Vrije Universiteit Brussel

Faculty of Science
Department of Computer Science
and Applied Computer Science

Coordination in Volatile Networks

Graduation thesis submitted in partial fulfillment of the
requirements for the degree of Lincense in Computer Science

Christophe Scholliers and Eline Philips

Promotor:	Prof. Dr. Theo D'Hondt
Advisors:	Stijn Mostinckx
	Charlotte Herzeel

JUNE 2007

Acknowledgements

This document would not be what it is today without the outstanding support of various people.

First of all we would like to thank professor Theo D'Hondt for promoting this thesis and giving us the opportunity to conduct our apprenticeship at the Programming Technology Lab.

Furthermore, special thanks go to our advisors Stijn Mostinckx and Charlotte Herzeel for all the help they have given us. Not only for bringing up the subject of this thesis, but also for the countless hours they have spent in discussions, proofreading and helping us in any possible way at any possible time. We own them big gratitude for their willing to help and listen to us, even in times when they were overwhelmed by work. Without their support and help, this thesis would never have become what it is today. We would also like to thank them for giving us the opportunity to go and accompany us to CALA Junior Days and CoDaMoS workshop. Moreover, we thank them for sharing this memorable year with us.

We would also like to thank dr. Wolfgang De Meuter for providing us with the necessary equipment for performing our experiments. Without these computers, we couldn't perform demonstrations on our presentations.

Moreover, we would like to thank dr. Maja D'Hondt for letting us present our work in her busy schedule and for the interesting discussion afterwards.

Thanks also to all members of the Programming Technology Lab, for their comments and discussions at the various meetings, as well as for their support, help and surrounding us with a first-class working atmosphere.

We also thank all professors and assistants who have guided us through our education and have shared their enthusiasm and knowledge with us.

Our last thank goes to our parents which have given us the opportunity to retrieve an excellent education at the *Vrije Universiteit Brussel*. Furthermore, for their support, care and understanding when we were stressed and needed someone to talk to. Thanks also to our friends and family for giving us time to talk and laugh and forget our stress moments.

We also thank one another for the endless discussions, support, belief and laughter. Together we achieved more than we could have realised alone.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

As stated by Moore [43], the number of transistors that can be integrated into a single chip doubles every two year. Moreover, the complexity of processor chips is proportional to the number of transistors embedded into a single chip. Making the transistors smaller is not only beneficial for making more complex and faster chips, it also lowers the power consumption of these chips. In the early years of computing, a computer could easily fill a whole room whereas today cellphones with a multitude of the power of these early computers match in the palm of your hand. Moreover, with the use of a variety of standard technologies like infrared, bluetooth and wifi we can connect these miniaturised devices in a mobile ad hoc network.

With the rise of these technologies, the dream of Weiser [66] where persons are surrounded by a cloud of small devices cooperating with eachother and adapting themselves to their context is getting closer. These small connected computing units, dubbed ambient devices, have characteristics which are significantly different from the conventional devices. One of these characteristics is that connections cannot be assumed stable, this stems from the fact that these devices are mobile and can leave a certain area in any moment in time. The development of context-aware applications coping with frequently occurring disconnections is a subject of various research projects.

Adapting applications to change their behaviour according to their context *can* be done by making use of conventional languages, however as shown by Turing every language proven to be Turing complete is as powerful as any other Turing complete language. We do not argue this fact but put forward the expressive power of the language and maintainability of the written code. A sterling example where expressiveness is even chosen over computational power is shown in the world of databases where the language SQL is used to describe the data needed from a database. In this dissertation we propose the Fact Space Model, a new programming model sculpted to offer fine grained control over the effects of disconnections in a mobile environment. A proof of concept of the Fact Space Model is the experimental programming language Crime.

1.1 Motivation

This dissertation advocates the use of a distributed reasoning engine in combination with a truth maintenance system in order to ease the development of context-aware applications. Context-aware applications reap the benefits of having a view on the environment by providing additional or smarter behaviour. An example application which provides additional behaviour by exploiting its view on the environment could provide the customers of a super market with the promotions of products residing in its environment. Smarter behaviour is provided by a mobile phone application which switches its profile to silent when entering a room where a meeting is taking place.

The implementations of context-aware applications suffer from the fact that the current context-aware frameworks do not provide the necessary means to react on *all* relevant changes in the environment, in particular when context information becomes unavailable. In a mobile environment where context information is exchanged by making use of a MANET network [14], the context information a device perceives is related to its connectivity with co-located devices. In the supermarket example, when the customer leaves the store, the provided context involving promotions should be withdrawn as this information is no longer useful for the user. This retraction of information must be performed each time devices disconnect.

Most existing coordination languages are unable to react on a combination of events occurring in the contextual environment. Although there exist languages that use an inference engine for reasoning about multiple events, they do not provide the necessary hooks for reacting on disconnections which is important in a mobile setting. Our solution proposes an inference engine at its core which enables reasoning about multiple events in a distributed environment. Furthermore, we incorporate a truth maintenance system in order to guarantee that only valid consequences are derived from the perceived environment.

One of the characteristics of a mobile environment is that devices must be able to discover one another without the need of a centralised coordination manager. This is because a centralised manager conflicts with the autonomy criteria of Ambient Intelligence [34] where each device must be able to operate independently. Therefore we propose a decentralised architecture for the Fact Space Model.

In order to deal with the issues of current mobile coordination languages, we propose the Fact Space Model. This model provides a logic coordination language for reasoning about fluctuating context in mobile ad hoc networks by the incorporation of a distributed reasoning engine in combination with a truth maintenance system to ease the development of context-aware applications.

1.2 Proof of Concept

To validate the proposed Fact Space Model, we developed an experimental coordination language, dubbed Crime, which is a direct translation of the principles put forward by the Fact Space Model. Crime provides the necessary support to exchange context information among different devices residing in a mobile environment where disconnections occur frequently.

Various experiments involving the implementation of context-aware applications have been conducted, such as the implementation of context-aware jukebox and chat applications. We observed that in order to implement these applications, it is often necessary to query past context. However, due to the fact that past context facts are automatically removed as new information becomes available, it was up to the Crime programmer himself to write plumbing code for recording and querying past context. Therefore, we have extended Crime to Crime Time, where temporal operators are added as primitives, to alleviate the programmer from ad hoc management of the context history.

Special care has been taken to explore and adapt optimisation to cope with the characteristics of a mobile environment. The incorporation of these optimisation into Crime resulted into a language were the intermittent disconnection are handled fast and efficiently.

1.3 Document Overview

As our proposed solution tackles the problems at hand by combining and extending the principles from existing coordination models, distributed reasoning systems and context-aware frameworks, the first part of this thesis is dedicated to each of these related fields.

The next chapter presents tuple spaces and federated tuple spaces, on which we based the Fact Space Model. These models provide both space and time uncoupling by making use of an associative shared memory. However, they offer no immediate support for reacting on a combinations of events. Our solution extends the tuple space model with a declarative language facilitating the need to react on multiple events in the environment. Declarative programming and logic programming languages are discussed in the next chapter. Chapter 3 discusses reasoning engines and truth maintenance systems, which are combined in the Fact Space Model enabling easily development of context-aware applications. Subsequently, chapter 4 describes existing context-aware frameworks and compares them to our experimental coordination language Crime.

Our solution, the Fact Space Model, is presented in the subsequent chapter. We first explain the different parts constituting the model, and next we discuss a concrete instantiation of this model, dubbed Crime. Thereafter we illustrate the use of Crime for implementing context-aware applications. This coordination model provides the necessary hooks to overcome the problems at hand with existing approaches, as for instance reacting on disconnections. Chapter 6 highlights the implementation details of this experimental language by supplying the necessary pseudo code for algorithms, and UML diagrams for the most relevant parts of our implementation.

The two subsequent chapters represent extensions to this basic implementation. A first extension proposes the introduction of temporal operators to enable reasoning about past context information. A second extensions implements optimisations to the basic Rete algorithm and presents the results of conducted experiments for benchmarking these optimisations techniques.

A last chapter concludes this dissertation by recapitulating the goal and results of this thesis and presenting our contributions and future work.

Chapter 2

Tuple Spaces

This dissertation advocates the use of a distributed reasoning engine in combination with a truth maintenance system in order to ease the development of context-aware applications. Before elaborating on these issues in chapter 5, we describe the historical background and outline the technical principles of the Tuple Space Model, the communication model used in the remainder of this thesis.

Communication between processes and applications is a very mundane feature of today's multitasking operating systems. One of the simplest communication mechanisms uses files where communication is performed by letting one process writing to, and another reading from them. Sockets are another commonly used communication mechanism allowing processes to communicate by explicitly naming the process it wants to transfer data to. Classic problems which arise when multiple processes communicate are deadlocks (when a process awaits resources that are locked by other processes) and starvation (when a process is blocked from a resource necessary to finish its task). A more complex communication model, COMMUNICATING SEQUENTIAL PROCESSES CSP [32] was invented during the development of the OCCAM language [33], which provides a formalism to express the communication among processes. With this formalism, independent processes which communicate solely by message passing can be modelled by the use of a few primitives and its operators. The classic problems of deadlock and starvation are supplemented with a set of additional issues when moving to a mobile distributed setting.

When expanding the communication among processes to a distributed environment, the notion of time is something that can not be taken for granted. Specialised algorithms like *Lamport timestamps* [40] must be used to allow the ordering of events in a distributed setting. Further issues arise when the communication channels are no longer reliable.

All communication mechanisms can be grouped into four models, namely *monitor*, *message passing*, *remote operations*, and *generative communication*. The first three categories are already discussed, the latter one is introduced by Gelernter [26]. The Tuple Space Model falls in this last category, where communication is realised by the use of a shared distributed associative memory in order to coordinate interprocess communication.

Coordination languages are concerned with the communication between the different entities in a system. These can be situated in a distributed environment or hosted on the same device. Coordination languages implement a communication model in contrast to computation languages which implement a computational model.

In most low level main stream languages like C [38] and C++ [63], communication is not embedded in the language itself, rather special purpose libraries are used to orchestrate communication which are forced to adopt the paradigm of the language. This is in contrast with coordination languages where the paradigm itself is dedicated to communication. LINDA was the first coordination language, it uses the model of generative communication for coordination, and needs a host language for its computation.

The remainder of this chapter is organised as follows, the next section provides an overview of the and how it differentiates itself from other coordination models. In section 2.2 the coordination language LINDA is described by means of some example applications, and some problems with the primitive operations of the language are discussed. Upon describing how the Tuple Space Model can cope with a distributed environment, we present extensions to the Tuple Space Model to operate in a mobile environment. First we discuss the LIME language which extends the model with transiently shared tuple spaces, and subsequently we present TOTA, which adds a special kind of tuple to the original model in order to orchestrate global coordination.

2.1 Tuple Space Model

The basic building block of the *Tuple Space Model* is a *tuple*, which contains the information the system can work with. These tuples all reside in a globally accessible memory called a *tuple space*. For example the tuple `<''Hello World", 42>` is a tuple of the kind `''Hello World''` having one integer field with value 42. A field can be data or executable code. A distinction is made between tuples which have solely data fields, and tuples which have at least one field containing executable code. The former kind of tuples are called *passive tuples* whereas the latter are called *active tuples*. The difference between both kinds is that a passive tuple can be stored directly into the tuple space. In contrast an active tuple contains code which must be evaluated by a special operator to a *passive value*. When all active fields have been processed to passive values, a passive tuple is constructed and inserted to the tuple space. All tuples reside in a tuple space which is accessible by a number of concurrent processes, possibly distributed over different devices. Processes can publish and withdraw facts from the tuple space which acts as a shared distributed associative memory. Communication between concurrent processes is handled solely through the tuple space. In order to let two concurrent processes communicate, one process publishes tuples to the tuple space and the other process withdraws them. Communication in the Tuple Space Model has some very distinct proper-

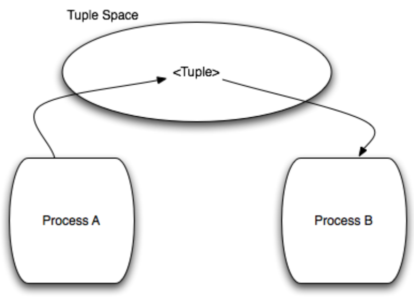


Figure 2.1: Tuple spaces: communication

ties that make it different from other communication models. Tuples published to the tuple space are available for *all* processes which results in an uncoupling in space as the receiver process does not need to be know beforehand. Time uncoupling arises from the fact that tuples can outlive the process that generated it. Tuple spaces therefore allow communication between processes that are not active at the same time. These properties are in contrast with communication models such as message passing where the receiver process must be named explicitly and interprocess communication is synchronous. Because of the synchronous nature of such message passing models, it is impossible to conduct communication between time uncoupled processes. Note that although some communication models allow communication with an unknown group of receivers, by means of broadcasting, they do not provide a time uncoupling. In the rest of this section the four basic operations *in*, *out*, *eval*, and *read* present are presented, and the distinct properties of the communication model are highlighted.

2.1.1 Pattern Matching and Unification

A tuple space can be seen as a data memory with specialised primitives in order to access the data residing in it. The primitives which are discussed in the section 2.1.2 are based upon the notion of pattern matching and unification. Here we give a short introduction to both the concepts pattern matching and unification.

Pattern matching is used to test whether a given data fits some pattern [1]. Considering an example datum, `has_attribute(Merlin, hat)`, we may say that this data matches the pattern `has_attribute(?a, ?b)`, with both `?a` and `?b` variables. The datum also fits with some other patterns like for example `has_attribute(?a, hat)` and `has_attribute(Merlin, ?a)`, where `Merlin` and `hat` are constants. An example pattern that does not match `has_attribute(Merlin, hat)` is: `has_attribute(?a, Merlin)`, because the second argument should be the constant hat instead of the constant Merlin.

The process responsible for verifying whether a datum matches a pattern is called the *pattern matcher*. The pattern matcher has three inputs, a datum, a pattern and a frame and returns the updated frame as its result. A frame is a datastructure consisting of a mapping between variable names and values, this mapping is called *the bindings*. This frame is augmented by the pattern matcher by adding new bindings during the matching. The algorithm verifies if the given datum matches the specified pattern in a way that it stays consistent with the variable bindings of the frame. When a inconsistency is found, the pattern matcher reports that the pattern doesn't match.

Unification is a special kind of pattern matching, where both the pattern and the datum can contain variables [1]. A *unifier* takes two patterns and a frame as its inputs and tries to a list of bindings that makes the patterns equal. So for example, the patterns `has_attribute(Merlin, ?a)` and `has_attribute(?b, hat)` can be unified and the resulting frame is extended with a binding for the variable `?a` to the value hat and for the second variable, `?b` the value equals Merlin. For the example to unify the pattern `has_attribute(Merlin, ?a)` and `has_attribute(?a, hat)` the unification fails as the variable `?a` can not be the constant Merlin and hat at the same time. In contradiction with pattern matching, unification of two patterns doesn't need to result in bindings for each variable occurring in one of those patterns.

2.1.2 Operations

Here we present the four commonly used operators in tuple space based languages.

The out statement

$$out(Name, P_2, ..., P_n). \tag{2.1}$$

The parameters starting from P_2 may all be actual or formal parameters. When we assume that P_2 up to P_n are all actual parameters, the **out** statement returns directly and publishes the tuple `<Name, P`$_2$`, ..., P`$_n$`>` to the tuple space. In the case that there is at least one formal parameter contained in the **out** statement, this is called *inverse structural naming*.

Consider a chat channel where users can send messages to other persons that are available and willing to chat. Alice wants to receive a message, which is simulated by the first line in the code excerpt of listing 2.1. When another user publishes a tuple `out(Alice, "Hello")`, Alice receives this message. Users are also able to send their messages randomly to all users that want to receive messages, by using the inverse structural naming mechanism, and can spam other users by publishing a tuple `out(?nickName :string, "Hello")`.

```
in(Alice, ?message :string).
out(Alice, "Hello").
out(?nickName :string, "Hello").
```

Listing 2.1: Example using inverse structural naming

The eval statement

$$eval(Name, P_1(A_2), ..., P_{n-1}(A_n)). \tag{2.2}$$

The parameters starting from A_2 must all be actual parameters. These actual parameters A_2 up to A_n are the arguments for the processes P_1 up to P_n. The **eval** statement forks a thread in parallel with the main thread, and this new thread waits until all processes have returned a passive value, whereas the main thread isn't waiting and keeps on running. When all passive values have been collected, a passive tuple is constructed and published in the tuple space.

The in statement

$$in(Name, P_2, ..., P_n). \tag{2.3}$$

The parameters starting from P_2 may all be actual or formal parameters. When we assume that P_2 up to P_n are all formal parameters, the tuple space is searched for a tuple matching the `Name` and structure specified by the **in** statement. When there's at least one formal parameter in the **in** statement, *inverse structural naming* occurs. Then the formal parameters are bound with the actual parameters of the found tuple, and the execution of the program continues. Upon completion of this operation the matching tuple has been removed atomically from the tuple space. When at the time that the **in** statement is executed no matching tuple is present, the execution is suspended until a matching tuple is published in the tuple space.

The use of formal parameters allows the programmer to access a group of tuples with the same signature. The usage of actual parameters in the **in** statement allows the programmer to give a more specific representation for the tuple. This way a topology between certain facts is introduced, and a specific-general relation among tuples is established. All actual parameters together with the tuple name are said to form the *structural name* of the tuple. In the following **in** statement the structural name of the requested tuple is Appointment-"New York"-Alice.

$$in(Appointment, ?date : integer, \text{“}NewYork\text{“}, Alice). \tag{2.4}$$

As can be seen, this structural name is a combination of only the actual parameters of the tuple which introduces a family of tuples. This structural naming can be compared with unification known from logic computer languages such as Prolog [62]. From another point of view the structural name can also be related to the select operation on a relational database.

Formal parameters can also be used in the **out** statement. When such a statement is published to the tuple space, it can be read by an **in** statement with an actual parameter to bound with the formal parameter.

$$out(Name, i : integer). \tag{2.5}$$

$$in(Name, 5). \tag{2.6}$$

The read statement

$$read(Name, P_2, ..., P_n). \tag{2.7}$$

The **read** statement is identical to the **in** statement except that the matched tuple is not removed from the tuple space. Publishing and withdrawal of tuples need to be executed atomically, otherwise duplications could arise from two processes that execute an **in** operation. Indeed, when such a operation is executed simultaneously only one process receives the tuple from the tuple space, other processes conducting the operations at the same time receive an other matching tuple or block until a matching tuple is published.

2.1.3 Properties

Space Uncoupling

Communication in the Tuple Space Model is performed using **out**, which does not designate a receiving process. A published tuple can be read by any process having access to the tuple space, allowing interprocess communication between processes that do not know each other. The process that executed the **out** statement does not know which process performed the **in** operation on the tuple it published.

In most mainstream communication mechanisms, data can be received from a process that is not known in advance. The Tuple Space Model extends these models to allow sending data to an unknown processes. This allows writing programs that do not specify the receiving process nor the sending process beforehand. This property can offer important advantages.

Consider for instance a distributed application where a computation is divided over the available hosts in the network. There is one *master host* which divides the computation and sends each host a part of the total computation. The hosts which perform the actual computation are dubbed *worker hosts* as they perform the actual work, whereas the master host only delegates. In order to make optimal use of the computational infrastructure, the master host should ensure that new computations are made available to the hosts which are finished. Implementing this in the Tuple Space Model is trivial as the sending of a new computation is just the publishing of a tuple. As we can not assume that all worker hosts perform their computation at equal speed some hosts finish their part of the computation before other worker hosts. Therefore the work is divided in more computations than there are hosts available, by this mechanism faster hosts can request more computations than slower hosts. When a host is finished with its computation, it can retrieve a new computation by performing an **in** operation. When there are still requested computations left in the tuple space, the worker host retracts this instantly form the tuple space. When there are no computations left, the worker host blocks until the master host publishes new requests in the tuple space. Applying this coordination scheme results in a computation which is balanced over the available hosts. Because of the space uncoupling worker hosts do not need to contact the master host directly to complete their task or to request a new computation.

Time Uncoupling

A tuple published by the **out** statement is withdrawn when the process that created the tuple terminates. The removal – on termination of the process – of a tuple can be avoided if the process explicitly specifies not to do so. This allows the tuple to outlive the program, and allows communication between programs which are not active at the same time. Note that time uncoupled processes are nothing new and are already provided in most operating systems using files. However, in the Tuple Space Model time uncoupling is embedded in the core of the language and no specialised language features are needed to have this property.

Sharing Variables

Because publishing and withdrawing tuples from the tuple space is ensured to be atomic, no special process is needed to shield shared variables. A shared variable can be easily represented by a tuple in the tuple space. Note that this is a powerful mechanism because no extra threads are needed to shield access to the variable. The distributed nature of the Tuple Space Model ensures that a published tuple is accessible by all processes. As shown later, the atomicity of the Tuple Space Model makes it possible to implement distributed semaphores the same way as shared variables.

2.1.4 Tuple Distribution

The main difficulties of concrete tuple space implementations lies in the distribution of tuples among network nodes. Various distribution models have been proposed [42] and the most common ones are explained here.

Centralisation

The centralisation model entrusts a designated host to regulate all operations on a tuple space. This host contains the whole tuple space and is accessible by all clients in the network. The centralisation model's main advantage is its ease of implementation. As the tuple space is located on one host, assuring atomicity of the operations is easier than with a distributed datastructure. The disadvantages of a centralised architecture relate to the system's scalability and reliability. When the number of nodes increases, the load for the hosts increases, making it a bottleneck for the system. Moreover, when stability and reliability are important a pure centralised approach is not sufficient as it introduces a single point of failure. Finally, as all communication must go through the designated host, the network resources are not used optimally.

Partitioning

In the partitioning strategy multiple hosts are designated to be the server for a specific subset of tuples with a common characteristic (e.g. for a particular set of names). The publishing of a tuple in the tuple space, is then routed to the precise server depending on a specific property of the tuple at hand. The choice of the partitioning function is crucial in such systems, since naive partitions of the tuple space can lead to an unbalanced distribution of the tuples over the available hosts. More complex systems dynamically adapt the partitioning function to the applications running on the hosts. Beside *data decentralisation*, partitioning also has the benefit that operations on the tuple space can be conducted concurrently as long as the tuples reside on different hosts. Partitioning does not solve the problems regarding reliability that centralisation introduces, as each partition uses a centralised architecture, as described in the previous paragraph.

Full Replication

Full replication makes replica's of the entire tuple space on multiple designated servers. The publishing and withdrawal of tuples requires the involvement of all hosts. As there are duplicate tuples residing on multiple servers, it's important to keep the whole tuple space consistent. Ensuring such consistency requires costly *locking protocols* which block the whole tuple space for a single operation. Some support for *fault-tolerance* is inherently supported by making use of full replicas. Using replication enables performing consistency checks at regular intervals. A disadvantage of full replication is the increased network communication for multicasting destructive operations to all clients in the network.

