   begin {
      alias (Comp <""+Saga+"Comp">);
      dep(Saga ad self, self wd Saga, Comp bcd self);
   }
   commit {
      alias (Comp <""+Saga+"Comp">);
      dep(CompPrev wcd Comp, Comp cmd Saga,
         Comp bad Saga);
   } }
```

Using the ACTA building blocks permits the aspect programmer to implement a wide variety of ATMS and also allows for the creation of a new ATMS. This is done by either tweaking a known ACTA specification or constructing a complete specification from scratch. We do not further introduce KALA and ACTA in detail here, as this is outside of the scope of this paper. For a more detailed introduction we refer to [4] and [1].

The use of KALA has shown to yield a significant advantage over manually implementing the transaction concern in the equivalent Java code [3, 4]. Not only do we achieve a good separation of concerns, but thanks to the domain-specific nature of KALA, a significant number of implementation concerns are abstracted, yielding much more concise code [3]. For example, we have an implementation of an application using the Sagas model, where the amount of transaction code in Java is 230 lines of code, scattered over multiple methods. The equivalent code in KALA is only 52 lines long and contained in one module.

Further examining such KALA code, however, we have seen that a significant amount of this code is redundant. For example, in Sagas the only changes between the code for the different steps is the code which is emphasized in the listing shown above. Therefore we have a large amount of code duplication within one saga definition. The only exceptions are the elements referring to the actual methods and compensating methods of the saga, i.e. the code emphasized above.

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 the same ATMS is used this 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 construction process implies that the application programmer is required to know the complex technical implementation of the ATMS.

We should avoid the need for an application programmer to write such 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 behavior. 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. Therefore KALA can be used as a common base, i.e. a *kernel*, for other programming languages, which further simplifies specifying the transactional properties of code for a given ATMS, trading-off generality.

## 4. A DSL FAMILY

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, or modify an existing ATMS, we can shield the programmer from the implementation concepts of an ATMS. Programmers that simply use an existing ATMS can work at a higher level of abstraction, using the ATMS as a black box component. This raises the abstraction level of the program to the level of the concepts present in the ATMS.

For example, in the saga example discussed in the previous section, the programmer should only concern himself with how the Sagas behavior is applied to the application and not with the implementation of this behavior. In other words, the programmer should only need to specify the code we emphasized. An example step specification could be as follows:

```
step packageName1.className1.methodName1(paramList1)
      compensate methodName1c(paramList1c)
      <actualslist1> (wraplist1);
```

To achieve such a higher level of abstraction we created 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. At compile time the DSL code is translated to equivalent KALA code. This KALA code is then woven into the base concern.

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 is always programmed

[1]Following the simple rule that the more code is written, the higher chance for errors in that code.

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, of which we discuss three here. 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 [8] and Sagas [5]. For each DSL we first give a brief introduction 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 an informal definition of this translation step. 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, et cetera.

## 5. TRANSACTIONS DSL

The first language we introduce here is the DSL for classical transactions. 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.

For example, consider the method with name `methodName` and 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:

```
trans packageName.className.methodName(paramList);
```

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.

```
packageName.className.methodName(paramList) {
   commit {terminate(self); }
   abort {terminate(self); } }
```

Using the transactions DSL here frees us from one implementation concern, which is termination of transactions.

## 6. NESTED TRANSACTIONS DSL

We built a DSL for nested transactions, which supports two ways to structure the transaction hierarchy. We discuss a specific form of nested transactions where the transaction hierarchy is equal to the method call hierarchy. We also specified the general form, without such equality, but due to lack of space we only discuss the first form here.

In the first form, 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.

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 `extends caller`.

Specifying a root transaction is, in other words, identical to specifying a transaction in the classical transaction DSL. In other words, 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:

```
trans packageNameR.classNameR.methodNameR(paramListR);
trans packageNameC.classNameC.methodNameC(paramListC)
   extends caller;
```

The equivalent KALA code, given below, uses a naming technique based on the identity of the current thread to identify the calling transactional method. The root transaction will register itself in the global naming registry using the current thread, and child transactions look up the parent using current thread as key. These transactions, such as declared for `methodNameC`, save the parent identifier locally and register themselves under the current thread, 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.