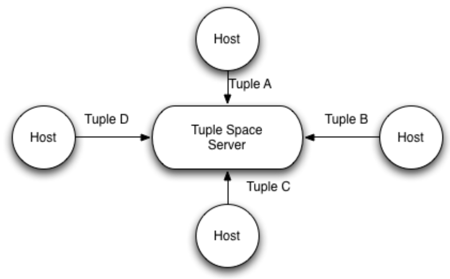


Figure 2.2: Centralised model of tuple space

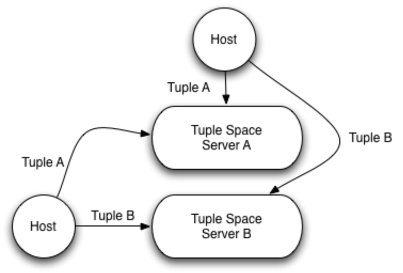


Figure 2.3: Partitioning model of tuple space

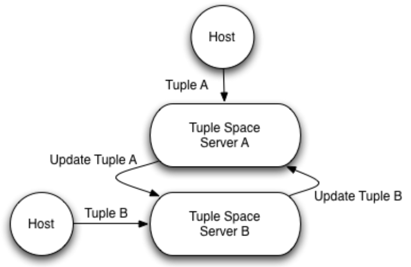


Figure 2.4: Full replication model

2.2 Linda

Linda is the first language that implemented the Tuple Space Model providing the full power of asynchronous interprocess communication, which is uncoupled in both time and space. Linda is not a conventional programming language, but rather it is a *coordination language*. This implies that Linda has as its domain the interaction between different processes rather than the computation performed by these processes. Linda can be embedded into any computational language (it has been embedded in a range of imperative languages like C [46], but also in functional programming languages like ML [58]). Such a complete language is capable of describing both the computation performed by a process as well as how it should coordinate activity with other processes. This section gives an overview of the Linda coordination language by specifying small examples which are typical for the Tuple Space Model.

2.2.1 Basic Communication

Code Example

As a concrete example of the use of the Tuple Space Model, consider the Linda code in listing 2.2. The process `encode` is started and performs a conversion of the string ``‘‘Active’’`` to the string ``‘‘Passive’’``. The **in** operation blocks until a `Message` tuple has become available. This process first performs an **out** operation which causes the publication of the tuple ``‘‘Hello World’’, 1`` in the tuple space. When the **in** operation is executed, the formal parameter `?number` is bound to the value `1`. The evaluation of ``eval(Message, encode(‘‘Active’’))`` results in the insertion of an active tuple. Afterwards, the **in** operation reads the passive tuple from the tuple state, and this causes the variable binding of `?state` to ``‘‘Passive’’``. The last line of the code prints the value of the formal parameter `?state` which equals ``‘‘Passive’’``; So, ``‘‘State = Passive’’`` is printed. The steps of this example together with the tuple space state is depicted in figure 2.5.

```
int main() {
    out("Hello World", 1);
    int number;
    in("Hello World", ?number);

    process String encode(String x) {return "Passive";}
    eval(Message, encode("Active"));
    String state;
    in(Message, ?state);
    printf("State = %s", state);
}
```

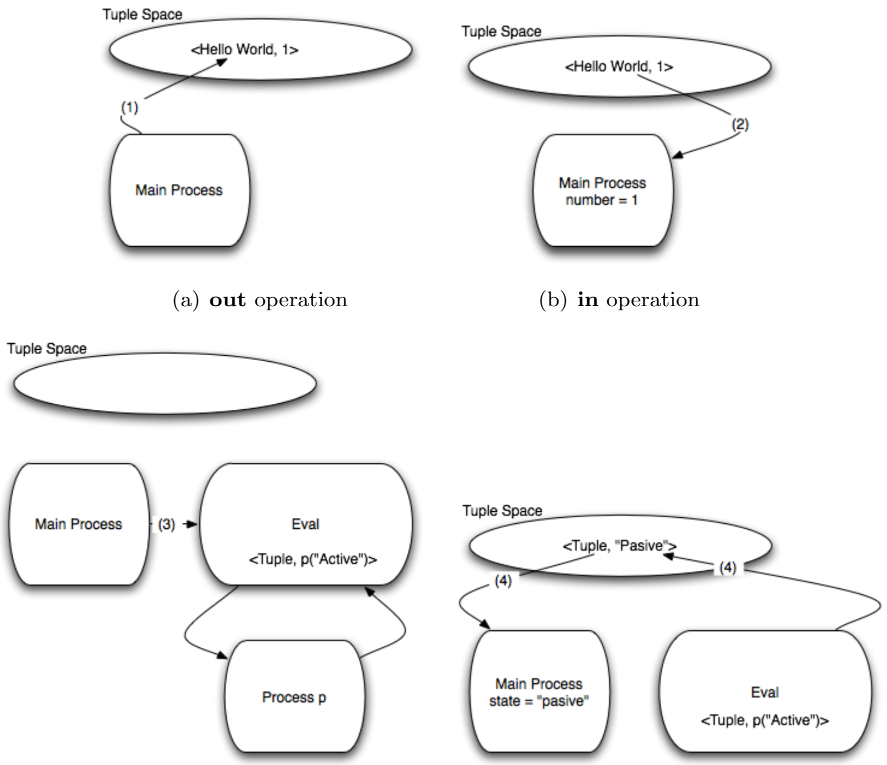
Listing 2.2: LINDA: *hello world* example

The insertions of active tuples is not blocked but a separate process is started for each template field that needs to be calculated. When all processes have returned a value, a passive tuple (recall that a passive tuple does not contain executable code at any of its fields) is constructed based on these return values and inserted.

Semaphores

LINDA is a coordination language because its basic primitives allow expressing concurrency control patterns to govern the interaction between different processes. A well-known concurrency control pattern is the *semaphore* invented by E. Dijkstra and first implemented in THE OPERATING SYSTEM [20]. It is a protected variable used to restrict access to a common datastructure. Semaphores can be used in general with N processes that may concurrently access the same datastructure. In the case where the access is limited to only one process at a time, we can use a *binary semaphore*. In the general case, when more then one process may access the resource, it's called a *counting semaphore*. Semaphores are commonly used to coordinate synchronisation in concurrent programs. The operations on a semaphore are **P** and **V**.

The semantics of the **V** operation is to increase the semaphore by one which is realised as one atomic operation. The **P** operator blocks access to the semaphore



(a) **out** operation (b) **in** operation

(c) evaluation (d) **in** operation

Figure 2.5: *Hello world* example: tuple space depicted on different timesteps

`P(Semaphore s) {`
`        Suspend until s > 0`
`}`
	`V(Semaphore s) {`
	`    s += 1;`
	`}`

Table 2.1: Operations on semaphores

s, until it has a value higher than zero. In LINDA every tuple having only one element is equivalent to a binary semaphore. To understand this, recall that the operations on the tuple space are guaranteed to be executed atomically. Therefore the **V** operation can be simulated by the insertion of a tuple in the tuple space. The **P** operation can be implemented by an **in** operation.

Note that the tuple space properties give the semaphores implemented in LINDA a time and space independent nature. Space uncoupling stems from the fact that the semaphore is accessible from different processes possibly residing

void P(Template semaphore) {	**void** V(Template semaphore) {
in(semaphore);	out(semaphore);
}	}

Table 2.2: Semaphores implementation in Linda

on different devices on the network, making the semaphore distributed. Time uncoupling is achieved because the semaphore can possibly outlive the process that created it. Hence Linda inherently incorporates semaphores.

Synchronous Communication

Another coordination pattern is the *rendez-vous pattern* which allows processes to perform synchronous communication. The **out** operation does not wait for the tuple to be withdrawn from the tuple space. This is not a hard restriction as we can easily simulate synchronous communication by a limited set of operations [26].

```
outs(t) =     in(A)  , out(t),   in(B)
ins(t)  =    out(A)  ,  in(t), out(B)
```

Note that this transformation can only be used by two processes conducting synchronous communication at the same time. When multiple processes want to communicate synchronously, new auxiliary symbols must be introduced; Namely one for every pair of communicating processes. Once again, the benefits of Linda communication is preserved: no prior knowledge of the data provider is needed by the data consumer, only the auxiliary symbols that are used.

Multiple Tuple Spaces

Over the years several extensions to the basic Linda language where proposed. A first extension to the original Linda model was introduced due to a lack of separation of concerns. Since tuple spaces are a shared datastructure in which all processes can read and write, information can not be hidden or scoped. Therefore, multiple tuple spaces were proposed. These tuple spaces can be made by the processes itself, and provide an abstraction layer to group tuples with the same properties. This extension led to a concrete implementation MTS Linda [8].

Multiple tuple spaces were influenced by the need to hide information. Because all tuples reside in the same tuple spaces, all processes in the system can access the same information. Non-related processes using the same tuple space can easily influence eachother, either by accident or maliciously. Also in large software systems where there are multiple users that run their programs concurrently, it is not acceptable that a user can see or modify other users' information. Beside of the coordination abstraction, a tuple space actually represents a datastructure which holds a multi-set of tuples and the Linda operations can be seen as accessors and mutators. With this view in mind, the natural way to address the scoping problem is to the tuple space as a datastructure which can be created, assigned, and passed around like any other datastructure. Hence making it a first class data type. However, there is no clear semantics on the various problems which arise when making tuple spaces first class. One of the problems that arises is to provide the semantics for a process which requests a tuple from a deleted or retracted tuple space. [8]

```
int main() {
  TupleSpace myts;
  myts.out("Test", 42);
  myts.rd("Test", ?i);
  eval(42, f(137));
  rd(42, ?i);
  out("Ts", myts);
  in("Ts", ?myts);
  TupleSpace A, B;
  A = B;
  if(A == B) {..}
}
```

Listing 2.3: Operations on multiple tuple spaces [8]

The code in listing 2.3 performs some operations on multiple tuple spaces. The first line of code in the main procedure creates a new local tuple space `myts` where tuples can be published and withdrawn form. When the **out** operation is performed on the local tuple space, by executing the second line in the procedure, the tuple `(‘‘Test’’, 42)` is inserted in that local tuple space. Executing a **read** operation on that tuple space results in a variable binding for the formal parameter `?i` to the value `42`. The evaluation `eval(42, f(137))` results in publishing an active tuple in the tuple space. The **read** operation that is performed after that evaluation reads a resulting passive tuple. The line of code beneath the **read** operation results in copying the local tuple space `myts` in the context tuple space, so the **in** operation performed afterwards retrieves the tuple space and binds the variable `?myts` to that tuple space. The execution of the line of code `A = B` results in assigning a copy of tuple space `B` to tuple space `A`. The if-test performed afterwards compares those two tuple spaces. Note that outside the main procedure, the local tuple space `myts` leaves its scope and is deleted.

2.2.2 Summary

Linda has been designed for a distributed environment where there is access to a globally shared tuple space. It provides primitives adopted from the tuple space model for the access to this memory. These primitives can be used to simulate the standard coordination patterns like semaphores and the rendezvous pattern. Moreover these primitives provide additional properties like time and space uncoupling. A first problem with Linda concerns the inability to hide information, this has been solved by the introduction of multiple tuple spaces. More problems with the basic set of primitives provided by Linda, are discussed in the next sections.

2.3 The Multiple rd Problem

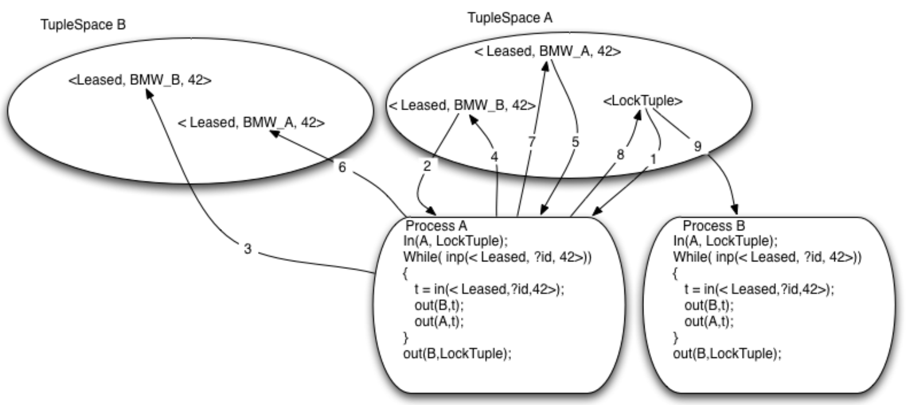
The LINDA model is not able to cope with the *multiple rd problem.* In order to solve this problem a new primitive operation **copy-collect** was introduced by A. Rowstron and A. Wood [52]. In this section we show the problem by means of an illustrative example and show why the new operator is a solution for the **rd** problem.
*"A multiple **rd** is defined as an operation where two or more processes are required to concurrently, and non destructively read one or more tuples from a tuple space which match the same template, where there are at least two or more tuples that match the template, and at least two of the processes can be satisfied by the same tuples"* A.Rowstron [52].

This section handles two rd-problems that arise by using the basic LINDA primitives. The claim is that the basic primitives of LINDA are not sufficiently expressive when trying to solve this **rd** problem.

To understand this, consider the tuple space with two or more tuples representing the leasing of cars `<Leased, ?CarId, 42>` to the customer with id 42. There are a lot of processes which need this information, and one of the operations of these processes is to retrieve all the the carId's which are rented to the customer with id 42. At first one may be tempted to solve this by just invoking multiple **rd**, but this isn't a correct approach because a second **rd** operation blocks the entire program. This forms the first problem that can be avoided by using *polling.* A second problem arises when there are multiple tuples that match. The definition of **rd** is to non-deterministicly retrieve a tuple from the tuple space, and thus invoking multiple **rd** does not allow the programmer to retrieve all tuples residing in the tuple space.

Semaphore Solution

As the operations on the tuple space are guaranteed to be atomic, a general solution to overcome the problem is to use a special *semaphore tuple* which is determined beforehand by all involving processes. When a process wants to access this particular kind of tuple – tuples with a type equal to `Leased` – it first has to read the semaphore tuple. Thereafter all the wanted tuples – tuples with type `Leased` and customer id equal to 42 – must be destructively removed to ensure that all tuples are retrieved. Upon completing the operation the reinsertion of the removed tuples and the locked tuple. Those removed tuples were kept for performing the actual action on them. This is a general solution because although the tuples are destructively removed from the tuple space, the other processes which may need these tuples are not aware of this removal as they cannot acquire the semaphore tuple and thus cannot read the tuples during the operation. When the operation is finished, all tuples are restored. This technique applied to the car rental company is depicted in figure 2.6. The numbers in the picture denote the time steps involved: the two processes execute concurrently, but process A retrieves the semaphore tuple and therefore executes first. The figure illustrates the steps up until processes B retrieves the semaphore.



```
Process A
In(A, LockTuple);
While( inp(< Leased, ?id, 42>))
{
   t = in(< Leased,?id,42>);
   out(B,t);
   out(A,t);
}
out(B,LockTuple);
```

```
Process B
In(A, LockTuple);
While( inp(< Leased, ?id, 42>))
{
   t = in(< Leased,?id,42>);
   out(B,t);
   out(A,t);
}
out(B,LockTuple);
```

Figure 2.6: Multiple **rd** problem: semaphore solution

The solution presented above is not an acceptable solution for a number of reasons. First of all: all processes need to follow the protocol and must know about the lock tuple. When there are unrelated processes working on the same tuple space, they may not follow the protocol. This scenario is very likely when extending an existing program where the tuples where accessed concurrently. Either way, if one process does not follow the protocol either by error or maliciously, the other processes can no longer assume they have access to all possible tuples.

As a second reason why utilising this approach is not acceptable in the context of our dissertation, is the loss of the benefits of the Tuple Space Model, which is done by serialising access to the tuple space. This is clearly not an acceptable property in a distributed setting. Moreover, in the car company example, we see that parallel access is not harmful at all because of the non-destructive nature of the **rd** operation.

Copy-collect

In order to solve the **rd** problem, a new primitive **copy-collect** is introduced. The general form of the primitive is:

```
x = copy−collect(ts_one,ts_two, template)
```

Copy-collect copies all tuples that match with the template from tuples space `ts_one` into tuple space `ts_two`. The return value of the primitive denotes the number of tuples that are copied to the destination tuple space. The interaction of the **copy-collect** primitive with the standard primitives is governed by the following rules:

When the **copy-collect** primitive is invoked concurrently with the **out** or **in** primitive on the first tuple space, and the tuple of that operation matches the template of the **copy-collect**, the normal behaviour of the operations **out** and

in is no longer guaranteed. Although, the other matching tuples are duplicated to the destination tuple space.

Copy-collect is not allowed to *live-lock*: it always stops and returns a value, this needs to be ensured by the implementer. As a **copy-collect** can take more time than an **out** operation, multiple **out** operations can be issued during the execution of the **copy-collect**. Because of these consequences, a naive implementation could live-lock. Another requirement is that the **copy-collect** primitive relays on *out-ordering*, this means that when one process invokes two **out** statements, the second can not be published in the tuple space before the first one.

Using copy-collect

The **copy-collect** primitive is a simple yet elegant solution to the multiple **rd** problem. The primitive allows multiple processes to concurrently duplicate a group of tuples to a destination tuple space. The process that copied the tuples is informed about the number of tuples that were duplicated. Knowing this number allows the process to retrieve all the tuples it needs by destructively reading them from the destination tuple space.

To recapitulate, applying the **copy-collect** primitive to the car leasing company results in the following processes.

```
copy-collect(T1, T2, <Leased, ?carId, 42>).
```

This is a good solution because during the **copy-collect**, other processes can still access the tuple space. Moreover it exhibits the same expressiveness as the other operations using the pattern matching facilities of Linda.

2.3.1 Asynchronous Tuple Space Access

When there are multiple matching tuples in the tuple space, the **in** and **rd** operations pick only one on a non-deterministic way. Some dialects of Linda allow all matching tuples to be retrieved in one atomic operation. These operations are referred to as **bulk** operations. These bulk operations have a non-blocking nature, and allow asynchronous access to the tuple space whereas the Linda primitives allow synchronous access. Hence, the Bonita [51] primitives were introduced for allowing this asynchronous access. These primitives provide a functional set of the Linda primitives and through their asynchronous access allowing more efficient programs to be written as multiple requests on the tuple space can be performed concurrently.

Bonita Primitives

rqid = dispatch(ts, tuple | [template, destructive | nondestructive])
The dispatch primitive replaces all the Linda primitives to insert or withdraw tuples from the tuple space. The tuple space to be used is indicated by the first parameter `ts`. The second parameter can be either a tuple or a template; When a tuple is given as second parameter, this tuple is placed into the tuple space

`ts`. When the second parameter is a template, the dispatch primitive retrieves a tuple from the tuple space. When retrieving a tuple, the last parameter indicates whether the tuple should be removed. The primitive is non-blocking and returns an identifier which can be used to retrieve the matched tuple.

rqid = dispatch_bulk(ts_one, ts_two, [template, destructive | nondestructive]) This primitive requests the movement of tuples from one tuple space `ts_one` to the tuplespace `ts_two` matching the template. The last parameter is again used to indicate whether the matched tuples should be removed from `ts_one`. Like with the **copy-collect** primitive, described in section 2.3, the number of moved tuples depends on the number of destructive operations undertaken during the execution of the **dispatch_bulk** operation. This primitive is non-blocking and the `rqid` returned can be used to obtain the number of tuples that have been moved by the operation.

arrived(rqid) This primitive checks that the result of the operation with identifier `rqid` is available. This can either be a tuple or a number indicating the quantity of moved tuples. The primitive is non-blocking and returns a boolean representing the availability of the result. When invalid identifiers are given to the arrived primitive, it returns false.

obtain(rqid) This primitive blocks until the result of the operation with identifier `rqid` is available.

Bonita Linda

The Linda primitives can easily be implemented with the Bonita primitives by forcing immediate synchronisation using the operation **obtain**. The code below shows how the **rd** operation can be implemented using the Bonita primitives.

```
int id, carId;
id = dispatch(ts_one, Leased, ?carId, 42, nondestructive);
obtain(id);
```

In order to implement the **in** operation it is sufficient to change non-destructive into destructive in the code above.

The **copy-collect** primitive can be implemented also easily with the Bonita primitives, as shown in the code below.

```
int id, carId;
id = dispatch(ts_one, ts_two, Leased, ?carId, 42, nondestructive);
obtain(id);
```

Performance

In the previous section it has been shown that the Bonita primitives are functionally equivalent to the Linda primitives. Now we exemplify how the Bonita primitives can be used to increase performance by parallelising computation and communication. There is no equivalent operation in LINDA to achieve this behaviour. The LINDA version of a communication and computation is shown below, as the **in** operation blocks the process, the communication and the computation is done sequentially.

```
int carId;
in(ts_one, Leased, ?carId, 42);
compute();
```

The optimised version of the above LINDA code snippet is shown below. Here the computed procedure is performed in parallel with the retrieval

```
int id, carId;
id = dispatch(ts_one, Leased, ?carId, 42, destructive);
compute();
obtain(id);
```

2.4 Lime

LIME [48] was the first coordination language based on the Tuple Space Model that adapted it to operate in a dynamically changing network. The time and space uncoupling properties of the LINDA model align well with a mobile environment. This maps directly to a mobile environment where the network topology is constantly changing. Furthermore tuple spaces provide a natural way to reason about the context perceived by a mobile device. However the LINDA tuple space needs to be globally accessible for all entities in the network. In a mobile environment, maintaining a globally accessible tuple space is hardly possible.

LIME breaks down the LINDA global tuple space into a hierarchy of tuple spaces as shown in figure 2.7. Every process has its own tuple space which contains the tuples it wants to share with other processes. The union of all tuple spaces which reside on the same host, are grouped together in a *Host Level Tuple Space*. These *Host Level Tuple Spaces* are again merged together in a so called *Federated Tuple Space* as depicted in figure 2.7.

This *Federated Tuple Space* can be compared with the LINDA tuple space, though the content of this tuple space is dynamically changed according to the connectivity of the hosts, as well as on the arrival of new processes on those hosts.

In LINDA, the tuple space coordinates the data that is associated with the process communication, whereas LIME uses the tuple space to provide context information depending on connectivity [48].

The smallest tuple space entity which is associated with a single process, is referred to as the *interface tuple space*, and it provides the access to the context of that mobile device. Clearly, breaking down the tuple space into a hierarchy provides a powerful abstraction, the process needs not to be concerned with the dynamically changing network topology which is reflected through changes in the tuple space.

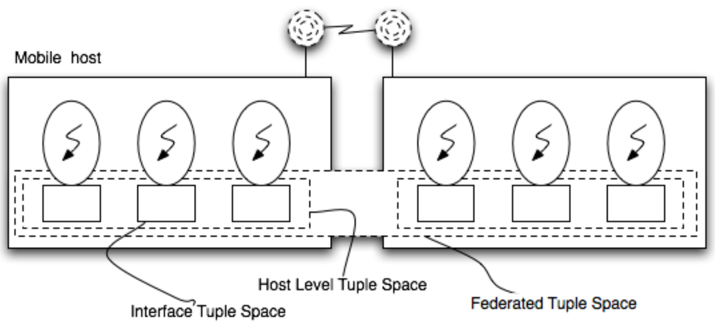


Figure 2.7: Lime: federated tuple space [48]

A Lime group consists of a group of devices within one another's range. All tuples of a Lime group are transiently shared, and the tuple space perceived by a mobile device through its interface tuple space is the conjunction of all interface tuple spaces in that group. This way, the interface tuple space acts like a view on the context of all members of the group.

Lime tuple spaces are named, and a process can access multiple tuple spaces by specifying the designated name. A special tuple space called *LimeSystem* is made accessible by the middleware which can be queried for special context changes like the availability of host and the available tuple spaces itself.

2.4.1 Transiently Sharing

Because Lime operates in a mobile environment, no particular hosts can be preferred to store a global tuple space. Hence the notion of a persistent tuple space like the one in Linda can not be introduced for the ever changing environment. Therefore Lime introduces the transiently shared tuple space, which imposes restrictions on the sharing mechanism depending on the connectivity. Because the sharing of tuples depends on the connectivity of the mobile hosts, some difficulties arise which are addressed by the use of specialised **out** operations as shown in the rest of this section.

2.4.2 Operations

Consider the example where process A wants to communicate with process B. Initially these processes are co-located, and process A publishes a tuple `t` for further processing by process B. However, before process B gets the chance to retrieve the tuple from the transiently shared tuple space, process A gets out of reach. Due to the transiently properties, the tuple `t` is no longer accessible by process B. This process has been depicted in figure 2.8. To overcome this unwanted removal, Lime introduces specialised primitives which deal with location. These operators are explained in the remainder of section.

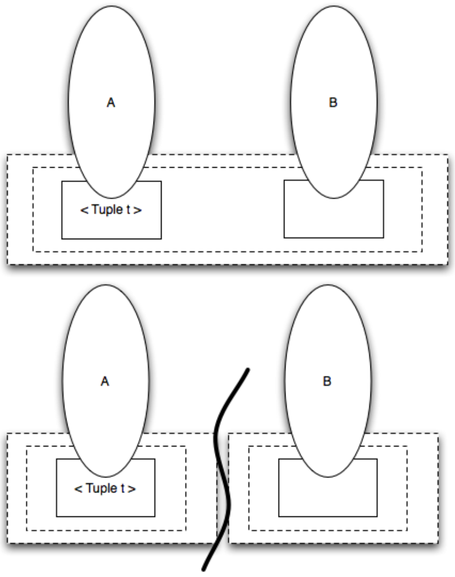


Figure 2.8: Lime: transiently sharing

The out[λ](t) statement

Lime extends the conventional **out** operation with a destination. All tuples are implicitly annotated with a current and destination location. The **out** operation has an optional parameter to explicitly specify the destination location. Publishing a tuple with a destination location involves two steps: when the destination is available, the tuple migrates immediately. Otherwise the tuple stays in the current tuple space and is migrated when the destination location is available. Such a tuple which awaits the arrival of its destination host, is said to be a *misplaced tuple.*

The rd[ω, λ](t) and in[ω, λ](t) statement

The **read** operation resembles the **in** operation performed on a tuple. Reading a tuple from the tuple space can be annotated with both a current ω and a destination λ location. This allows the process to inspect the *misplaced tuples* currently residing in the tuple space. The **in** operation conducts the same behaviour but this operation is also able to let the processes reason about *misplaced tuples* currently residing in the tuple space.

2.4.3 Lime Reactions

Adapting mobile software according to changing context takes up an important part of most mobile applications. It might seem that this can be achieved easily in LINDA by waiting for an **in** statement which announces a context change. However this solution gets cumbersome when there are a lot of context changes to react on. Furthermore, the blocking semantics of **in** imply that in order to react on different context changes a different thread should be started for each possible context change. Therefore LIME has introduced *reactions*, which execute a predefined piece of code when a specified tuple is inserted into the tuple space. When the LIME reactions are activated, they execute a piece of code.

Reactions are split up into weak and strong reactions: *strong reactions* are executed atomically by the LIME engine. This implies that the reaction is executed immediately after a matching tuple was inserted. Being able to react atomically with the insertion of a tuple clearly puts restrictions on the reacting code that has to be executed, as it blocks the entire tuple space. Therefore, the scope of strong reactions is restricted to that of the host tuple space and not the whole federated tuple space. *Weak reactions* do not have this limitation, however they are not guaranteed to be executed atomically. To avoid deadlocks, weak and strong reactions are unable to use LIME's blocking operations.

2.5 Tota

Like LIME, TOTA [41] is also based on tuples, however in this model coordination is not achieved by a tuple space distributed among the mobile hosts in the network. TOTA provides the facilities for tuples themselves to hop from host to host using a migration policy. These tuples do not belong to a specific host but are injected by a host in the network and flood the network according to a pattern specified by the injected tuple. So, TOTA tuples can be viewed as a distributed datastructure representing some kind of distributed contextual information.

The network topology of a group of TOTA nodes is organised as a peer-to-peer network where every node has access to a limited number of co-located hosts. Every host can store tuples and provides the necessary support to let the tuples propagate to other connected hosts. A tuple can no longer be viewed as merely data, it is in fact a mobile program that hops from host to host. TOTA defines a tuple with content C and *propagation rule* P, $\mathbf{T=(C,P)}$. The content is a list of fields, and coincides with the tuples from LINDA. The propagation rule P describes whether a tuple can be propagated and has the opportunity to transform the tuple. The latter ability introduces a very powerful mechanism as the propagation rule is able to maintain a state while flooding through the network.

By letting the propagation rules change the tuple information, the passing of tuples is changed from merely making copies of the tuples to a distributed computation by passing the TOTA tuple around. Because of the propagation rules, the distribution of tuples is no longer restricted to the physical network topology but can be extended to any virtual network topology supported by the physical one. For example, limiting the propagation of tuples to a certain distance from the host which published the tuple, can easily be achieved by computing the distance to the original host every time the tuple is migrated. The propagation stops when the tuple's distance is too far from its originating host.

TOTA not only supports active distribution of the tuples, but also dynamic adaptation. As the underlying network topology is changed, these changes are propagated to tuples and tuples can be automatically repropagated taking into account the changed network content. Reconfigurations of the network, like the addition of a host to the network are propagated by checking the propagation rules of the already stored tuples and the repropagation of to tuples to this newly added host in the network may be triggered depending on the propagation rules. Using the same mechanism, the distribution of tuples adapts it according to movements of a host through the network.

Communication in TOTA is thus reduced to *injecting* tuples to the network and detecting tuples in the local tuple space. Applications written in TOTA are able to access the local tuple space, publish tuples with a content and a propagation rule, and can be notified about changes in the context. The *injection* of a TOTA tuple to the network, can be compared with a raindrop falling into a pool, thus inducing a global change of the surface. Similarly the view of the context of a process can be changed as a result of a local injection by another device.

2.5.1 Architecture

TOTA supports coordination among mobile devices, each running the TOTA middleware. Each mobile device keeps a list of reachable hosts which adapts to reflect changes in the network topology. This may resemble the connectivity in a MANET network [14] but this is not necessarily the case. This constant detection of addressable hosts in the system is something that is built into the core of the TOTA system. While the precise definition of addressability may differ from network to network, the TOTA middleware must be able to reflect on these changes.

Another responsibility of the middleware is the storage and propagation of the tuples in the network. Beside propagating, it must also update these tuples as the network topology changes. In order to accomplish this task, every tuple must be uniquely identified, as the content of a tuple can change while it is propagated, this can't be used as identifier. Therefore, the tuples are tagged with a unique id that is generated by the middleware and is not accessible at the application level. This identification tag is composed of a sequence number reflecting the amount of tuples already injected by a host, and the id of the host itself (its ip address in a closed network environment).

The TOTA middleware can be split up into three main parts: the *API*, the

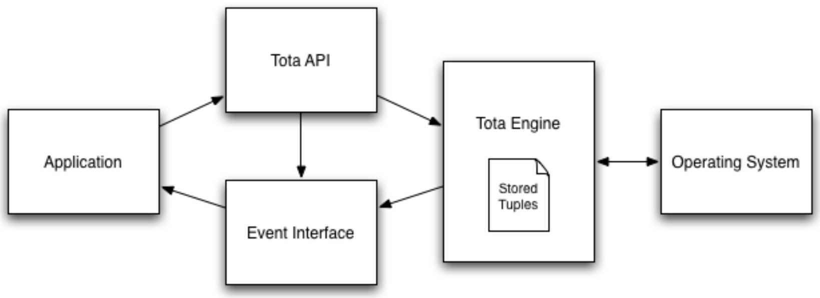


Figure 2.9: TOTA architecture

event interface and the *engine*. The API is a layer between the engine and the applications using the middleware. It specifies how tuples must be published and allows the middleware to query the local tuple space. The event interface allows the registration of *event handlers* in the tuple space. Since everything can be represented by a tuple, every event can be modelled as the insertion of a tuple, the event interface provides the registration and unregistration of event handlers on the local tuple space. To conclude the engine itself performs the propagation of the tuples through the network.

2.5.2 The Tota API

The underlying principles of the TOTA model could be implemented in any language. Here we present the basic operations provided by the TOTA API by a concrete implementation in JAVA. The insertion of a tuple can be achieved through the *inject* method provided by the TOTA API. After the injection the tuple starts propagating through the network according to its propagation rule. The construction of a tuple and how to define its propagation rule is explained later. The local tuple space can be queried by the `read` and `delete` methods. `Read` returns all the tuples in an `ArrayList` which matches the provided template. `Delete` returns a similar list but also removes them from the local tuple space. These operations can be compared with the **bulk** operations provided in some LINDA implementations.

```
public void inject (Tuple tuple)
public ArrayList read (Tuple template)
public ArrayList delete (Tuple template)
```

As the TOTA middleware has an event system, the API provides two methods to register and unregister event handlers. `Subscribe` installs an *event handler* which is triggered upon the arrival of a tuple in the local tuple space, which matches the provided template. `Unsubscribe` unregisters all event handlers which where registered with a template matching the template provided as an argument.

```
public void subscribe (Tuple template, String reaction)
public void unsubscribe (Tuple template)
```

The methods provided by the TOTA middleware presented all depend on the representation of the tuple itself. Because the tuples are not merely data as in LINDA and LIME, more attention is given to this representation and an abstract class of a model tuple is provided by the API. This abstract class already implements a great part of the default behaviour for the tuple propagation. The propagation strategy of this abstract class is a breath first, expanding *ring propagation* [41]. In addition to the propagation strategy, four abstract methods must be implemented to control the content updates, and the tuple behaviour. These abstract methods are used in the propagation strategy as shown in the code in listing 2.4.

```
if(decidePropagation()) {
        doAction();
        updatedTuple = changeTupleContent();
        if(decideStore()) {
                totam.store(updatedTuple);
        }
}
```

Listing 2.4: TOTA: `decidePropagation` method [41]

The `decidePropagation` method provides a way to limit the scope of the propagating tuple. At the one extreme when `decidePropagation` always returns true, the tuple is sent to all hosts in the network, in a breath first way, as specified by the default propagation strategy. At the other extreme, when the `decidePropagation` always returns false, the tuple is not propagated at all. As one can imagine, this method can be used to implement scoping control, like propagation to nearby hosts. The `doAction` provides the tuple the full power of the TOTA API at the host it is propagated to, so it can read, delete, and inject tuples. By this method the TOTA API can be extended with for instance global operators that do a network wide deletion of a certain tuple. Such functionality is not provided by the TOTA API itself, but can easily be implemented by letting the `decidePropagation` always return true, and implement the deletion of the tuple in the `doAction`.

The method `changeTupleContent` allows the programmer to change the tuples content before it gets stored or is propagated to other host in the network. This mechanism is used to let the tuple dynamically adapt as it's being propagated through the network. For example, when the content of the tuple contains a counter then the `changeTupleContent` method could decrease it by one every time the tuple is propagated. Then the `decidePropagation` method can be used to check this counter in order to decide to let the tuple propagate or not. This way the tuple itself can decide to only propagate only a certain distance from the host it was injected from. To conclude the `decideStore` decides whether the tuple should be stored in the local tuple space or not. This method allows the programmer to implement tuples that are just passed along without being stored.

2.6 Conclusion

This chapter gave an overview of the Tuple Space Model and some of the most important languages based upon this model. The Tuple Space Model described in section 2.1 is a coordination mechanism with some very appealing characteristics to coordinate applications in a distributed environment. The Tuple Space Model provides both time and space uncoupling by making use of an associative shared memory. This memory can be implemented centralised or partitioned over the available hosts in the network as shown in section 2.1.4. Subsequently we have presented LINDA, the first coordination language. This language which is based upon the tuple space model has been explained by means of some typical examples. Moreover we have shown that the basic primitives of LINDA can easily simulate a distributed semaphore. In section 2.3 the multiple **rd** problem which is inherently to the tuple space model has been identified and the need to extend the basic primitives of LINDA with the **copy-collect** primitive has been shown.

In section 2.4 we moved from the distributed environment to the mobile environment. A first coordination language for the mobile environment discussed is LIME, which extends the LINDA tuple space with transiently shared tuple spaces. The Fact Space Model that forms the basis for the CRIME language, is based upon this model. The use of transiently shared tuple spaces enables mobile applications to share tuples depending on their connectivity. Subsequent we have explored TOTA which extends LIME with a special kind of tuple which can be seen as a small computing unit. As this special tuple is propagated through the network it can provide global coordination over a network of mobile applications. However, reacting on a combination of tuples is not directly supported by the LIME or TOTA coordination languages. One possible mechanisms is to extend the coordination languages with a declarative language, this approach is also followed by CRIME. The use of a declarative programming language to reason about context is shown in the next chapter.

Chapter 3

Reasoning Engines and Truth Maintenance Systems

This dissertation advocates the use of a distributed reasoning engine in combination with a truth maintenance system in order to ease the development of context-aware applications. In the previous chapter we have introduced tuple spaces in general and the LIME-distribution model which underlies CRIME, the distributed reasoning engine proposed in chapter 5. In this chapter an overview is provided of existing reasoning engines and truth maintenance systems which are the most relevant to our work.

Reasoning engines allow the development of programs which reason about knowledge represented by rules and facts. Truth maintenance systems ensure that when the knowledge available to the reasoner can change non-monotonically, the reasoning engine retracts no longer justified conclusions. Before diving into the specific concepts relating to reasoning engines and truth maintenance systems, this chapter presents a general introduction to logic programming to establish the necessary terminology.

This chapter also discusses the two most important reasoning strategies backward and forward chaining: we argument the use of a forward-chained reasoning engine over a backward-chained one in a mobile environment. Such a reasoning engine is also dubbed a *production system*. Subsequently, the first large scale production system language OPS5 is briefly explained. Section 3.3 then elaborates on the RETE MATCH ALGORITHM – which was first explained in OPS5 – and which significantly increases the performance of production systems.

Having explained the historical background on forward reasoning engines, we present an overview of more recent approaches which have guided design of CRIME. The first one is JESS, which is a reasoning engine/production system based on the RETE algorithm embedded in JAVA. Two following sections explain two distributed reasoning engines, namely DJESS and UBIES. DJESS is a distributed version of JESS, enabling the language to share contextual data among different inference engines and reacting on this shared knowledge. The second distributed reasoning we discuss is ubiES which uses a reasoning engine to automatically detect context information.

A last part of this chapter presents truth maintenance systems as the Fact Space Model we present in chapter 5, uses a justification-based truth maintenance system in order to optimise the operations performed in the Rete network incorporated in that proposed model.

3.1 Logic Programming

The logic programming paradigm allows the programmer to specify the results which should be computed instead of how it is to be computed. In this paradigm, one can simply specify a problem domain in terms of rules and facts which are then used by the reasoning engine. This section highlights the principles of declarative programming as well as the two most known reasoning strategies, namely backward chaining and forward chaining.

3.1.1 Facts and Rules

As we have said in the introduction of this section, a logic program consists of facts and rules. Facts are statements known to be true, e.g.. *"Alice in entertainment room"*. Such facts are stored in a knowledge base, can be viewed as a datastructure keeping a consistent model of the world. Informing a logic programming language of a fact, is performed by asserting into the knowledge base: the fact is said to be *asserted* to the knowledge base. Removing a fact from the knowledge base is called the *retraction* of a fact.

Consider a knowledge base consisting of only one fact *"Alice in entertainment room"*. The knowledge base can be queried: when we ask the question *"Alice in entertainment room ?"*, we receive *true* as an answer because the knowledge base is populated with the fact *"Alice in entertainment room"*. Another query, *"Alice in conference room"* results in *false* because the knowledge base does not contain the fact *"Alice in conference room"*. Note that the answer returned is independent of the actual location, the knowledge base is only a representation of the real world and thus could be wrong.

The latter query illustrates another property of a logic programming language, which is that they often assume that if a fact is not known to be true, it should be assumed false. This is related to the *closed world assumption* known from database systems [59]. As a consequence, when a query of the form `NOT` $\texttt{fact}_x$ is posed, it succeeds unless the fact $\texttt{fact}_x$ is present in the model of the world. This behaviour is called *negation as failure* [23].

Rules, also called productions, express a relationship between different facts. For example the rule `If (''In entertainment Room'')` $\wedge$ `(''Weekend'') Then (''Profile is Loud'')` expresses that when both facts *"In entertainment Room"* and *"Weekend"* are known to be true, it must also be true that *"Profile is Loud"*.

The examples presented thus far are quite limited since they represent concepts which inherently express relations (e.g. the location of a person or the profile of a cellphone) as simple text strings. A much more expressive formalism, predicate logic, can be used to reason about these relations as first class entities and thus derive for instance all persons present in the same location. Such queries would be impossible to express in propositional logic.

For example consider the rule to retrieve all persons in the kitchen. A rule for querying all these persons uses the facts `location(?person, ?room)`. This rule, in Prolog notation, is shown in listing 3.1.

```
inKitchen(Person) :- location(Person, kitchen).
```

Listing 3.1: Prolog: all persons in the kitchen

3.1.2 Backward Chaining

Backward chaining, tries to find available data that supports the goals or hypothesis. Languages using backward chaining typically don't use the Lisp-like rule-syntax but use a head-tail notation instead as is shown in listing 3.2; The head of the clause represent the *then*-part of the rule whereas the *if*-part is represented by the body of that clause. The most well-known logic programming language which used a backward chaining reasoning engine is Prolog [62],

```
head :- body.
```

Listing 3.2: Head-tail notation for rules

The backward chaining algorithm finds those clauses whose head part is satisfied.

In order to illustrate this, consider the rule from the previous section rewritten in head-tail notation.

```
is_wizard(?name) :-
        has_attribute(?name, magic wand),
        has_attribute(?name, hat),
        has_attribute(?name, rabbit),
        artist_name(?name).
```

Listing 3.3: Prolog: *wizards* rule

Suppose the knowledge base contains the same three facts as in the example above:

```
has_attribute(Merlin, hat)
has_attribute(Merlin, rabbit)
has_attribute(Merlin, magic wand)
```

Listing 3.4: Prolog: knowledge base

The set of goals, contains all facts which must be proven to be true. When a new goal `is_wizard(Merlin)` is added to the set of goals, the head of the clause that was given above unifies with this goal, so all the conditions are added to the set of goals. The new goal set now contains the following queries:

```
has_attribute(Merlin, hat)
has_attribute(Merlin, rabbit)
has_attribute(Merlin, magic wand)
artist_name(Merlin)
```

The first three queries in the goal set are satisfied because the knowledge base contains three facts that unify with one of these queries. So, the only goal is to find a fact that unifies with `artist_name(Merlin)`. When this fact is added to the knowledge base, this goal is satisfied as well, the goal set becomes empty and the query `is_wizard(Merlin)` is satisfied. A well-known programming language that uses a resolution strategy based upon backward chaining is PROLOG [62]. Backward chaining algorithms are said to be *goal-driven*: given a query, the algorithm tries to solve it by finding a head that is unified with this query and resolving the sentences in the corresponding body.

3.1.3 Forward Chaining

Forward chaining is one of the two frequently used reasoning strategies, the other being backward chaining which is explained in the previous section. A forward chaining reasoner starts from the available data, and applies all applicable rules to derive new data until no more rules can be applied. A well-known example of a language which uses a forward chaining algorithm is CLIPS [27]. It provides a LISP-like syntax whose general structure is shown in listing 3.5.

```
(name_of_the_production
        conditions
        =>
        actions)
```

Listing 3.5: CLIPS: general rule

Forward chaining starts from known facts, kept in the knowledge base, and triggers those rules whose prerequisites are satisfied. The triggering of such a rule causes the execution of the conclusions of that rule, possibly resulting in modifications to the knowledge base. The algorithm continues on searching for such prerequisites of rules that are met until no more rules can be fired. As an example consider the *wizards*rule in listing 3.6. This rule has four preconditions and one action. The preconditions specify that a person with a certain artist name – `?name` is a variable – which has a `magic wand`, a `hat` and a `rabbit` should be considered a wizard. The *wizards* rule adds a new fact `is_wizard` to the knowledge base when all its prerequisites are met.

```
(wizards
        has_attribute(?name, magic wand),
        has_attribute(?name, hat),
        has_attribute(?name, rabbit),
        artist_name(?name)
        =>
        is_wizard(?name))
```

Listing 3.6: CLIPS: *wizards* rule

Consider that we have asserted the following facts in the knowledge base:

```
has_attribute(Merlin, hat)
has_attribute(Merlin, rabbit)
has_attribute(Merlin, magic wand)
```

Listing 3.7: CLIPS: knowledge base

Since the condition `artist_name(Merlin)` is not present in the knowledge base, the rule cannot be triggered. However, when a new fact `artist_name(Merlin)` is added to the knowledge base, all conditions of the rule are satisfied and the fact `is_wizard(Merlin)` inserted in the knowledge base. This may result in the triggering of additional rules which have the fact as a prerequisite.

The knowledge base reaches a fixed point when no new inferences are possible, because all sentences that could be concluded (part of the then-part of the rule) are already contained in the knowledge base. Forward chaining is called a *data-driven* technique since it does not attempt to derive a set of goals but reacts on changes to the data in the knowledge base [53].