```
packageNameR.classNameR.methodNameR(paramListR) {
   name(self <Thread.CurrentThread()>);
   commit {terminate(self); }
   abort {terminate(self); } }

packageNameC.classNameC.methodNameC(paramListC) {
   alias(parent <Thread.CurrentThread()>);
   name(self <Thread.CurrentThread()>);
   begin {
      dep(self wd parent, parent cd self);
      view(self parent);}
   commit {
      del(self parent);
      name(parent <Thread.CurrentThread()>);
      terminate(self); }
   abort {
      name(parent <Thread.CurrentThread()>);
      terminate(self); } }
```

Because of the semantics of naming, dependencies, views and delegation (discussed in detail in [3, 4]), 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.

## 7. SAGA DSL

Considering the concepts present in the Sagas model, we find more concepts present than in the previous ATMS. We identify the following concepts: the saga, the steps and their compensating steps. All three concepts 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 parameters are shared between methods, as defined in [3, 4].

The code below shows an example declaration of a saga as a number of steps. Each has 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 allow for a varying number of steps, as indicated by `[...]` in line 10.

```
saga packageName.className.methodName(paramList) {
   step packageName1.className1.methodName1(paramList1)
      compensate methodName1c(paramList1c)
      params <actualslist1>
      wrap (wraplist1);
   step packageName2.className2.methodName2(paramList2)
      compensate methodName2c(paramList2c)
      params <actualslist2>
      wrap (wraplist2);
[...]
   step packageNameN.classNameN.methodNameN(paramListN);
}
```

The above code contains five different concepts: the saga, the steps, their compensating steps, the parameters for these steps and which of these are wrapped. As this directly corresponds to the concepts of the Sagas model, 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, code in the Saga DSL is as concise as possible.

Regarding the KALA code generated from the above DSL program, we do not include the full code here, due to lack of space. Instead, we refer to section 3 where we have shown the KALA code for one step of a Saga.

Considering the large size and obvious complexity of the KALA code, 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 Saga DSL is as concise as possible and at a higher level of abstraction, which therefore eases implementation.

Revisiting the example Saga implementation of section 3, which requires 52 lines of KALA code, this only requires 10 lines of code in the Saga DSL. We stared with 230 lines of Java code, and have reduced this to 10 lines, i.e. slightly over four percent of its original size.

## 8. CONCLUSION

KALA is a domain-specific aspect language which covers the domain of ATMS. In KALA, the basic building blocks of ACTA are combined to form the specification of the transactional behavior of Java methods. Using the same ATMS, or parts of an ATMS, multiple times in an application results in code duplication. This is because the transactional behavior of these different methods is, in essence, the same.

Because KALA gathers all transactional specifications together and brings them at the level of abstraction of the ACTA model, we can more easily reason about this code than the equivalent Java code. This allowed us to conceive new avenues of abstraction and remove code duplication. If we forgo the ability to create or modify models, we can add an extra layer of abstraction through the use of a DSL per model. The use of such model-specific languages allows us to isolate the aspect programmer from the implementation details of the model. Instead, the programmer now focuses solely on the concepts of the ATMS being used. Such model-specific aspect languages make the aspect programs as concise as possible. 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 reduces the learning curve for a programmer learning a new DSL.

## 9. REFERENCES


[1] P. K. Chrysanthis and K. Ramamritham. A formalism for extended transaction models. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 103–112, 1991.

[2] A. K. Elmagarmid, editor. *Database Transaction Models For Advanced Applications*. Morgan Kaufmann, 1992.

[3] J. Fabry. *Modularizing Advanced Transaction Management - Tackling Tangled Aspect Code*. PhD thesis, Vrije Universiteit Brussel, Departement Informatica, Laboratorium voor Programmeerkunde (PROG), July 2005.

[4] J. Fabry and T. D'Hondt. KALA: Kernel aspect language for advanced transactions. In *Proceedings of the 2006 ACM Symposium on Applied Computing Conference, to appear*, 2006.

[5] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM SIGMOD Annual Conference on Management of data*, pages 249 – 259, 1987.

[6] S. Jajodia and L. Kershberg, editors. *Advanced Transaction Models and Architectures*. Kluwer, 1997.

[7] J. Kienzle and R. Guerraoui. Aop: Does it make sense? - the case of concurrency and failures. In *Proceedings of ECOOP 2002*. Springer Verlag, 2002.

[8] E. B. Moss. Nested transactions: An approach to reliable distributed computing. Technical report, Massachusetts institute of Technology, 1981.

[9] A. Rashid and R. Chitchyan. Persistence as an aspect. In *2nd International Conference on Aspect-Oriented Software Development*. ACM, 2003.

[10] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of OOPSLA 02*. ACM, 2002.