3.1.4 Conclusion

We have explained the two most commonly used derivation strategies used by reasoning engines. The use of a backward-chained engine is preferred when the user has a specific need to query the database in order to extract information out of it, e.g. getting all persons residing in the kitchen. When the user does not have a specific goal, the use of a forward-chained engine is preferred. As forward chaining is data-driven the user does not need to specify a query in order to start the searching process. Forward-chained reasoning engines were traditionally used for *expert systems* which reason about high level domain knowledge much in the same way as we propose to reason about context. A more elaborate discussion between backward chaining and forward chaining in the context of this dissertation is given in chapter 5.

3.2 OPS5

OPS5 is a production system language designed for building expert systems. The language was introduced by Charles L. Forgy [35]. Problems are described in a production system by making use of a set of unordered productions. Similar to the rules in a declarative language, productions can be regarded as *if-then* statements. A production system contains a state of the world it is modelling by making use of a working memory, similar to the knowledge base of a logic programming language. This working memory contains working memory elements, which can be regarded as facts. All these elements together form the current state.

An example of a *production definition* is shown in listing 3.8. This production qualifies somebody as a wizard, namely when he has a `magic_wand`, a `hat` and a `rabbit`, besides those attributes he should also have an `artist name`. Unlike in Prolog, OPS5 supports named attributes in their facts, this name is proceeded by ^. In order to ensure that all these attributes belong to the same person a variable <`name`> is used. The action to perform when a wizard is found, is the addition of an element to the working memory which specifies that a certain person is a wizard.

```
(p is_wizard
        (has_attribute ^name <name> ^attribute magic_wand)
        (has_attribute ^name <name> ^attribute hat)
        (has_attribute ^name <name> ^attribute rabbit)
        (artist_name ^name <name>)
        -->
        (make fact ^name <name> ^is_wizard true))
```

Listing 3.8: OPS5: production definition of *wizards* rule

The *if*-part of a production, the part before the arrow, contains several *prerequisites.* In the example production of listing 3.8, there are four prerequisites. The first three preconditions specify the attributes a wizard must have, the fourth specifies that a wizard should have an artist name. A precondition is satisfied when it can be unified with a working memory element. When all the preconditions of a production are satisfied by the current state (the set of all working memory elements) the rule is triggered. The *consequences* of a production, the part after the arrow, are then executed which is called *firing* the production. In the example rule there is one action in the consequences of the rule, which adds a fact to the working memory.

The algorithm for deriving a solution for the problem described by the production system uses three operations. The first operation is *match*: which finds all productions whose preconditions can be unified with a working memory element: we say that those productions are triggered. The set of all triggered productions is called the *conflict set.* The second operation is to apply *conflict resolution* on the conflict set. The *conflict resolution strategy* that is used, specifies the order in which matched productions are fired. A simple example of conflict resolution is to fire the rules in alphabetic order. Firing the production is done by the *act* operation, which is the third operation. This operation performs the actual execution of the actions in the consequences of the production.

The production's consequences consists of one of the following actions: *make*, *remove* or *modify.* The make action instantiates and adds a new working memory element to the working memory, whereas the action remove leads to the removal of the appropriate element in the working memory. The modify action can be simulated by first removing a working memory element followed by the addition of an element to the working memory.

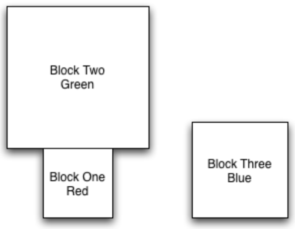
The working memory can be considered as a database that contains the data were the productions react upon. The data in the working memory is contained in the working memory elements.

3.3 Rete

Rete [24] is a forward chaining algorithm where pattern matching is optimised. Rete uses two kind of memories, namely a *working memory* and a *production* *memory*. The working memory contains facts that represent the current state of the production system, whereas the production memory contains productions, also referred to as rules. The facts in the working memory are represented by working memory elements that contain several fields. Each of these fields contains exactly one symbol, which is a non-variable. Considering the example of a block world [21], where blocks of various colours can be positioned beneath, on top of, or beside another block. So, a possible working memory could contain seven working memory elements that represent the current state of the blocks:

```
colour(block1, red)
colour(block2, green)
colour(block3, blue)
position(block1, beneath, block2)
position(block1, beside, block3)
position(block2, on, block1)
position(block3, beside, block1)
```

Listing 3.9: *Block world* example [21]



A rule of the production memory has a set of conditions that must be met in order to execute the set of actions that form the second part of the rule. A rule is typically denoted by using the LISP-like syntax that was shown in listing 3.5.

The example production *block_on_ground_production* shown in listing 3.10 has two prerequisites and one single consequence. A block `?a` can be considered as positioned on the ground whenever that block is placed besides another block, `?b` that isn't standing on another block `?c`. When these prerequisites are all met, a new fact `position(?a, on, ground)` is added to the working memory.

```
( block_on_ground_production
        position( ?a, beside, ?b ),
        not position( ?b, on, ?c  )
        =>
        make position( ?a, on, ground ) )
```

Listing 3.10: Block on ground production

The actual matching algorithm considers the conditions of the rules in the production system and searches for those productions in the system whose conditions are all met. When such a production is found, the actions that form the consequences of that rule are executed.

3.3.1 The Algorithm

The RETE algorithm represents rules as a network of nodes containing memory tables. The construction of the RETE network itself are explained by using the example *wizards* production that is shown in listing 3.11 which was already explained previously. The RETE network of the `wizards`-production is shown in figure 3.1. The rest of this section gives an explanation of how this network is built.

```
(wizards
has_attribute(?name, magic wand),
has_attribute(?name, hat),
has_attribute(?name, rabbit),
artist_name(?name)
=>
is_wizard(?name))
```

Listing 3.11: Rete *wizards* rule

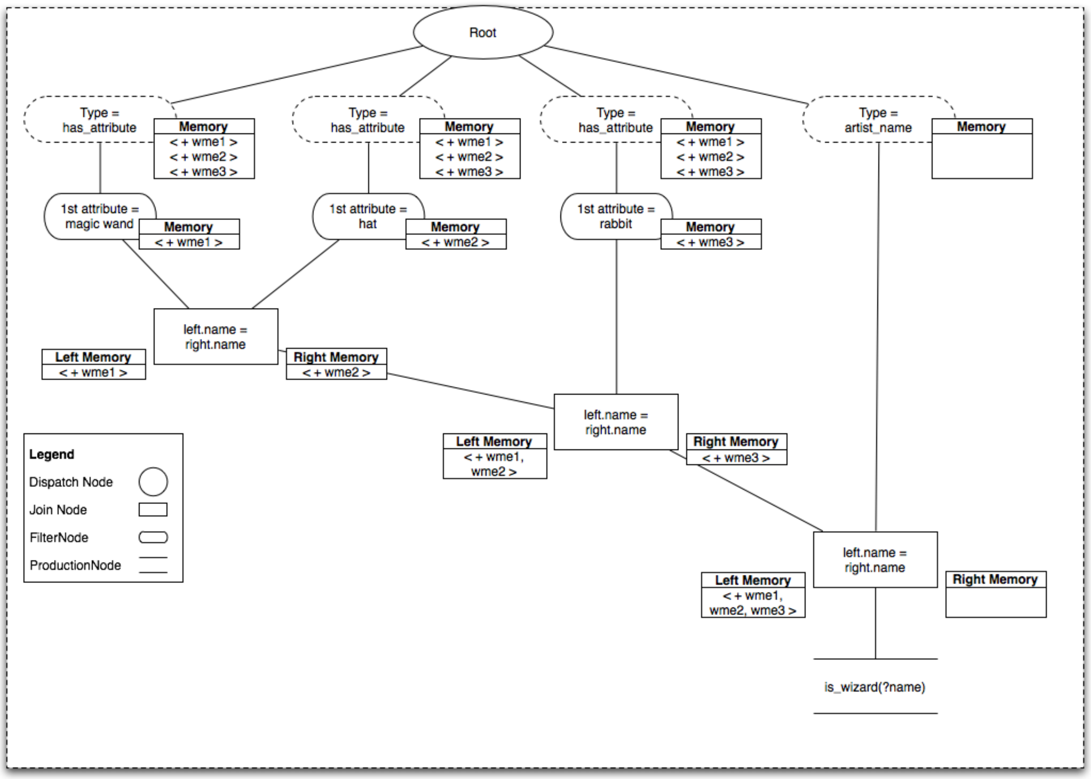


Figure 3.1: Rete network of the *wizards* rule

Rete reacts on updates from both the working memory and the production memory to maintain consistency. Hence, whenever a working memory element is added or removed from the working memory, the algorithm should check which conditions are met. The same reasoning holds for the addition or removal of a rule to the production memory.

The Rete algorithm builds a network which is conceptually divided in two parts dubbed the alpha and the beta network.

Alpha Network

The first part of the RETE network, the *alpha network*, is responsible for building nodes that perform constant tests (i.e. whether the attribute corresponds to "hat") on the working memory elements. The working memory elements that pass the test are kept in the *alpha memories.* The nodes of the alpha network are also responsible for performing tests on the *intra-elements*, which are variables that occur more than once in the same condition of the production. An example of such an intra-element `?name` is shown in listing 3.12. The rule `easyName` makes a new fact for every person with a first name equal to his last name.

```
(easyName
         person(?name, ?name)
         =>
         easyName(?name)
```

Listing 3.12: RETE intra-element

The alpha part of the RETE network consists of *filter nodes.* There are two kind of filter nodes: a first one filters out those working memory elements that have a wrong type. So, as there are four conditions in the *wizards* production, four filter nodes are added to the root node of the network, one for each kind. Three filter nodes test whether the type equals `has_attribute` and a last one tests on `artist_name`. For the first three patterns in the prerequisites of the production, all constants test require another filter node, one for `magic wand`, `hat` and `rabbit`. These constant filter nodes are attached under the filter node that tests the type of the precondition they appear in. Those nodes form the alpha part of the *wizards* production network are depicted in figure 3.2.

The second kind of filter node filters out elements which have a non-consistent binding for their intra-elements. So for example in listing 3.12 there is one filter node to test the consistency of the intra-element `?name`. This filter node filters out person(Duck, Donald) to its children but filters on person(Bamm, Bamm).

Beta Network

The *beta network* forms the second part of the RETE network and operates on two or more working memory elements. The nodes of the beta network, which are called *join nodes*, are responsible for testing the consistency of the variable bindings across different prerequisites of a production. Similar to alpha memories in the alpha network, the beta network uses *beta memories* to store some information. In the case of the beta memory, partial instances of the production – called *tokens* – are kept. A join node has two inputs and for each of its parents there is a separate memory available, namely the left and the right one. Tokens that are passed to the join node from the left parent are stored in the join node's left memory (called a *left activation*), whereas those coming from the right (*right activation*) are stored in the right memory.

In order to build the beta network the filter nodes of the alpha network are combined pairwise by a join node that tests for consistent variable bindings. A

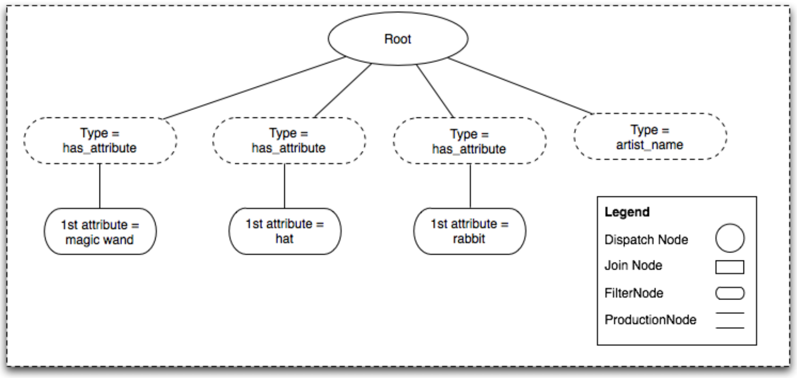


Figure 3.2: Alpha network of the *wizards* rule

join node is a two-input node that has at least one filter node as its parent. The last join node has a terminal node, *production node* or *p-node*, as its child. This node executes all actions in the consequences of a rule, when all conditions of that rule are met. So, each production node can be linked with a unique rule. Note that these actions – addition or removal of facts in the working memory – can again trigger other rules. This is called chaining. The complete RETE network for our running example is given in figure 3.1.

Assuming the working memory contains the elements listed below, the memories of both the alpha and beta part of the network need to be updated.

```
wme1 : has_attribute(Merlin, magic wand)
wme2 : has_attribute(Merlin, hat)
wme3 : has_attribute(Merlin, rabbit)
```

When a new working memory element `artist_name(Merlin)` is added to the working memory, the network is updated again. This working memory element passes the filter `artist_name` and so it is added to the alpha memory of that filter node. This leads to the sending of a token to the children of that filter node. In this case a single join node, where the token is *matched* against the tokens in the other beta memory of that node. The left memory of the join node already contains a token < `+ has_attribute(Merlin, magic wand); has_attribute(Merlin, hat); has_attribute(Merlin,rabbit)` >, the join node tries to match this token with the newly inserted one, < `+ artist_name(Merlin)` >. The only test that this join node performs is the consistency check of the variable `?name`. Because the occurrence of that variable has the same binding in both tokens, namely `Merlin`, a new token (combination of the matched tokens of the inserted token and tokens belonging to the other memory) are sent to the children of this join node: < `+ has_attribute(Merlin, magic wand); has_attribute(Merlin, hat); has_attribute(Merlin, rabbit); artist_name(Merlin)` >.

This token is inserted in the production node, and causes the addition of a new working memory element, `is_wizard(Merlin)`, to the working memory. This addition can be explained because the terminal nodes of the network, production nodes, perform the actual execution of the actions in the consequences of the productions whose conditions are all met.

Comparison with Relational Databases

The division of the alpha and beta network with their own specific nodes resembles the structure that is obtained by relational databases. The working memory itself can be represented by the relation, whereas a production can be simulated by asking a query on that currently available relation. The constant tests that are performed by the nodes in the alpha memory resemble the select operation, whereas the join operation that is used in relational databases can be represented by the tests in the join nodes of the beta part of the RETE network.

3.3.2 Efficiency

The RETE algorithm is able to increase the efficiency of normal pattern matchers because it uses optimisations that decrease the computation time by caching intermediate results in the memories of its nodes. First of all, the algorithm is state-saving: the use of memories in both the alpha and beta part of the network that store intermediate results between successive changes doesn't lead to recomputation of previously calculated results when new queries are launched. Furthermore, the algorithm shares nodes that are used by the conditions of several productions, so the memories of those nodes can also be reused, as explained in the remainder of this section.

Node Reuse

As is already mentioned before, the reusability of certain nodes and memories causes the RETE algorithm to be more efficient than a naive implementation without node reuse. Considering the *wizards* production given earlier, there are three prerequisites with the same kind in the rule. So, instead of building three different filter nodes to test on this kind, those filter nodes can be grouped together in one single filter node that has several children. Even for this simple rule, less memory needs to be allocated, as can be seen in figure 3.3.

Modifications at Runtime

This paragraph discusses the addition and removal of productions that can take place at runtime. These changes are partially supported by OPS5 [35]. However, for the addition of a production, it does not entirely support the wanted behaviour: the existing working memory elements in the working memory are not matched with those new production, resulting to inconsistencies. Only the working memory elements that are added after the addition of the production are matched with this rule.

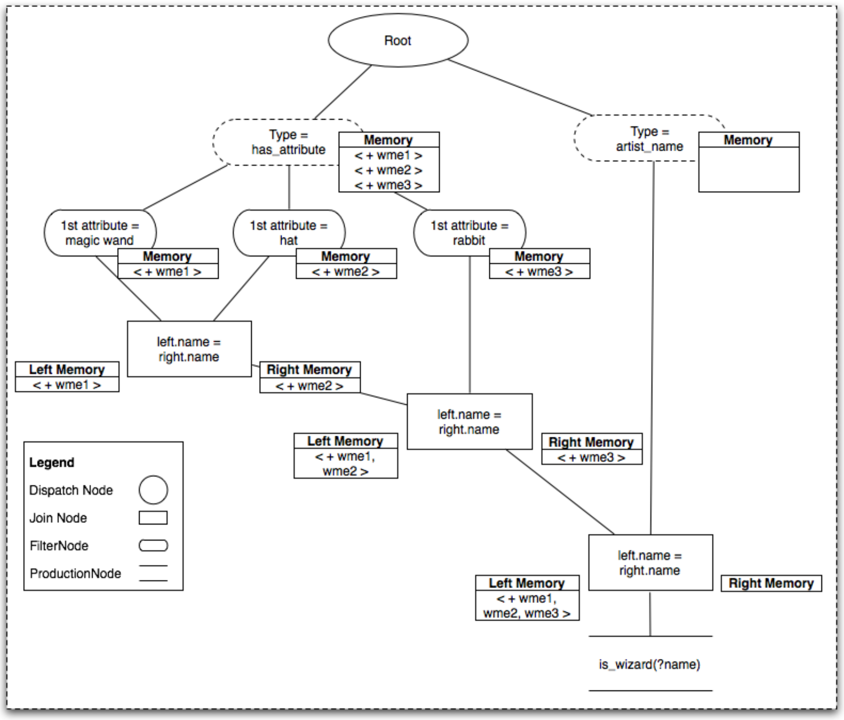


Figure 3.3: Rete network with reuse of filter node

The removal of an existing production uses a bottom-up approach to update the Rete network. First, the production node is removed, and the memories of its ancestors are also updated (tokens are removed), and the link to that production node is removed from their set of children. The cleaning up of the tokens in the memories can only be done when this memory is not shared for other nodes, which is typically the case when nodes are being reused to save memory. When the nodes are being removed the memory space that was used can be deallocated.

3.3.3 Details of the Basic Rete Algorithm

A first detail we discuss here is the introduction of one or more negated patterns in the conditions that form the prerequisites of a production. This section first discusses the use of a single negation, followed by the negation of a combination of patterns.

Single Negation

Suppose the *wizard* rule is changed as below.

```
(wizards
         has_attribute(?name, magic wand),
         not gender(?name, woman)
         has_attribute(?name, rabbit),
         artist_name(?name)
         =>
         is_wizard(?name))
```

Listing 3.13: *Wizards* rule with second prerequisite negated

The second pattern is proceeded with a *not* symbol. So according to this new production all wizards are not known to be female. A negative node or not node which is a new kind of RETE node – beside the filter, join and production node that were already discussed in a previous paragraph 3.3.1 –, is needed for testing the negation in the second prerequisite.

This negative node takes a token $< + \textit{wme1} >$ from the left, and propagates this token further to its children unless there is a working memory element whose variable binding is consistent with that of *wme1*. The negated node resembles a join node, but differs in the way matched tokens are calculated. When the right memory contains tokens that match with the tokens inserted from the left, the inserted token is not be passed to the children of the negation node. This can be explained because the tokens in the right memory of the node all have a second attribute which equals `woman`.

The RETE network for this production is depicted in figure 3.4. In this figure we see that instead of a join node which is shown by a single lined rectangle, a negated join node is introduced, which is shown by a double lined rectangle. This negated join node's left parent is the filter node for `magic wand`, and the right parent is the filter node for `woman`. The network in figure 3.4 uses a working memory that contains the following working memory elements:

```
wme1 : has_attribute(Merlin, magic wand)
wme2 : has_attribute(Merlin, hat)
wme3 : has_attribute(Merlin, rabbit)
```

So, there are no tokens in the right memory of the not node and the hence tokens in the left memory of that node are all passed to the children of the node. When the working memory element `gender(Merlin, man)` is added to the working memory, because there are no tokens in the left memory. So, the token < `+ has_attribute(Merlin, magic wand)` > are passed to the children of the negated node. The other tests of all the join nodes below this node are satisfied, and the production node ensures the addition of a fact `is_wizard(Merlin)`.

When the following working memory elements are added,

```
wme4 : has_attribute(Alice, magic wand)
wme5 : has_attribute(Alice, hat)
wme6 : has_attribute(Alice, rabbit)
```

the addition of `gender(Alice, woman)` – referred to as wme7 – to the working memory causes the addition of a token in the right memory of the negation node. The left memories of the not node and the join nodes of the beta network are also updated: `wme4`, `wme5`, and `wme6` are added in the appropriate node. Note that

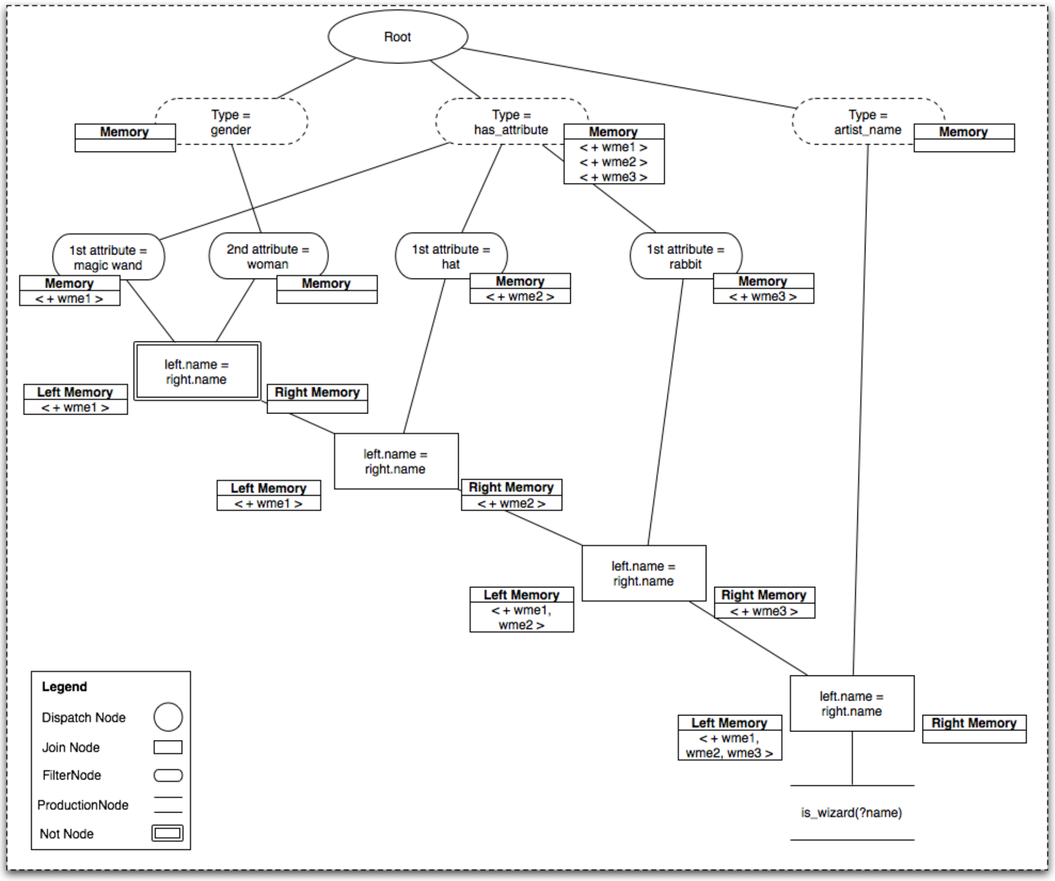


Figure 3.4: Rete network with the second pattern negated

the addition of `wme4` up to `wme7` happens atomically. Now, there is a match between `wme4` and `wme7`, and in this case the token `has_attribute(Alice, magic wand)` is not propagated through the network.

Conjunctive Negations

Conjunctive negations test the absence of several working memory elements. The implementation of these negated conjunctive conditions resembles the one when only one pattern is negated. Consider the block world example that was already used earlier, and the production given below:

```
colour(block1, red)
colour(block2, green)
colour(block3, blue)
position(block1, beneath, block2)
position(block1, beside, block3)
position(block2, on, block1)
position(block3, beside, block1)

(on_floor
        position(?x, on, ?y),
        position(?y, beside, ?z),
        not { position(?z, on, ?a),
              colour(?z, green) }
        =>
        on_floor_not_beside_green(?y))
```

Listing 3.14: *Block world* example with negation

The variable binding for *?z* inside the not subexpressions refers to the variable binding for *?z* outside that subexpression. The usage of new symbols braces, around the not subexpression supports the nesting of such expressions. This makes it possible to nest arbitrary combinations of *for each* (∀) and *there exists* (∃) quantifiers.

The *on_floor* production searches for a block, `?y`, that has another block on top of it and stands beside another block, `?z`, that stands on the floor and hasn't got a green colour. Given the working memory elements listed above, the first block – block1 – satisfies all the conditions so **`on_floor_not_beside_green(block1)`** is added to the working memory.

The use of conjunctive negations is crucial since ordinary conditions have the semantics ∃xP(x), requiring ∀xP(x) to be rewritten as ¬∃x¬P(x). For example to check whether every red block has a blue block on top of it can be reformulated as that there is no red block that does not have a blue block on top of it. So, the conditions of the rule for that formulation look like [21]:

```
not { colour(?x, red),
      not { position(?y, on, ?x),
            colour(?y, ?blue) } }
```

Listing 3.15: Example: nesting conjunctive negation

3.4 Jess

JESS is a high-performance symbolic reasoning engine for the JAVA platform [25]. This languages makes it possible to write JAVA software, for the domain where knowledge is represented by rules. JESS supports some special features that other reasoning engines do not so fully support: the language supports both backward and forward chaining, allows queries to the working memory, can manipulate and reason about JAVA objects and is also a JAVA scripting environment. JESS uses the RETE algorithm described in section 3.3.1 as its inference engine. The rules can be specified in two formats: the first format is a pattern based language, the JESS language, the other is by using XML, called JESSML.

3.4.1 Jess Rule Language

The Jess rule language primitive data elements consists of constants, lists, and variables. Moreover, Jess allows functions to be used in rules. These functions encode an imperative control glow using *if*, *while*, *for*, *foreach* and *try*. All facts are specified by a *template* consisting of a name and attributes, also called *slots*. This closely resembles a class from an object oriented language. The working memory contains several facts and the rules of the reasoning engine can have an addition, removal and modification of those facts as their actions.

Jess supports three kind of facts: *unordered*, *shadowed*, and *ordered*. *Unordered facts* support named attributes, for example:

```
(automobile (make Ford) (model Explorer) (year 1999))
```

As can be seen in the example from the Jess users manual, the syntax of the Jess rule language is the same as CLIPS [27] which is s-expression based. The template for this fact can be defined as follows (example from Jess users manual):

```
(deftemplate automobile
        "A specific car."
        (slot make)
        (slot model)
        (slot year (type INTEGER))
        (slot colour (default white)))
```

Listing 3.16: Jess: automobile template

As this template shows, it's possible to specify the type of the value of a certain slot as well as a default value for it. The second type of facts are shadow facts. These facts are a special kind of unordered facts: they serve as bridges to the Java objects. Shadowed facts make it possible to put Java objects in the working memory and reason about them. Just as the unordered facts, these shadow facts need a template which in this case should just refer to the corresponding Java class. So, reconsidering the car example given above, we might be using *Car* objects that look like the following excerpt of code:

```
public class Car {
        private String brand_;
        private String model_;
        private Integer year_;
        private String colour_="white";

        // some additional methods
}
```

Listing 3.17: Jess: `Car` object

The template for a shadowed fact car is

```
(deftemplate Car
        (declare (from-class Car)))
```

Listing 3.18: Jess: shadowed fact car

This `from-class` declaration creates slots that correspond to JAVA beans properties. When including the `include-variables` declaration the public member variables of the class are used to define some extra slots:

```
(deftemplate Car
        (declare (from-class Car)
                 (include-variables TRUE)))
```

Listing 3.19: JESS: define extra slots

The last kind of facts are the ordered facts. These are used when the slot names are redundant and there's no point in naming the attributes. For example a fact that is representing a single number could be written as *(number 42)* instead of *(number (value 42))*. The template for ordered facts uses the specification that the fact should be ordered: *(declare (ordered TRUE))*.

JESS also supports the use of *modules*. A module defines a namespace for rules and the templates for the different kind of facts. By grouping rules and templates together in modules, complex systems become easier to write, to debug and to maintain. Besides the managing of large quantities of templates and rules, these modules also introduce a control mechanism: the rules of a module are only fired when this module is active (we say the module has the *focus*). There can only be one module at a time that has the focus. When no module is explicitly specified, the constructs are considered as belonging to the MAIN namespace, that forms the current one at that moment.

3.4.2 Salience and Conflict Resolution

The JESS rule language uses *salience* and *conflict resolution* to determine the order in which rules must be triggered. Every production/rule has a *salience property* which resembles a priority for that rule. Rules with a higher value for this property are triggered before rules with a lower salience-value. These values for that property can be integers, global variables and even function calls. Besides this salience property, JESS also uses a *conflict resolution strategy* to impose an order on rules with the same salience that can be depth-first or breadth-first. When using the depth strategy, the most recently activated rules are triggered before other rules that have the same salience value. The breadth conflict resolution strategy causes the triggering of the rules with the same salience in the order that they are activated. All activated rules which are not yet triggered are grouped together in an agenda. The agenda applies a resolution strategy on the triggered rules. To make this more concrete consider the example shown below, here two rules are declared. One to call fire assistance when a building is on fire, another rule is for calling the ambulance in case when somebody is hurt during a fire. Normally it is not deterministic which rule would fire first, however by specifying the salience, the *AlarmHurt* rule fires first because its salience value is higher.

```
(defrule FireAlarm
        (declare (salience 10))
        (burning (building ?name))
```

```
          =>
        (call "Fire Assistance")
)

(defrule AlarmHurt
        (declare (salience 50))
        (burning (building ?buildingName))
        (Hurt (name ?name))
          =>
        (call "Ambulance, burning wounds")
)
```

Listing 3.20: JESS: *Fire alarm* rules

3.4.3 Features

As was already mentioned, JESS supports some very special features. First of all, both backward chaining and forward chaining can be used for the inference engine. Furthermore, it is possible to use JAVA code from JESS and to embed JAVA in a JESS application. JESS also supports multiple rule engines: one program can have several engines. Each of these engines, RETE objects, have their own agenda, rule base and working memory, and all functions can be called in a separate thread. JESS can be used in multithreaded environments by using several synchronised locks – the most important lock is the one on the working memory.

Extending Jess

JESS allows users to define their own functions. This can be done by extending an available JAVA class, `Userfunction`, and implementing two methods: `getName` and `call`. This first method returns a string that represents the name that the JESS code uses, and the latter one is the actual method that is called. This call method has two arguments, the first argument is a vector representing the JESS code that called this function, the second one is the context. Listing 3.21 presents an example from Jess users manual:

```
import jess.*;

public class ExMyUpCase implements Userfunction {

        public String getName() { return "my-upcase"; }

        public Value call(ValueVector vv, Context context)
                                throws JessException {
                String result = vv.get(1).stringValue(context).toUpperCase();
                return new Value(result, RU.STRING);
        }
}
```

Listing 3.21: JESS extension

When the following excerpt of JESS code is executed, the method call receives a ValueVector containing `(RU.SYMBOL)my-upcase` as its first argument and `(RU.STRING)"foo"` as its second argument.

```
(load-function ExMyUpCase)
(my-upcase foo)
```

3.5 DJess

The first distributed reasoning engine described in this dissertation is DJESS, for distributed JESS, which is an extension of JESS allowing it to be used as a lightweight middleware for sharing contextual knowledge [12]. Figure 3.5 depicts

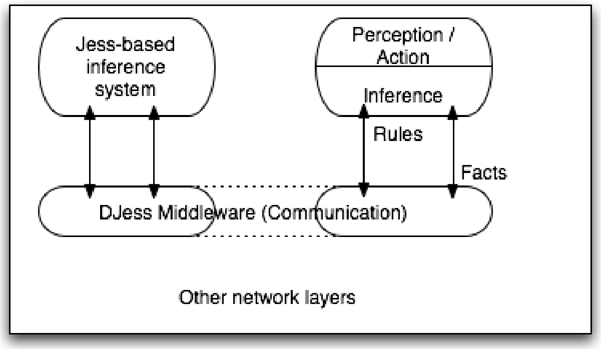


Figure 3.5: Relation between Jess and DJess [12]

the relation between the languages DJess and Jess itself.

Besides the transiently sharing of facts, which represent the contextual information, the sharing of reactive behaviours, rules, is also supported by the language. By using a shared global memory DJESS supports both temporal and spatial uncoupling. Its shared memory resembles the tuple spaces, introduced by LINDA [8]. the tuple space.

3.5.1 Architecture

DJESS connects several inference systems by a *web of inference systems* (WoIS) that has a *shared working memory* (SWM). This web of inference systems is maintained by a *WoIS manager* which manipulates the list of *WoIS members* (list of all the inference systems that are joined) and the SWM. This shared working memory contains facts of several inference systems, and the rules of the different inference systems react and manipulate as being local facts of the inference system itself. Actually, every inference system in DJESS has a local copy of the shared facts in its own working memory. The transparency of this shared memory can be explained because the JESS primitives for manipulating facts of a working memory are reused in DJESS. As was already explained in section 3.4 which describes the JESS rule language, JESS supports three kind of facts: *unordered*, *ordered*, and *shadow*. The latter type of facts are used to build the shared working memory for the web of inference systems. Shadow facts make it possible to use JAVA beans as if they were normal facts of a working memory. These facts are dynamically linked to those objects, so those shadow facts serve as mirrors between the fact and these JAVA beans, also called *proxies*. All proxies of a certain fact are grouped together by a *ghost fact*, which is implemented as a JAVA remote object. Ghost facts enable DJESS to reuse the implementation of the RETE algorithm of JESS. JESS's modules, that were already explained in section 3.4.1, are used to support *namespaces* for the different kind of facts. The programmer can choose at assertion time whether a fact should be private or shared.

Figure 3.6 represents how facts are synchronised. As can be seen on the figure, three JAVA virtual machines (JVM) are used for storing local copies of the shared working memory. f_i represents a shared fact whereas p_i are the according proxy objects and g is the ghost fact that is used for the actual synchronisation. When the second inference system leaves the web of inference systems, the inference system should be removed from the list of members in the WoIS. Furthermore, this removal results in copying that ghost fact on the other JAVA virtual machines and copying the references of the ghost facts whenever a copy of the shared facts is wanted.

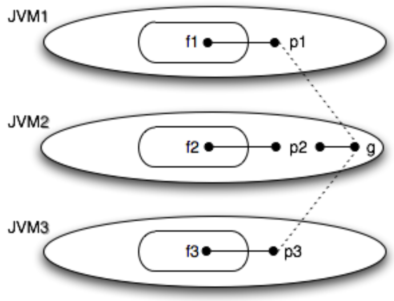


Figure 3.6: DJESS facts synchronisation [12]

DJESS prohibits problems caused by the distributed multitasking environment such as deadlocks and starvation, by means of a *two-phase locking process*: the firing of a rule is preceded by the locking of the acquisitions and after the actual execution the locks are released.

The locking mechanism obtained by DJESS has some crucial problems: if a fact is locked by a certain process and that process dies, the fact remains locked forever. Furthermore, current research of DJESS is concerned about handling network failures in order to maintain consistent shared memory. Moreover, DJess does not handle intermittent disconnections which are inherent to a mobile environment.

3.6 UbiES

UbiES is another example of a distributed reasoning system [39] .

> This expert system, proposes a method to automatically detect and utilise users' context data for expert systems using the ubiquitous computing technologies. [39]

3.6.1 Architecture

The different components of the ubiES architecture are depicted in figure 3.7. ubiES consists of four parts: the *context subsystem*, the *database subsystem*, the *dialogue subsystem* and the *knowledge base subsystem*. The first subsystem, the context subsystem, is divided in four parts namely the contextual database, the context inference nodule, the action request module and the events acquisition module. There are two possibilities to derive context: a first technique is letting the user define knowledge beforehand. The second approach to get the context by various sensor mechanisms, such as for example RFID. This context is saved in the context database. The database management subsystem provides hooks for supporting distributed access to the stored data. The dialogue subsystem provides a customisable user interface, which changes the contents it displays according to the whereabouts of the user. The knowledge base subsystem dynamically changes the knowledge base according to the context of the user. These changes are propagated to the dialogue subsystem which presents them to the user.

3.7 Truth Maintenance Systems

Truth maintenance systems were introduced by Doyle in 1979 [22] as a solution for the problems that knowledge representation systems encounter. Those systems don't guarantee the absolutely certainty: an inferred fact (a fact that is derived from other facts) could possibly be retracted when additional information is added to the system, consider for instance the use of negation in rules.

3.7.1 Truth Maintenance

This *belief revision*, truth maintenance or reason maintenance, revises its set of beliefs when new information is added or retracted about a fixed world. Whereas belief update forms a combination of belief revision together with the reasoning about changes. Belief update reacts on changes in the world (set of beliefs) itself, so no additional information is needed for updating this set of beliefs.

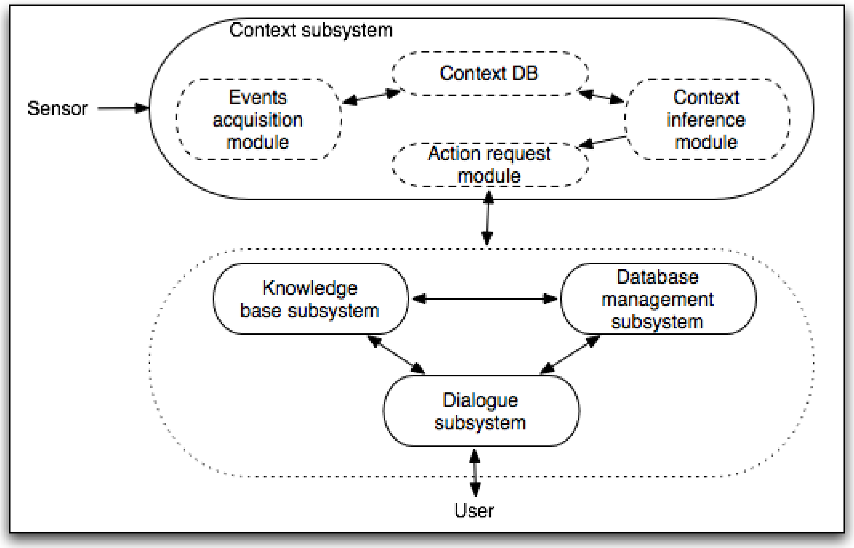


Figure 3.7: Overall framework of ubiES [39]

Suppose the knowledge base contains a fact A, and the negation of fact A should be added to the knowledge base as well. This should simply cause the retraction of that fact A from the knowledge base. However, all derived sentences that were derived from A should also be retracted, for example when a rule A => B is available, the fact B is derived from A and added to the knowledge base. The retraction of A should also lead to the retraction of the fact B, assuming there is no other justification for believing B. Suppose that beside the rule A => B another rule C => B resides in the system, and the knowledge base contains both the fact A and C. The retraction of the fat A form the knowledge base in this situation, does not lead to the retraction of fact B as it is still justified by the fact C.

3.7.2 Truth Maintenance Systems

A *truth maintenance system* revises its set of beliefs and guarantees to avoid contradictions. A very naive truth maintenance system could order the facts from F_1. to F_m Whenever a fact is retracted the system should be reversed to the previous state F_{n-1}. This reversal causes the removal of the fact as well as all its derived facts. Afterwards, all facts can be added again to the knowledge base. This process is illustrated in figure 3.8.

This process guarantees a consistent knowledge base. However this approach is very naive and inefficient because the retraction of one fact cause the retraction and reassertion of all facts after the insertion of fact n together with all the operations to be consistent with the derived facts.

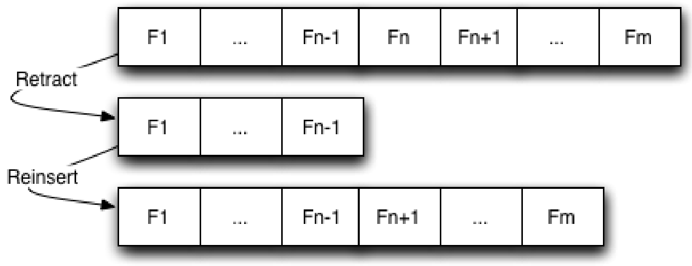


Figure 3.8: Retraction of fact F_n by a naive TMS

A truth maintenance system works together with the inference engine that acts as a problem-solver. The implementation of a truth maintenance system can be either implicitly, in the application itself, or explicitly in the problem-solver tool. As figure 3.9 shows, the truth maintenance system forms the connection between the inference engine and the working memory. The working memory contains facts that represent problem states, so this is dynamic data, whereas the knowledge base contains static data that represent the initial state for the problem solver. The truth maintenance system is revised whenever the context

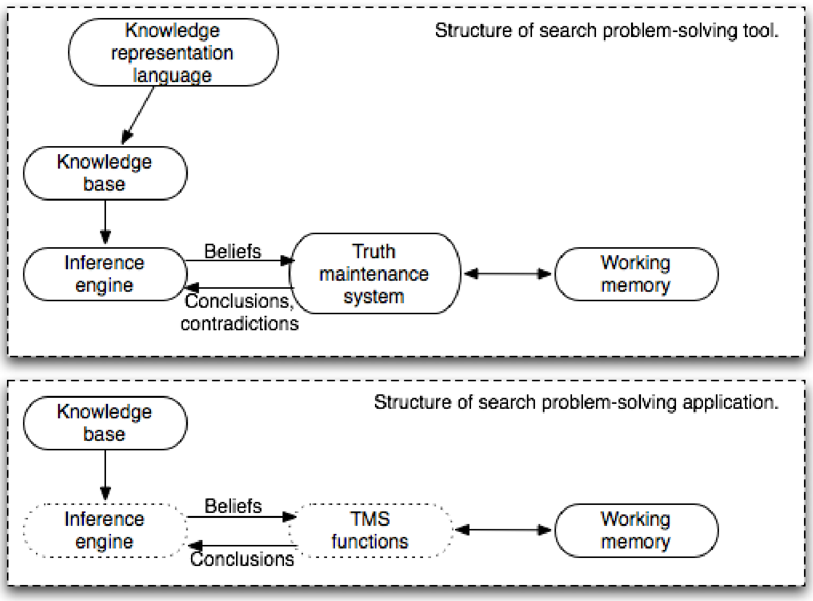


Figure 3.9: Architecture for search problem-solving tool and application [61]

changes, which can be caused by the addition of a new assumption or retraction of one that was already believed. An assumption is believed to be true although there's a lack of evidence for the contrary; So assumptions allow reasoning about incomplete information. The context of a truth maintenance system is uniquely determined by its environment, the set of assumptions, as well as all the derived data. Besides the belief revision, the truth maintenance system is also responsible for handling contradictions.

3.7.3 Justification-based Truth Maintenance System

This type of TMS uses *justifications*, which represent the dependencies between propositions [61]. For example consider the following rule:

```
(defrule profile
        (room entertainmentRoom)
        =>
        (profile loud))
```

This rule sets the profile of a mobile phone to loud while in the `entertainmentRoom`. When this is the case, the JTMS, keeps track of the justification to set the profile on loud. { `(room entertainmentRoom) justifies (profile loud)` }. These justification are used to make retractions faster. When the person walks out the `entertainmentRoom`, the JTMS only needs to check these justifications where `(room entertainmentRoom)` is a member of. So in the example this means that the JTMS must check the justification, { `(room entertainmentRoom) justifies (profile loud)` }. Because there are no other justifications for `(profile loud)`, this fact may be removed. Suppose that there was another justification, { `(user-setting loud) justifies (profile loud)` }, then the fact `(profile loud)` may not be removed, because it is still supported.

JTMS are usably used in a setting where facts have a high chance of being re-inserted. The idea is to mark a removed fact as being out, instead of deleting it. { `(user-setting loud)`$_{out}$ `justifies (profile loud)` } When the fact is reinserted later, the fact is marked as in again and the justifications are used to find which facts must be active again. { `(room entertainmentRoom)`$_{in}$ `justifies (profile loud)` } Note that all the justifications of a certain fact must be marked in, for example when we only want to set the profile on loud during the weekends during the week this results in the following justification: { `[(room entertainmentRoom)`$_{in}$ `ˆ(weekend true)`$_{out}$ `] justifies (profile loud)` } In this situation when the user walks into the entertainmentRoom the profile is not set to loud because the fact `(weekend true)` is marked out. Using this marking mechanism, the system builds a chain of inferences, and does not need to rederivate everytime a deleted fact is reinserted. It only needs to keep track of the justifications.

3.8 Conclusion

This chapter discussed logic programming as this forms the core of the Fact Space Model we propose in chapter 5. Subsequently we highlighted the Rete algorithm which optimises pattern matching for a forward-chained inference engine. Reasoning engines that are most related to our work, were presented and their use in a distributed environment has been evaluated.

The principles of declarative programming combined with a forward chaining reasoning engine, described in this chapter, offer a suitable abstraction for reacting on events. Hence, logic programming solves the problem at hand of the coordination language Lime that was described in chapter 2. Furthermore, the Rete algorithm and the use of a justification based truth maintenance system increase the efficiency of a forward-chained production system to deal with systems where the knowledge base changes frequently. This algorithm is used in the Fact Space Model that is described in chapter 5. The next chapter describes context-aware systems which are the domain of the Crime middleware presented in this dissertation.

Chapter 4

Context Systems

This dissertation advocates the use of a distributed reasoning engine in combination with a truth maintenance system in order to ease the development of context-aware applications. The previous chapter gave an overview of how a reasoning engine can be used in order to infer new contextual information by means of some predefined rules. This enables the user to write the desired behaviour in a declarative manner. This chapter highlights the use of context-aware systems to acquire such contextual information. Some of these systems use reasoning engines similar to those explained in the last chapter. This allows those systems to directly respond to complex changes in the context specified by rules. Context acquisition can be realised based on event channels or by using a logic language. In chapter 5 we advocate the use of a separate logic language to reason about the context since this allows the programmer to make a clear distinction between reasoning about the context and acting upon the context. This is in contrast with the channel based approach where a similar distinction can not be made because the context reasoning is incorporated in the computation of the application itself.

This chapter first gives a short introduction to context in the light of context-aware systems. Subsequently, some frameworks are presented: we begin with frameworks based on the *event-based* approach, namely CONTEXT TOOLKIT, JCAF, and WILDCAT. The subsequent frameworks, CHISEL, GAIA and CORTEX are all based on a logic language. The last part highlights COCOA which is based on *Stigmergy*. To conclude, an evaluation of all those frameworks is given.

4.1 Context

While *context* is a commonly used word, and the every day meaning is clear, its definition from the Merriam-Webster dictionary[1]

[1]http://www.m-w.com

"The interrelated conditions in which something exists or occurs."

is too vague to be applicable in the design of context-aware applications. Beside having an indistinct vague definition, context is also used in a variety of different branches of computer science.

Therefore various proposals have been made to give a more usable definition of context in the light of mobile applications. Most of these proposals fall into two categories, namely enumeration and categorisation of context. A first definition of *context* was given by Schilit and Theimer [57] which states that context is

"The location of use, the collection of nearby people, hosts, and accessible devices, as well as the changes to such things over time".

Other similar definitions – proposed by Brown [9], Ryan [54], and Dey [17] – extend this enumeration with the notion of time, identity, focus of attention, and even emotional state. Although these definitions were meant for practical use, they do not specify how to deal with information they exclude.

Schilit address this problem by proposing a categorisation of context as follows [56]:

1. *Computing Context*: the devices in earshot like mobile phones, displays, the connectivity and the available network resources.

2. *User Context*: who is working with the application, which other users are nearby, and their current activity like for example participating in a meeting.

3. *Physical Context*: weather conditions, battery levels, lighting , or even the rotation of the display.

Chen and Kotz [13] remark a great absence of time, in this categorising definition, and propose the addition of time as a fourth category. Another useful context is retrieved when storing context during a certain period of time, referred to as the *context-history.* This context-history is often overlooked by context-aware applications [13].

We conclude our survey of context definitions with Dey [2], who proposes an operational definition which makes it easier for designers to determine whether information can be regarded as *context* or not.

"Context is any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the applications themselves."

4.1.1 Context Acquisition

Direct Hardware Sensor Access

In this scenario the assumption is made that the application reads the information directly from the sensors, converts the raw data into context and reacts upon it. This approach has some disadvantages, for instance when trying to extend an application to let it also take into account additional context parameters, the acquisition procedure must be rewritten. As the application is almost hardwired to the sensors, this discourages code reuse. An additional disadvantage comes from the fact that this implementation strategy is unsuitable for a mobile environment as some sensors need to be accessed by multiple context clients. In order to manage the access to the sensor at least some kind of *locking manager* is needed to coordinate the access to the sensor. Note that when working with very limited devices it might not be possible to provide a lot of abstractions as the size of the memory to contain the program might be to small for this. Furthermore, in time critical applications the time overhead of a complex abstraction might not be justified. For example applications written for the chips controlling an airbag.

Facilitated Middleware Infrastructure

This approach prescribes the use of middleware to encapsulate the low level sensor access and typically transforms the sensor data to a meaningful value. In contrast with the direct sensor access, this approach improves modularity as the application itself does not need to be rewritten anymore when additional context parameters are added.

Additionally, the code written for data acquisition can easily be reused for a similar hardware setting. A middleware infrastructure is not always suitable as the hardware for running the applications on, is in most cases very limited. However, systems such as the CONTEXT TOOLKIT which is described in section 4.2, illustrate that only a minimal amount of support needs to be provided by the sensor nodes provided that they can communicate with more than one device.

A disadvantage when all devices perceive the environment themselves, comes from the possibility of poor sensor data, when different sensors are used or when malfunctioning sensors provide wrong information, co-located devices perceive the same context differently and conflicting information may be propagated through the network. Therefore some middleware architectures use a context-server as explained in the next section.

Context Server

Rather than having to perceive the context themselves, applications can rely on a context server. This server can then be accessed concurrently by multiple applications and improves the reusability of sensor data. The server removes the burden of each device having to perceive the context itself, and since the resources of these devices are rather limited, this may sound like an excellent solution. However, this strategy is in direct violation with the nature of mobile ad hoc networks because the server must always be accessible and have a complete picture of every possible situation in the world, something that can only be applied in a confined setting.

Widgets

Context widgets extend the abstraction of a device driver in order to hide the complexities involved with providing access to sensors. The basic idea behind *widgets* is to abstract over the type of context information, rather then over devices. By making the abstraction at the context side, similar hardware devices can be replaced without needing to change the applications using them. Communication with devices is then replaced by message passing to the widgets and callbacks. These widgets are controlled and preconfigured by a *widget manager*.

4.1.2 Context Sharing

For the second important aspect of context-aware applications, we turn to Winograd [67] who describes three different context sharing models.

Centralised

In a centralised architecture devices do not need to be concerned about the detection of other devices. Instead, they connect to a centralised server which tracks the context of all devices. When devices need to know who is located in their environment, they query the central server. An advantage of a centralised architecture is its easy implementation as all information is kept in one place. However, as all devices need to connect to this centralised service, the scalability of this approach is limited.

Peer-To-Peer

In a peer-to-peer network devices establish a connection with devices in the proximity. Because these connections are established between the mobile devices themselves, without the need of a known server as was the case for the centralised architecture, the scalability increases . The communication form is always, as the name suggests,"from peer to peer", assuming that a sender must know the receiver which does not cope with the ambient environment where unknown devices pop-up as it goes. This is in contrast with the blackboard model explained in the next paragraph.

Blackboards

This technique shifts the focus from a process-centric view to a data-centric view. Information among processes is distributed through a distributed shared associative memory. Processes can subscribe to this shared memory waiting for a special event to happen (encoded as a message). Communication is performed asynchronously, and so both space and time uncoupling is provided. For a more in to depth overview of this model see chapter 2.

4.1.3 Common Structure

Many context-aware systems use the same distributed components for reducing the complexity of context-aware applications. The common components of the architectures are discussed by means of an example application.

Example

Consider an example application where a user wants the profile of his mobile phone to be switched to silent whenever he enters a room where a meeting is taking place in order to not disturb any of the participants.

So, first of all there is a sensor needed for detecting whenever a person enters a room. Furthermore, some additional sensors can be required to determine the activity that is taking place in that room. This could for example be realised by using RFID, an upcoming technology which can be seen as an advanced barcode scanner but without the need to have visual contact with the scanned objects. For instance, a sensor that measures the sound could be useful to determine whether there's a meeting going on or not. A sound and location *widget* produce contextual information according to the sensed data. The produced context information of both widgets is stored by the room *context service*. Additional to the storage of context information, the context service is also responsible for providing access to this information. *Context clients* are applications that adapt their behaviour according to the sensed context. Clients obtain their context information by querying the context service. Beside querying context, a context client is also able to register his interest in certain context and whenever changes occur to this interesting context information, the context client is notified.

This common structure is depicted in figure 4.1.

4.2 Context Toolkit

The first context-aware framework finds its roots in the *GUI widget* [18]. The aim was to apply the experience gained from the GUI widget design in order to create the context widget. A first useful property of the GUI widgets is that they hide the hardware being used from the programmer. For example, it does not matter to the programmer what type of brand the user's mouse has. By hiding this information, the hardware devices can be interchanged with a minimal impact on the application. Furthermore GUI widgets also hide the interaction with the user itself: most of the time the application has to implement a single callback to be notified of the user's interaction with the widget. Another important advantage of the GUI widget is its reusability which stems from the fact that it can be used as a black box. All those advantages formed the inspiration for the development of the Context Toolkit.

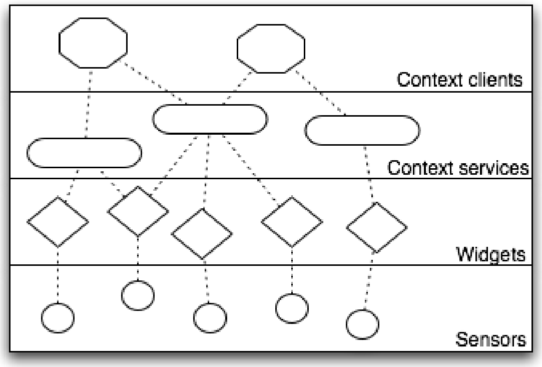


Figure 4.1: Common structure used by context-aware frameworks

First of all we explain how context is modelled by this framework. Then a description of the different components compromising the Context Toolkit is given. Subsequently the communication between the context client and the context service is discussed. To conclude, some advantages and disadvantages of this framework are highlighted. This is done in respect to the common structure of the framework, discussed in section 4.1.3, and the Fact Space Model, we discuss in chapter 5.

4.2.1 Context Toolkit's Context Model

The context model provided by the Context Toolkit consists of all the output produced by the widgets. The user can be notified by changes to the context or can actively retrieve the state of the context. Because the representation of the context given by a widget might be too low level for the application at hand, the Context Toolkit also provides abstractions which allow to combine the output of several context widgets to derive a higher-level context. For example the output of a location widget and the output of a calendar widget could be combined to detect a ongoing meeting. A more in to depth description is given in the next section.

4.2.2 Context Toolkit's Architecture

The Context Toolkit's architecture perceives the physical world by making use of sensors. The context acquired by these sensors can be collected and transformed in the framework to derive higher-order information. This section describes the different components of the architecture and their relation to the common structure discussed in section 4.1.3. First of all, the architecture is briefly explained by making use of an example application, followed by a more-detailed description of all its components.

Example

In this section we revisit the example application shown in section 4.1.3 and explain how the Context Toolkit's framework can be specialised for its implementation. When someone enters a room where a meeting is taking place on a weekday, the profile of his mobile phone should be switched to silent.

The overall architecture of the Context Toolkit for this example application is depicted in figure 4.2. The connections between the components represent the interactions between them.

First of all two sensors are needed for retrieving the context information. Location information is gathered by the `RFID` sensor which detects when someone enters the room. The output of the `RFID` sensor is passed to the location widget, which is able to determine the person's name by using the *RFID to name interpreter*. Similar to the location widget, the calendar widget uses the *day of the week interpreter* to determine if the actual date, that is retrieved by the `iCal` sensor, is a weekday or a day of the weekend. The location and calendar widget are combined by the *Profile aggregator*. This aggregator stores the information of both widgets and makes this information available for the `Profile` context client. The context client needs a context service for performing the actual executions, namely the switching of the mobile phone's profile. This is realised by the `Profile` actuator.

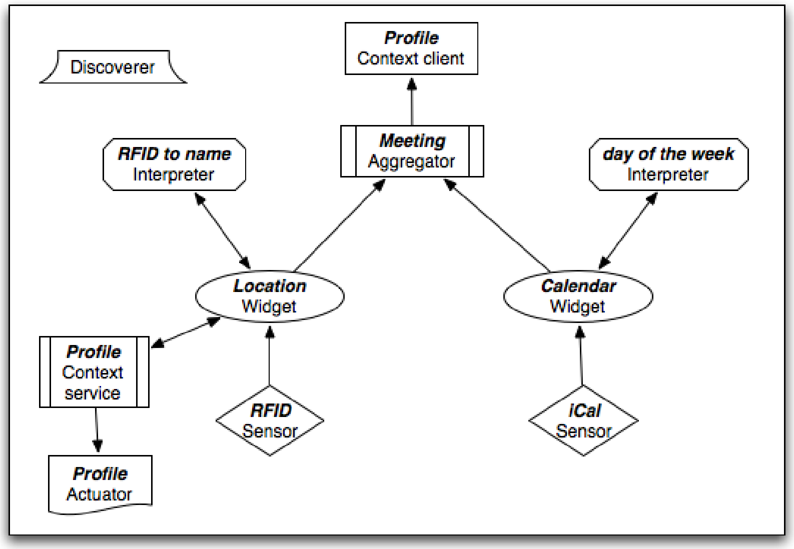


Figure 4.2: Context Toolkit's architecture [18]

Widget

A first component of the architecture is the *widget*: it separates the specific sensors used from the contextual information they provide. Introducing widgets in the framework results in hiding the complexity of the hardware that models the world for the actual applications. Furthermore, widgets enable reusability for context sensing as several context clients can reuse the same widget. Context widgets don't only encapsulate the context information, they also provide methods for accessing the contextual information they retrieve from their sensors. Context clients are able to request the context information, query attributes of the widgets or they can be notified by the widgets when significant context changes occur.

Aggregator

Sometimes, the contextual information provided by one widget is part of the information needed by a context client to react upon it. *Aggregators* collect multiple pieces of contextual information that is logically related and make it available for context clients in one software component. Using aggregators decreases the complexities of context delivery to the context clients that use this context information, as the context client doesn't need to query each of those widgets, but only the single aggregator that combines the output of those widgets. These aggregators can be reused between several applications that are interested in the same context information. An aggregator has the same capabilities as the widget: it can notify context clients of relevant changes, it can be queried for updates, and its stored context can be accessed by the context clients.

Interpreter

Interpreters transform context information to higher-order context information by taking one or more context sources and producing new context information. For example, when there are a lot of people in a room who are all seated around a table and most of the time there's only one person talking. By combining these contextual sources, one could guess that there's a meeting taking place in that room. The combination of several context sources and deriving a higher-order context information is realised by interpreters. By introducing interpreters, this derivation of higher-order context information doesn't need to be realised by the application itself, and other applications that need the same derivation can reuse the interpreter. Note that an aggregator is only capable of combining several widgets and is unable to interpret these combined context sources.

Context Service

The previously described components of the architecture of the Context Toolkit are responsible for the actual context acquisition and delivering it to the interested *context clients*. The fourth component, context service, executes the actions on behalf of the application. The context service controls the changes in the environment by using an actuator that can be regarded as an output.

Actuator

Actuators are abstractions with the opposite intention of widgets: whereas widgets retrieve context information, actuators actively change the context by performing an action. This action is triggered by a certain contextual state. The actuator perceives the context by using a context client. When the contextual state to act upon is perceived by the context client, it informs the actuator which executes its actions.

Discoverer

The last component of the architecture is the *discoverer* that controls a registry of all the capabilities in the framework e.g. the available widgets. A discoverer can be used by context clients for finding a particular context component that is interesting for them. When any of the previously described components starts, it should notify the discoverer of its presence and how it can be contacted. For example, a widget notifies the discoverer of what kind of context information it provides.

4.2.3 Conclusion

The architecture of the Context Toolkit maps very well on the general architecture used in most context-aware systems described in the previous sections. The Context Toolkit architecture and the components used in it are used as a reference to explain the subsequent context-aware systems.

4.3 JCAF

The Java Context Awareness Framework (JCAF [4]) allows programmers to build context-aware applications. As with most context-aware frameworks, the focus of JCAF is to provide the ability to react on changes in the environment. Moreover, applications should be able to register their events of interest. For example, on a change in context, so that they can be notified when such events occur. JCAF is an example of an *event-driven* framework which supports distributed context services. These distributed context services are connected with eachother in a peer-to-peer network.

4.3.1 JCAF Context Model

JCAF allows modelling relational context between an *entity* and other entities residing in their environment. Entities abiding in one another's environment are called *context items*. Consider the example to express that "Alice" is located in the conference room. This expression is modelled in JCAF as follows: Alice is modelled as an entity that has a *relation*, namely "is located", with the conference room. The conference room in his turn is modelled as an item in the context of Alice. Note that an entity can also be a context item, for example when we reformulate the previous example to: the conference room is populated by Alice. Then we see that Alice is a context item of the conference room.

4.3.2 JCAF Architecture

In order to model relational context, abstractions of an *entity*, *relation* and *context item* map directly to interfaces provided by JCAF. An apparent difference with the Context Toolkit is the direct abstraction of an entity. Whereas context in the Context Toolkit is indirectly captured as output of all context widgets, here the context is logically attached to an entity.

Context services in JCAF provide access to the context of a group of entities. The context of an entity can be changed through the context service it is part of. In JCAF everything that reads or changes the context by accessing the *context service* is called a *context client*.

Widgets, which are called *context monitors* in JCAF, update the context of a certain entity. For example a location widget uses a RFID scanner for updating the "located" relationship of the scanned entities.

Context actuators are registered to a specific relation of the context service and the actuators allow to react whenever that relation has changed. For example, to implement an *in-out board*, a context actuator could register itself to the relation "location" of a context service. The context service then informs the context actuator when the "located" context of a certain entity has changed. The actuator then changes the in-out board according to this context change.

Another difference in functionality of the context service is how access to context is facilitated. Whereas in the Context Toolkit context widgets can be accessed directly, the widgets in JCAF must be accessed through the context service.

The remainder of this section highlights the components of JCAF's architecture.

Entity

An *entity* models some real world object, for example a cellphone. Entities are modelled in JCAF by implementing the `GenericEntity interface`. Entities are notified of changes in the context by the `contextChanged` method.

An example of a cellular phone entity that changes its location when a context change is triggered, is shown in listing 4.1. When a user enters a conference room, the `contextChanged` method is invoked and changes the profile of the cellphone to silent, when the user enters any other room the method changes the profile to loud.

```
public class CellPhone extends GenericEntity {
```

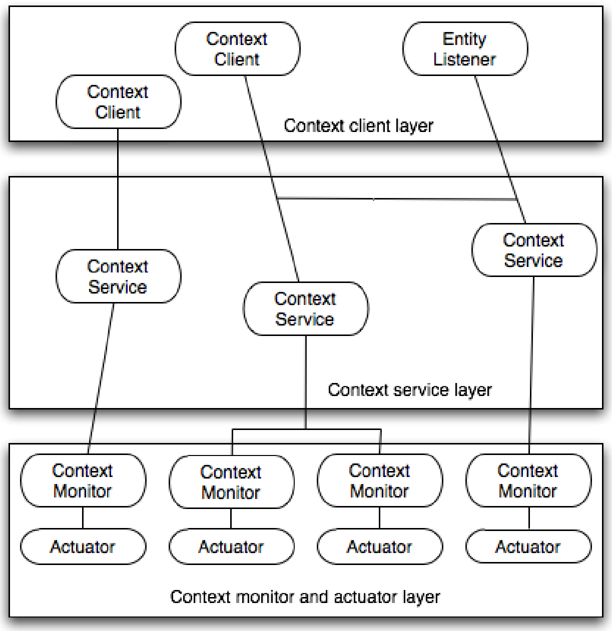


Figure 4.3: JCAF architecture [4]

```
  public void contextChanged(ContextEvent event) {
    try {
      this.location = ((Location)event.getItem()).getId();
      if(this.location == conferenceRoom) {
         Profile.setMode(SILENT);
      }
      else{
        Profile.setMode(LOUD);
      }
    }
    catch(ClassCastException e) {
      System.error.println("Unknown event, Exception");
    }
  }
}
```

Listing 4.1: JCAF: entity example

Context Service

Whereas in the Context Toolkit the *context service* acts as a repository for context, the context service in JCAF is extended to accommodate entities. This is realised by letting entities register themselves to a context service. Context services provide access to their entities and allow these entities to change their context. The registration of a cellphone entity to its local context service is shown in listing 4.2.

First, a new mobile phone called `myPhone` is created. Then this entity is added to the context service with the `addContextEntity` method. The last line of code retrieves the `myPhone` entity from the context service by using its id. Other methods to retrieve entities from a context service include, `getAllEntities` and `getAllEntitiesByType`. Those additional methods respectively return all the entities a context service accommodates or only those of a specified type.

```
CellPhone myPhone = new CellPhone();
getContextService().addContextEntity(myPhone);
getContextService().getEntity(myPhone.getId());
```

Listing 4.2: JCAF: registration of a cellphone to a context service

The context service includes a functionality to change the context of a certain entity. Changing context of an entity by making use of the context service is shown in listing 4.3. The second line in the code excerpt makes a new relation, namely “location”. Then by making use of the `addContextItem` method we set the “located” relationship of `myPhone` to `conferenceRoom`. The entity is informed of this change by the `contextChanged` method, as shown in the previous section.

```
Location conferenceRoom = new Location("conferenceRoom");
Located located = new Located();
getContextService().addContextItem(myPhone.getId(), located, conferenceRoom);
```

Listing 4.3: JCAF: changing context by using context service

Widgets

Widgets in JCAF can be accessed either asynchronously or synchronously. Asynchronous widgets report changes as they sense it. For instance, a push button sends an event everytime it is pushed. Synchronous widgets can be queried for their state because they do not notify the context clients whenever a change occurs. Instead of being notified, context clients must query the state of the widgets. For example a synchronous switch does not send an event whenever it is switched, but when a query is sent, it responds its status (on/off).

A widget for tracking the location of entities by making use of bluetooth is shown in listing 4.4.

```
public class conferenceRoomWidget extends AbstractMonitor
                                  implements BlueToothListener {
  public conferenceRoomWidget(String service_uri) {
    super(service_uri);
    located = new Located();
  }

 public void blueToothEvent(BEvent event){
    if(event.ENTER_ROOM) {
       getContextService().addContextItem(
                            event.getId(),
                            located,
                            newLocation("conferenceRoom"));
    }
}
```

Listing 4.4: JCAF: widget for tracking the location of entities

Actuators

Two types of *actuators* exist in JCAF. One type of actuator, a *context event*, performs an action whenever a certain relationship has been altered. The other type, the *entity listener*, performs an action whenever a certain context item of the entity it is listening for, has changed.

4.3.3 Peer-To-Peer Communication

Context services can be connected in a peer-to-peer network. Remote context service can be added and removed at runtime by specifying their address. The main purpose of setting up such a network is to allow the lookup of entities which reside on a remote context service. Entities can be searched by making use of the `lookupEntity` function of the context service:

```
public void lookupEntity(String id,
                         int hops,
                         RemoteDiscoveryListener dl);
```

The first parameter, `id`, of this method specifies which entity to search for. The number denoted by `hops` indicates the distance from the originating context service to search for that entity. The `RemoteDiscoveryListener` is notified when a new entity has been found.

4.3.4 Conclusion

The JCAF architecture allows to search context services residing on other devices by making use of an identifier. However these context services must be in connection with a known context service. The framework also does not give any support when context clients disconnect from their context service, therefore assuming a stable connection with the context service.

4.4 WildCAT

WildCAT is a Java toolkit for building context-aware applications. Its innovative feature stems from the fact that it proposes a hierarchical structure to model context information which facilitates querying and dynamically discovering of context information. Moreover, since this structure is merely an overlay structure over existing *widgets*, it ensures that their output can be introduced in the correct place in the context model. Access to the context information can be either synchronously or asynchronously, as is described in section 4.4.3.

Recall that context-aware applications typically exhibit three characteristics. First of all, the applications should support dynamic discovery of context information. Secondly, the applications should be able to reason about that context information. Finally context-aware applications should be notified when interesting events – changes to the context – become available. Those three characteristics are fulfilled by WildCAT's API.

4.4.1 WildCAT Context Model

Context in WildCAT is modelled as a set of *widgets* represented in a tree structure. This structure is important because it provides the programmer with an easy and deterministic way to specify a widget. The hierarchical structure needs not to be reflected by the widgets, but only in the information that they provide. The information provided by widgets in WildCAT is represented by *widget attributes*, that are key-value pairs.

Domains group widgets which provide related contextual information, so a domain separates the different aspects of the context. This separation enables specific implementations for these aspects. As specified in [16], some example domains provided by WildCAT are:

- *sys*: hardware resources, these may represent storage devices or input devices like a mouse or a touchpad.
- *net*: topology and performance of the network like the number of dropped packages or the available bandwidth.
- *geo*: geophysical information, this information may include the typical physical ones, like the GPS coordinates and weather conditions, but are not limited to these. Under this domain are also physical environments included, like for example a classroom.
- *user*: user preferences, characteristics and current state. Examples include being in a meeting, having class which is different from being located in a classroom, or wanting to be informed about the changes of the weather condition.

4.4.2 WildCAT Architecture

Beside providing a hierarchical structure to model context information, the WildCat framework also contains an *extension interface* and a *data acquisition framework*. The extension interface enables the specification of new domains. The data acquisition framework is responsible for acquisition of data via widgets, which are referred to as *sensors*. Widgets in WildCAT can be *active* or *passive*: an *active widget* runs in its own thread and provides the information asynchronously. *Passive widgets* are also asynchronous for the programmer's point of view, but is queried by the framework at regular intervals. Both kind of widgets are explained in more detail in the remainder of this section.

Sensors

Sensors, or *active widgets*, run in their own thread and announce their information as it comes available. An example of an active widget is shown in listing 4.5. The JAVA class `RoomSensor` implements the `Sensor` interface provided by WILDCAT. The `RoomSensor` implements the `BlueToothListener` interface in order to be updated about the current location. All active widgets must notify changes to their context by using a *samples listener*. In the example this is `myListener`. When the user walks into a room, the `blueToothEvent` method is called with a `BEvent`. If the event is triggered by the entering of a room, the key-value attribute is changed. Specifically the key "location" is modified to the new room which we read from the `BEvent` by calling the `location` method. When we do have changed our `SampleSet`, we inform the listener `myListener` of this change.

```
public RoomSensor implements Sensor
                  implements BlueToothListener  {
  SamplesListener myListener;
  SampleSet mySet;
  ...
  public void blueToothEvent(BEvent event){
    if(event.ENTER_ROOM) {
        mySet.put("location",event.location());
        myListener.sampleChanged();
    }
  }
  ...
}
```

Listing 4.5: WILDCAT: active widget

Passive Sensors

Passive sensors, or *passive widgets*, do not inform changes to their environment directly, instead they keep track of their context and return the state of their context when asked. A passive version of the `RoomSensor` is shown in listing 4.6. Instead of implementing the `Sensor` interface, the `PassiveSensor` interface is implemented. Again the `BlueToothListener` interface is implemented to be informed about the detected bluetooth devices in the proximity. Instead of directly informing a `SampleListener` of the change in location when the `blueToothEvent` method is called, this information is stored in `mySet` until the method `sample` is called.

```
public RoomSensor implements PassiveSensor
                  implements BlueToothListener  {
  SampleSet mySet;
  ...
  public void blueToothEvent(BEvent event){
    if(event.ENTER_ROOM) {
        mySet.put("location",event.location());
    }
  }

  public SampleSet sample() {
     return mySet;
  }
  ...
}
```

Listing 4.6: WildCAT: passive widget

Extension Interface

In order to inform the framework of *widget* extensions, we have to *register* these to a *domain*. This is done by making use of a XML configuration file. An example where we register the passive `RoomSensor` to the *geo* domain under the name `room` is shown in listing 4.7. Because the `RoomSensor` is passive widget we can specify whenever samples should be taken from it. In this example, WildCAT probes the sensor every half second.

```
<context-domain name="geo">
  ...
  <resource name="roomSensor">
    <passive-widget name="roomSensor" class="RoomSensor">
      <schedule> <periodic period="5000"/> </schedule>
    </sensor>
  </resource>
  ...
</context-domain>
```

Listing 4.7: WildCAT: extension

Events

WildCAT uses *events* to represent changes to the context: these changes can be caused by the addition, removal or modification, of a widget attribute, or the occurrence of a certain condition which can be specified by an expression. All these changes occurring to the context are observable, and correspond to primitive changes in the data model of WildCAT.

Paths

In order to use the context model provided by WildCAT, the programmer must be able to address a certain widget or widget attribute. The addressing scheme adopted in the WildCAT framework finds its roots in the Uniform Resource Identifier [7]. Thereby making a separation between the addressation scheme of a widget and its implementation. This abstraction is called a *path*, and can be used to denote a widget, a widget attribute or a collection of these.

Some examples of such paths are shown in listing 4.8. The first path in listing 4.8 represents the access to the GPS widget, whereas an example path for representing the coordinates attribute of the GPS widget is given by the second path. The two last paths are collections: the first one is a collection of widget, and the last one is a collection containing all attributes of the GPS widget.

```
geo://location/input/gps
geo://location/input/gps#coordinates
geo://location/input/*
geo://location/input/gps*
```

Listing 4.8: WildCAT paths [16]

4.4.3 Communication

In WildCAT, communication with widgets can be performed both synchronously as asynchronously. Communication is performed through the Java `Context` class that is shown in listing 4.9. The first five methods of the `Context` class form the synchronous interface, whereas the three last ones allow the registration of asynchronous listeners.

```
public class Context {
 String[] getDomains();
 Path[] getChildren(Path res);
 Path[] getAttributes(Path res);
 boolean exists(Path res);
 Object resolve(Path res);

 long register(ContextListener listener, int eventKinds, Path path);
 long registerExpression(ContextListener listener, int eventKinds, String expr);
 void unregister(long regId);
}
```

Listing 4.9: WildCAT context interface [16]

As is already mentioned, both synchronous and asynchronous requests are supported by WildCAT. A synchronous request allows the discovering of the structure of the context as well as querying the value of a certain widget attribute. For instance to discover all input devices concerned about location, the following code – listing 4.10 – retrieves all paths to such resources. This allows the programmer to discover hardware resources at runtime, and possibly change its behaviour depending on the available resources.

```
context.getChildren(new Path("geo://location/input/*"));
```

Listing 4.10: WildCAT pull operation

Asynchronous requests enable the notification when some interesting events occur as specified by the provided path expression. These requests follow the *publish-subscribe* pattern: the user first specifies his interests in some events by registering them such that the middleware sends its notifications whenever the specified event occurs. An example of such an event subscription is given below in listing 4.11. This example showcases the behaviour necessary to inform the `deviceListener` every time the `roomSensor` detects a new room.

```
context.register(deviceListener, RESOURCE_CHANGED,
                 new Path(geo://roomSensor#location));
```

Listing 4.11: WildCAT push operation [16]

4.5 Conclusion

WildCAT is based upon events provided by the widgets it incorporates. The focus of WildCAT is to provide a hierarchical structure to easily access the widgets provided by the framework. There is no predefined mechanism for devices to communicate, therefore the communication must be performed ad hoc.

4.6 Chisel

Chisel [37] is a framework for Java which enables applications to adapt their behaviour in response to changes in the context by using a policy driven approach. Adaptations can be performed at runtime using *reflection*, and in particular the use of *meta-types* in order to control the non-functional requirements of the application. Therefore, the design of mobile applications, implemented using the Chisel framework, should only implement the core functionalities and cotter the non-functional requirements using a meta-type, which can be plugged in at any moment. The remainder of this section highlights how these meta-types are plugged in and how this effects the design of the applications.

4.6.1 Chisel's Context Model

Contextual information is gathered by separate *widgets* and made accessible through the use of an *event-driven* programming language. Changes in the context are notified to the *context client* under the form of *events*. These events can come from the widgets themselves or can be predefined in a *policy script*. For example the triggering of an event `wakeUpTime`, every day at eight. Events can also be triggered by widgets, for example an event `locationChanged`. This policy language and how it can be used is explained by means of the architecture of Chisel.

4.6.2 Chisel's Architecture

Binding *meta-types* to specific classes is performed by the *meta-level adaptation manager* which takes input from a *policy scripting language* and from managed services. The architecture of Chisel is shown in figure 4.4. As we can see in this figure, the framework itself consists of three building blocks. In this section we focus on the innovative approach to use reflection as an adaptation mechanism for context-aware applications. Note that adaptation of the application itself is distinct from the general architecture described in section 4.1.3. First an overview of *reflection* is given and how this can be used in order to adapt applications, then the policy scripting language is highlighted.

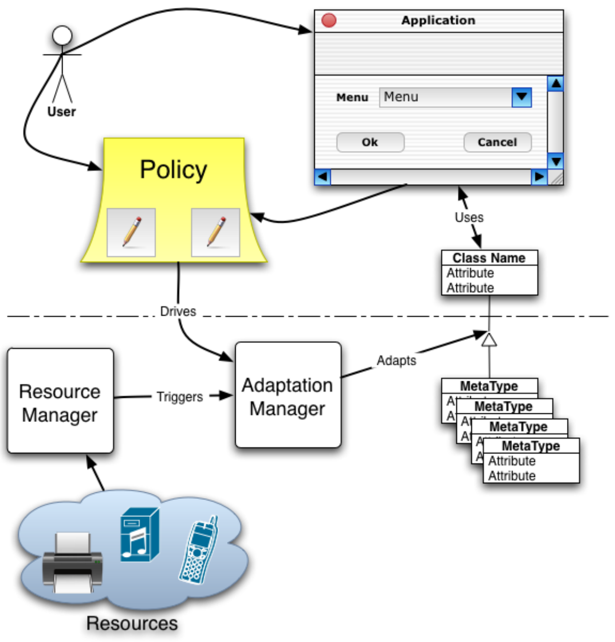


Figure 4.4: CHISEL's architecture [37]

4.6.3 Meta-Types

Computational *reflection* is the ability for a program to reason about and alter itself. Reflection is achieved by reifying the program's state as meta-data. This allows the program to change its own behaviour at a very fundamental level. Examples of such changes include modifying the way variables are looked up, or how functions are called. For class-based object-oriented languages, Shäfer [55] proposed the concept of a *meta-type* as a scoping technique for *adaptation spanning*. Only those objects assigned to this meta-type behave differently than other objects. Meta-types can be assigned to an object at runtime. The interfaces between these meta-types and the object level are referred to the as the *meta-object protocol* (MOP). This MOP allows the user to change the programming language itself by for example changing the way method invocations must be performed. The key concept is to group the non-functional requirements in a meta-type, such as for example logging or the choice of network medium. These meta-types can be implemented by meta-classes providing non-functional behaviour for the class. It's the power of this meta-type that has been exploited to make runtime adaptations. The system used by CHISEL is IGUANA/J [28] which extends the JVM using the JIT interface to provide a meta-object protocol for JAVA.

4.6.4 Policy Driven Adaptation

Central in the CHISEL approach is the usage of an *adaptation manager*. This manager controls the adaptations which must be performed in response to changes in the context. The choice between different adaptations is driven by a *policy*, which is defined as *"a rule governing the choices in behaviour of a managed system"* [15]. Beside the policy, the adaptation manager also receives input from various context resources. The design of the context manager consists of seven components which are depicted in figure 4.5. The rest of this section highlights the functionality and the interaction between these components with the running example.

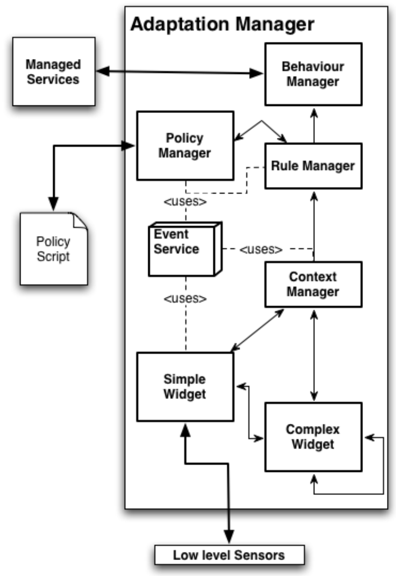


Figure 4.5: CHISEL: adaptation manager [37]

Policy Manager

The *policy manager* is provided with a *policy script*, it constantly keeps track of changes to this file in order to reflect these immediately into rules and events. This ensures that the policy script can be adapted at runtime. The events are passed to the *event service* in order to be processed by the *context manager*. The rules are passed to the *rule manager*.

CHISEL incorporates a description language to specify the policy, this language consists of a series of simple if-then rules which specify which adaptations must be triggered according to the current context. Beside the rules, the language also allows the specification of new events dynamically.

The rule listed below specifies the adaptation of a cellphone when the `LocationChange` event is triggered and the location is the meeting room. This location change is triggered by a location widget. When a new `Event` is triggered, all the rules are checked and the appropriate rules are evaluated. When a rule is satisfied, the behaviour manager is instructed to perform the change in behaviour. In the example shown below, the change in behaviour is to assign `quiteMetaType` to the `CellPhoneIncomingCall` class.

```
ON LocationChange:
    CellPhoneIncomingCall.quiteMetaType
        IF LocationWidget.position = meetingRoom
```

Listing 4.12: CHISEL: *switch profile* example

Context Manager

The *context manager* monitors *widgets* which can be compound to form more complex widgets, very much like the aggregation component of the architecture of the CONTEXT TOOLKIT 4.2. Widgets can trigger events or can be queried directly by the scripting language. When relevant context changes occur, the context manager in conjunction with the rule manager identify the rules affected by this context. When there is a rule which is triggered by this context change this results in an adaptation performed by the behaviour manager. The location widget is a simple widget which is connected directly to the context manager and the event service. When the location changes, it triggers a `LocationChange` event. This informs the `Event Service` that a new event has been triggered.

Behaviour Manager

The *behaviour manager* performs the actual context adaptation by making these changes according to rules, by associating a different *meta-type* to one of the objects it manages. It keeps a repository of the available meta-types and can load new meta-types dynamically at runtime. For example in the *switch profile* example it keeps track of the `CellPhoneIncommingCall` objects, and changes their meta-type to `quiteMetaType` when being located in the conference room.

Rule Manager

The *rule manager* receives input from the policy manager and from the context manager. It combines this information and triggers new adaptations by informing the behaviour manager of a particular adaptation that needs to be performed, given the context and the policy.

4.7 Conclusion

The focus of the Chisel framework is the adaptation of the programs by making use of meta-types.

4.8 Gaia

Gaia is a framework for building context-aware applications [50]. The focus of Gaia lays in the derivation of higher-order context by using rules, or machine learning techniques which have the advantage that no rules should be explicitly specified by humans.

The remainder of this section is organised as follows. First the modelling of context is discussed. Subsequently the architecture of Gaia is explained, followed by a discussion about the deduction of new context. Thereafter the communication between *context service* and *context client* is discussed and as a conclusion we compare Gaia to the other frameworks discussed in this chapter.

4.8.1 Gaia's Context model

The basic structure for the modelling of context is the *context predicate*. The name of the first-order predicates represents the kind of context that is being modelled, like for example location or time. An example of a predicate of type location is `location(Alice, enters, room F134)`. It is also allowed to have relational operators in this predicate as is exemplified by the predicate `time(Brussels, "<", 05/06/07 23:59)`, which states that the time in Brussels is earlier than 05/06/07 23 hours 59 minutes.

Operations

Gaia allows writing more complex expressions by using boolean operations, like for example a conjunction. Listing 4.13 contains examples of those boolean operations. The first example uses the `and` operation and represents the context that Alice is entering room F134 and that there's a meeting taking place at that location. The second line of the listing refers to the context that there is a meeting going on in room F134 or that there's no activity at all in that room. The last example uses the `not` operation and refers to the context that Alice is not in room F134.

```
location(Alice, enters, room F134) AND activity(meeting, in, room F134)
activity(meeting, in, room F134) OR activity(none, in, room F134)
NOT location(Alice, in, room F134)
```

Listing 4.13: Gaia: using boolean operations

Furthermore, Gaia also enables the use of existential and universal quantifiers. Example of context expressions using quantifiers are given in listing 4.14. The first expression of listing 4.14 refers to the context that there's at least one person who's in room F134. The context that is represented by the other expression is that all the people are present in room F134.

```
∃person  location(p, in, room F134)
∀person  location(p, in, room F134)
```

Listing 4.14: Gaia: using quantifiers

4.8.2 Gaia's Architecture

This section describes the different components of the architecture of Gaia's framework. Each component is highlighted and compared in relation to the components of the other context-aware frameworks discussed previously.

Example

Reconsider the example application where the user's mobile phone should be switched to silent whenever he is entering a room where a meeting is taking place on a weekday. There are two widgets needed for retrieving both the location and date contextual information. The context provided by the `location` and `iCal` *widgets* can be combined by an aggregator. An interpreter can be used for deducing higher-order context information based on the context information the aggregator has combined. Both functionalities of the aggregator and interpreter are realised by Gaia's *context synthesiser*. The `Profile` *context client* adapts its behaviour according to the current context, retrieved by the widgets. Context can be obtained by querying the widgets or by listening for events that are sent by them. The architecture for this example application is depicted in figure 4.6. The remainder of this section explains all the components of the architecture in more detail.

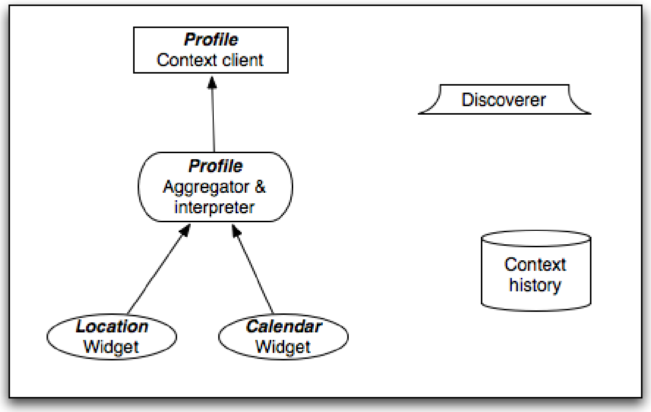


Figure 4.6: Gaia context infrastructure [50]

Context Provider

A first component of the GAIA's architecture are the *context providers*, which is modelled by widgets in the architecture explained in section 4.1.3. Widgets collect several types of context information and allow context clients to access them. How this access can be realised is explained in section 4.8.4.

Context Synthesiser

Another component of the architecture is the *context synthesiser*, which is a combination of both the aggregator and interpreter of the common structure that are components of the CONTEXT TOOLKIT's architecture. First of all, this component is responsible for inferring higher-level context based on sensed context. This sensed context information can be retrieved from several widgets and combined by an aggregator. There are two approaches for the derivation of higher-level context that can used, namely *rules* or *machine learning techniques.* Both approaches are discussed in section 4.8.3.

Context Provider Lookup Service

The *context provider lookup service* of GAIA's architecture represents the discoverer that is a part of the architecture of the CONTEXT TOOLKIT (discussed in section 4.2.2). The discoverer allows finding widgets and advertising the set of context those widgets provide. When a context client queries the discoverer, the discoverer tries to find a widget that provides the context the client needs and returns the results to the appropriate context client.

Context Consumer

Another component of the architecture is the *context consumer* which models the context client that is discussed in section 4.1.3. Context clients retrieve several types of contextual information and adapt their behaviour depending on this context. The actual retrieval of this context information is explained in section 4.8.4.

Context sensitive clients are built by defining a set of rules that govern the behaviour of the context client according to changes in the context. Context clients have an interface which enables the changing or addition of new rules dynamically. Context sensitive behaviour can be specified in a configuration file. Listing 4.15 is an example of a configuration file that specifies different actions to be taken when someone enters a room. Each context client has such a configuration file that associates a specific context situation with a set of methods that are invoked whenever that context situation is detected.

The configuration file in listing 4.15 specifies the behaviour when someone is entering the room. A first scenario says that when someone is entering a certain room a welcome message should be displayed on his mobile phone. The method `ShowWelcomeMessage` that is specified in the configuration file should be implemented by the application developer. This context is associated with a priority equal to one, as can be seen in the configuration file. The second context specifies that whenever someone enters a room where the light is turned off, a welcome message should be displayed as well as the turning on of the light in that room. The priority associated with this context equals two. So, for example when someone is entering a room and the light is turned of, the welcome message is only displayed once, because only the actions of the context with the highest priority are executed. The last context specifies that whenever someone enters a room where a meeting is taking place, a welcome message should be displayed and the profile of that person's mobile phone should be switched to silent. This context is associated with a priority equal to three.

```
∃_Person p ∃_Room r location(x, entering, r)
      ShowWelcomeMessage()
      Priority 1

∃_Person p ∃_Room r location(x, entering, r)
AND lightning(r, off)
      ShowWelcomeMessage()
      TurnLightOn()
      Priority 2

∃_Person p ∃_Room r location(x, entering, r)
AND activity(meeting, in, r)
      ShowWelcomeMessage()
      SwitchToSilent()
      Priority 3
```

Listing 4.15: GAIA: example configuration file specifying different behaviours

Context History

GAIA enables context clients to not only using the current context but also to past contexts. Context is stored in the *context history* by using a timestamp that indicates when this context was used. It is possible for context clients to query the context history about such past contexts.

4.8.3 Context Synthesiser

The architecture of GAIA resembles the one of the CONTEXT TOOLKIT. This section discusses GAIA's *context synthesiser* because it differs from the interpreter used by the CONTEXT TOOLKIT for inferring higher-level context. As was already mentioned before, this component of the architecture is a combination of both the aggregator and interpreter of the CONTEXT TOOLKIT. This component is able to infer new context by either using rules or machine learning techniques. We explain both approaches but emphasise the part of inferring higher-level context by using rules. This relates to the Fact Space Model, described in chapter 5, because it also uses rules for deriving new context information.

Infer New context Using Rules

Using rules for deriving higher-order context is quite simple by using the first-order predicates and its operations that are described in section 4.8.1. Listing 4.16 shows an example rule. The first one deduces that there's a meeting taking place in a certain room when the mobile phone's profile of all the persons in that room is silent.

```
∀_Person p ∃_Room r location(p, in, r)
     AND profile(p, silent)
     => activity(meeting, in, r)
```

Listing 4.16: Gaia: inferring new context using rules

Sometimes it might be the case that several rules can be true at the same time and that multiple answers are returned. For example, it is possible that the activity in that room can be either meeting or none at all. Both rules are presented in listing 4.17.

In case only one answer is preferred, *conflict resolution* is applied. This can be realised by giving each rule a certain priority and only deducing the higher-level context of the rule with the highest priority.

```
1. ∀_Person p ∃_Room r location(p, in, r)
     AND profile(p, silent)
     => activity(meeting, in, r)
2. ∀_Person p ∃_Room r location(p, in, r)
     AND profile(p, silent)
     => activity(none, in, r)
```

Listing 4.17: Gaia: using rules with an identity

Infer New Context Using Machine Learning Techniques

The previously described approach for inferring higher-order context is based on using logical rules. Using rules for deducing new context has the disadvantage that these rules must be specified by humans. Introducing machine learning techniques results in a flexible way for adapting on changes of context.

For example, the sound of the profile of a user's mobile phone is context that can depend on several contexts, like the mood he is in, the location where he finds himself and possible some other context information as well. Gaia allows the prediction of what sound the person wants to hear at a certain moment by making use of machine learning techniques as the Bayesian algorithm.

4.8.4 Communication

Context clients can obtain context information either synchronously by using the *query-answer* protocol, or asynchronously by using the *subscribe-notify* protocol. The first method for the context retrieval is used by context clients that want to obtain the current context and perform services for the user adapted to that current contextual information. The latter approach is suitable for context clients that want to perform certain actions on context changes. Both approaches are highlighted in this section.

Query-Answer Protocol

Context clients can obtain context information by sending a query to the widget. The notation of a query resembles the one Prolog uses [62]: one or more fields of the context predicate is a variable. In GAIA variables are denoted with a question mark in front of the variable's name, so for example the variable person is notated as `?person`. For instance, when a context client wants to know what activity is taking place in room F134, it could send the query that is shown in listing 4.18 to the widget that provides context of type `activity`. This approach for retrieving context is suitable when the context clients want to adapt to the current context.

```
activity(?activity, in, room F134)
```

Listing 4.18: Query to widget that provides context of type "activity"

Subscribe-Notify Protocol

The second approach for retrieving context information is by letting the context client subscribe to those widgets that produce interesting context for that client. So, whenever such a context is sensed by the widget, it notifies the registered context clients. Each widget produces its own events, so every context client can listen to the events of the widgets in whose context it is interested.

4.8.5 Conclusion

GAIA uses a powerful mechanism to coordinate by using first order logic. However it fails to provide this mechanism to the application in order to react asynchronously on context changes.

4.9 CORTEX

CORTEX is a research project – from 2000 until 2004 – which states the following as general objective, as stated on their homepage [2]:

> "The CORTEX project will investigate appropriate architectures and paradigms for the construction of applications composed of collections of what may be called sentient objects – mobile intelligent software components that accept input from variety of different sensors allowing them to sense the environment in which they operate before deciding how to react."

[2] *http://www.dsg.cs.tcd.ie/dynamic/?category_id = −12*

4.9.1 CORTEX Architecture

This section explains the CORTEX architecture. First of all, we reconsider the example application and present the architecture CORTEX uses for implementing this problem. Thereafter the components of the architecture are described in more detail.

Example

Reconsider the *switch profile* example application that is already used before, the profile of a user's mobile phone should be set to silent whenever he enters the meeting room at a weekday. So, the `location` and `iCal` sensor are needed for providing the accurate context information. This contextual information is retrieved by the *sentient object* which is able to combine both sensed contexts and to infer higher-order contexts from the current context information. This new context is then sent to the `Profile` *actuator* which switches the profile of the mobile phone to silent. The architecture used by CORTEX is depicted in figure 4.7. The remainder of this section describes the components in more detail.

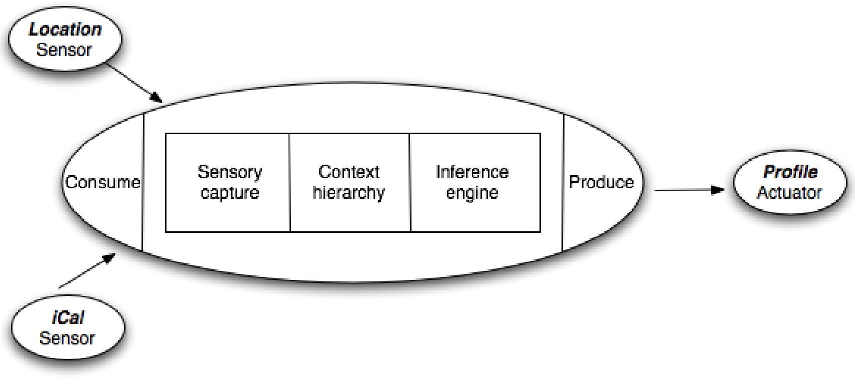


Figure 4.7: CORTEX: sentient object model [60]

Sensor

A first component of CORTEX's architecture is a *sensor* which refers to the widget that is discussed.

Sentient Object

The sentient object is the core of the CORTEX architecture. It takes input from the widgets and steers the actuators. The benefits of introducing this extra layer is that an inference engine is encapsulated in this component, which derives valid higher-order context by combining input retrieved by the sensors. How this is performed is shown in section 4.9.2.

Actuator

The last component of the architecture of CORTEX is the *actuator* that is responsible for consuming the retrieved context.

4.9.2 Sentient Object

As is already mentioned, the *sentient object* combines the functionalities of the *aggregator*, *interpreter* and *context service* that are components of the architecture of the CONTEXT TOOLKIT (which are discussed in section 4.2.2). The sentient object itself exists of three subcomponents that provide the functionalities of the according components of the CONTEXT TOOLKIT's architecture.

Sensory Capture

The first subcomponent of the sentient object is the *sensory capture* that deduces higher-order context based on the sensed contextual information. Furthermore, this component performs *sensor fusion* for managing the uncertainty of the sensed data. The sensor fusion that is used by CORTEX is based upon *Bayesian networks* which enables a probabilistic measurement of derivations of context from possibly noisy sensor data. So, this subcomponent refers to both the interpreter component of the CONTEXT TOOLKIT's architecture, and the aggregator component that combines the context of different widgets. The input of a sentient object can be another sentient object. Note that the sentient object itself can be seen as an aggregator combining different widgets and passing further this derived context to another sentient object.

Context Hierarchy

Context hierarchy forms a second subpart of the sentient object which encapsulates the current context derived from the *sensory capture component*. This encapsulation is based on the *context-based reasoning paradigm* that states that context determines the actions to be performed. So, the behaviour of the sentient object is influenced by the context it retrieves from the sensors widgets. This subcomponent, explicitly keeping track of the current context, isn't represented in the architecture of the CONTEXT TOOLKIT.

Inference Engine

The last subcomponent of the sentient object is the *inference engine*. This part of the sentient object ensures its context-awareness by using conditional rules. This subcomponent is responsible for the changes in the behaviour due to the currently sensed context information. So, it can be regarded as a context service that is explained in the common structure (section 4.1.3). When comparing it with the components of Gaia's architecture, the inference engine maps to the context synthesiser which could infer higher-order context by using rules or machine learning techniques, as is explained in section 4.8.3. The inference engine is based upon CLIPS [27] which uses the Rete algorithm that is described in chapter 3. This algorithm is also used for the implementation of Crime that is explained in chapter 6.

A example rule for switching the mobile phone is shown in listing 4.19. The `Switch-Phone` action is called when the user is in the conference room during a weekday. The `Switch-Phone` action is an extension specified by the programmer and is not part of the CLIPS language itself.

```
(defrule rule
  (location (room  conferenceRoom))
  (day (weekday true))
 =>
 (Switch-Phone silent)
```

Listing 4.19: CORTEX: *switch profile* rule

4.9.3 Conclusion

CORTEX proposes the use of a sentient object which transforms the input from widgets to actions performed by actuators easing the development of mobile intelligent software components by providing a declarative language for the derivation of higher-order information. By making use of a declarative programming language the flexibility increases as the logic itself must not be implemented.

4.10 Cocoa

This framework, Coordinated Context Awareness [5], is inspired by biology and more precise by the phenomenon *Stigmergy* [29]. A first observation is that insects in a colony can communicate with eachother by using the environment, and thus they do not need direct contact with eachother in order to communicate. A second important observation is the fact that the communication between two local insects can have a large propagation through the entire population resulting in a global change of behaviour. Holland [3] noted that the state of the environment and the entities in this environment determine the future changes of the environment and the entities within. The idea of using Stigmergy as a coordination mechanism has been adapted in various projects [10], [36]. In the Cocoa framework the idea of Stigmergy has been formalised in the following equation [5]:

$$C_{V_{e_n}}(t) = \{C_{e_i}(t) : C_{e_i} \in C_G(t) \wedge L(e_i, e_n) = true\} \tag{4.1}$$

4.10.1 Cocoa's Context Model

Equation 4.1 defines the *contextual view* of a single entity e_n in the environment at a particular time t. Not the entire available context is important for this entity, therefore only a subset from the global context $C_G(t)$ is relevant, namely the contextual views of all entities e_i within a certain proximity of e_n defined by the proximity function L.

Now that the contextual view for an entity e_n is defined, the entity should also be able to change its behaviour according to $C_{V_{en}}$. This mapping between a certain context and a powerset of the set of behaviours B is defined by the function S [5]:

$$S : C_{V_{en}} \to P(B) \tag{4.2}$$

4.10.2 Cocoa's Architecture

The Cocoa architecture is split up into two large parts, namely context acquisition and the stigmergy runtime which uses this context.

Stigmergy Runtime

The stigmergy runtime is driven by a Yabs script, this language is defined in various files, together making a hierarchy. This hierarchy is defined by extending existing objects.

```
woman extends person { /* body */ }
```

Listing 4.20: Yabs script

The language has three primitives, namely the *proximity function L*, *behaviour set B* and the *mapping function S*. The behaviour of an entity can be modelled in Yabs using these three primitives.

Proximity Function L Yabs can handle proximity in three different ways: A first possibility is to define a circular proximity (line 1). This means that the context of entities that reach within a certain distance is shared among those entities. This might not suit every application: for example, rooms are most likely not to be circular. Therefore, another way for defining the proximity is to describe it by specifying a polygon (line 2), this allows the programmer to clearly define the boundary of a certain environment. A third possibility is the abstraction of the environment by using a symbolic name (line 3), this allows the programmer to shield the application from entities which are close by, but which do not participate in the current context. The different specifications of the proximity function are shown in listing 4.21.

```
1  proximity(10)
2  proximity(-1, 1, 1, 1, 1, -1, -1, -1)
3  proximity(PROG_LAB)
```

Listing 4.21: Yabs proximity function

Behaviour Set *B* The definition of a behaviour can not be implemented in the scripting language itself, however the COCOA framework specifies an API which can be used to implement a specific behaviour. In the script a mapping between the JAVA implementation and the behaviour used in the rest of script can be realised as is shown in listing 4.22. This binding can of the behaviour onto a specific JAVA implementation can then later be used in the mapping function.

```
behaviour ring_on = "edu.vub.example.ringOn"
behaviour ring_off = "edu.vub.example.ringOff"
```

Listing 4.22: YABS behaviour definition

Mapping Function *S* The mapping functions is responsible for the mapping between a certain *conceptual view* and a behaviour. Before such a mapping can be established, both the context and the behaviour must be specified. The specification of the behaviour is already explained in the previous paragraph. The specification of context is realised by the specifying a number of predicates as shown below in listing 4.23.

```
context in_meeting
location_widget.location = "conferenceRoom"
time_widget.time = weekday
```

Listing 4.23: YABS context definition

In the example in listing 4.24 we specify a context where the location widget indicates that we are in the conference room and the time widget indicates that it's a weekday. Now that we have specified a certain context, we can use this context to specify a mapping.

```
map[in_meeting] onto {
   ring_off()
}
```

Listing 4.24: YABS mapping function

This mapping ensures that our mobile phone does not ring when we are in the conferenceRoom during a weekday.

4.10.3 Conclusion

The context model of COCOA defines the conceptual view as union of all contextual views of the entities in within a well defined distance. The ability to share contextual information between entities and react upon these by specifying the reactions in a scripting language provide by COCOA make it a powerful framework. However, like CORTEX, it does not give a meaningful semantics to the retraction of information.

4.11 Conclusion

The classic coordination frameworks such as the Context Toolkit, JCAF, and WildCAT are all based upon the notion of event channels. None of these toolkits allow reacting on the retraction of information. The sophisticated adaptation mechanisms adapted in the Chisel framework 's even-driven approach lacks support to adapt in respect to a combination of events. It's also not clear how context should be distributed. Gaia uses a powerful mechanism to coordinate by using first order logic, however it assumes that connections are stable which is in conflict with the nature of a mobile setting. CORTEX extends Gaia by using instead of first order logic, a production system CLIPS that is based upon forward chaining. However, they do not provide a meaningful semantics to the retraction of information. Cocoa uses the idea of Stigmergy for coordination. Coordination is driven by a scripting language but like CORTEX fails to give a meaningful semantics to the retraction of information. An overview of the different systems discussed is shown in table 4.1.

Context System	**Context Model**	**Architecture**	**Context acquisition**
Context Toolkit	Widget outputs	Based on GUI	Event channels
JCAF	Relations	Context services	Event channels
WildCAT	Widget outputs	URI	Event channels
Chisel	Widget outputs	Meta-Types	Logic Language
Gaia	Predicates	MVC	Predicate logic
Cortex	Sensor outputs	Sentient object	Logic language
Cocoa	Contextual view	Stigmergy	Stigmercy

Table 4.1: Context systems: summary

Chapter 5

Fact Space Model

This dissertation advocates the use of a distributed reasoning engine in combination with a truth maintenance system in order to ease the development of context-aware applications. Previous chapters have given an overview of the tuple space model, reasoning engines, and context-aware systems.

This chapter proposes the Fact Space Model, a model which combines the latter three concepts in order to provide fine-grained support to deal with the effects of disconnection and allow reactions to be specified on multiple events. Coordination in the Fact Space Model is based on the use of the federated space known from LIME [44], however in our model it is conceived as a distributed knowledge base.

Entities in the Fact Space Model perceive the environment by available transiently shared facts which can be used to adapt an application's behaviour accordingly. The conditions to adapt an application are described by making use of a logic coordination language whose rules record the causal link between facts and the conclusions that may be drawn from them. Fact spaces differ from tuple spaces as the latter do not record causal links. These links are used in the Fact Space Model to reverse the effects a fact had on the system, when the fact is retracted. Whereas in the Tuple Space Model only the presence of a tuple is relevant, in the Fact Space Model both the assertion and the retraction of facts have consequences. As facts are retracted when a device that published them disconnects, the Fact Space Model offers fine-grained support to deal with the effects of disconnection. By making use of rules which define a logical relationship between several facts residing in the fact space, reactions on multiple events can be expressed more naturally than in LIME where there is a one-to-one mapping between the events and reactions. Reactions in LIME are triggered by the change of only one context event. In order to react on multiple tuples in LIME, multiple event handlers must be installed and the combination logic must be hard-coded into these event handlers.

The remainder of the chapter is organised as follows: the next section gives an overview of the different components of the Fact Space Model. Subsequently, section 5.2 describes CRIME (Consistent Reasoning in a Mobile Environment), a prototypical implementation of the Fact Space Model in which we have conducted our experiments. A selection of these experiments is then presented in section 5.3, among others a smart jukebox application.

5.1 The Fact Space Model

The Fact Space Model is a coordination model for mobile applications communicating over a mobile ad hoc network. It offers mobile applications a consistent view on their environment which can be used to provide additional or smarter behaviour. The view of an application on its environment consists of facts published in a *federated fact space*. Concretely, facts are locally published by applications and transparently shared between nearby devices as long as they are within communication range. Applications have the ability to react upon the appearance of facts, by making use of rules specified in the *logic coordination language* offered by the Fact Space Model. These rules can be seen as a mapping from (a combination of) facts onto a conclusion. Conclusions may consist of the addition of new facts to the fact space or the execution of *application-specific actions*. In contrast to LIME, the Fact Space Model gives applications the ability to react upon the retraction of an action which is achieved by the specification of a compensating action which is triggered when the fact is retracted. As the view of an application's environment is kept consistent by automatically retracting the facts of nearby devices when they disconnect, the Fact Space Model provides fine-grained control over the effects of disconnection. The remainder of this section provides a more detailed explanation of both the federated fact space and the logic coordination language.

5.1.1 Federated Fact Spaces

The view an entity has of its environment consists of all published facts by all entities within reach. Facts can be used to represent various types of information ranging from physical information over the availability of certain services up to task scheduling information. The transparent distributions of these facts is ensured by making use of a federated space, as originally proposed in LIME [44]. The difference with LIME'S federated tuple spaces lays within the perception of the federated space as a knowledge base containing facts. As a consequence of this difference in perception, not only the assertion but also retraction of facts is a meaningful event which may be used as an indicator to adapt the application at hand.

The Fact Space Model equips applications with at least two fact spaces, a *private fact space*, and one or more *interface fact spaces*. Facts residing in the private fact space are not shared, whereas facts residing in an interface fact space are exchanged with other applications in connection range. Consider the *switch profile* example used in previous chapters, where the main concern is to change the profile according to the user's location. In this example the facts modelling the user preferences could be located in the private fact space, as

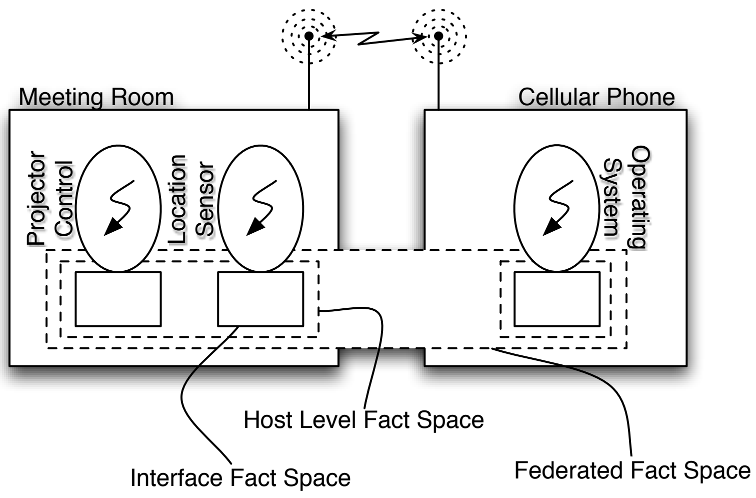


Figure 5.1: A federated fact space

this information is only used by the application itself. Detecting the location of a mobile phone is typically derived from the information published by a device in earshot, for example a RFID scanner. Facts providing the location information must be published into an interface fact space (and thus shared) as shown in figure 5.1. All interface tuple spaces are aggregated into a *host level fact space* which provides applications with the ability to use contextual information derived by other applications. For example one application could derive the location information, which then can be used by all applications on the same device. Moreover, coordination between applications residing on the same devices can also be modelled by making use of the host level fact space. The host level fact spaces of devices which are currently in range are aggregated into the *federated fact space*.

The discovery mechanism of the Fact Space Model is adopted from Lime: when two devices discover eachother, their host level fact spaces are atomically and transparently merged into a federated fact space. As facts are exchanged and asserted during this merging process, applications can possibly adapt themselves to their newly sensed environment. As the sharing mechanism of the Fact Space Model is based upon connectivity, when a device disconnects all facts that where shared by this device are retracted from the host level fact spaces of the devices it was previously connected with. Again, the coordination language offered by the Fact Space Model provides mechanisms to provide fine-grained control over these disconnections as is explained in the next sections.

5.1.2 Logic Coordination Language

The federated fact space described in the previous section ensures that the view of an application on its environment is kept consistent by translating changes in the environment into the assertion or retraction of facts shared by co-located devices. Here we present a minimal logic coordination language giving applications the ability to react on changes in their environment by making use of rules.

Consider an example rule which collects all online printers residing in the environment into a list of available printers, furthermore those printers should have a dpi which is at least 300. The rule written in the logic coordination language offered by the Fact Space Model is illustrated in listing 5.2. The rule must be interpreted as follows: trigger the application-specific action `addToPrinterList` *if and only if* the `public` federated fact space contains a `printer` fact whose `dpi` is at least 300.

```
trigger action addToPrinterList(name, ip) if
        public fact printer(name, dpi, ip) and
        dpi >= 300.
```

Figure 5.2: Rule to add printer functionality to an application

Like the Tuple Space Model explained in chapter 2, the Fact Space Model can be implemented in a variety of languages. In order to make a clear separation between the model and our implementation of the Fact Space Model, namely the language CRIME, pseudo code is used to explain the logical coordination language provided by the Fact Space Model itself. As can be seen in the example rule of figure 5.2, logic variables are written in italics. Actions are written in the base language implementing the Fact Space Model, e.g. in the language **C** this could be a function `addToPrinterList` expecting two variables. In a class-based object-oriented language this could be a method `activate` from the `addToPrintList` object. Finally, as can be seen in the example, facts can be *quantified* to denote the fact space in which they should be found or asserted. This *quantification* allows the use of multiple fact spaces similar to the multiple tuple spaces as seen in dialects of LINDA, however in the Fact Space Model fact spaces are not first class. The intention of this design decision is to avoid the problematic semantics of first class tuple spaces as discussed in chapter 2.

The distinction between the rule specified above and a similar *reaction* in LIME, is that the former implicitly provides a hook to respond to the disappearance of a fact. This is achieved by the enforcement that any custom actions also provides a compensating action. For example the compensating action for `addToPrinterList` in **C** could be implemented by a function `addToPrintList_Compensate`. In a class-based object-oriented language this could be the invocation of a `deactivate` method. Note that compensating actions are not required to restore the application's state prior to the execution of the `activate` method.

Consider the example to switch the profile of a cellular phone depending on its location. One solution for this example by making use of the Fact Space

```
private fact room(meetingRoom, silent).
private fact room(office, general).

trigger action switch(?profile) and profile(?profile) if
  public fact location(myID, ?room) and
  private fact room(?room, ?profile).

trigger action switch(default) if
  not profile(?p).
```

Figure 5.3: Fact Space Model: facts and rules to change the profile of a cellular phone

Model is shown in Figure 5.3. In this example, the preferences concerning the wanted profile are modelled by making use of a set of private `room` facts. Concretely the user in the example wants his profile to be silent in a meeting room and general in an office room.

Furthermore, the example consists of a rule which specifies to execute the `switch` application-specific action as well as adding a private `profile` fact when the rule is triggered. The rule is triggered when the cellular phone enters a particular room for which the user has made his preferences clear, in the example, a meeting room and an office. To finish the example, a rule is provided which switches the profile of the cellular phone to `default` when no explicit `profile` is prescribed[1].

As the rule reacts on the presents of both a room **and** location fact, the same implementation in LIME would require the and condition to be hard-coded in the reactions. The latter exemplifies the strength of using a separate coordination language concerned about the contextual knowledge, where the reaction on multiple events is incorporated in the language itself.

5.2 Crime

This section gives an overview of the logic coordination language CRIME (Consistent Reasoning in a Mobile Environment) which is an experimental implementation of the Fact Space Model. CRIME has been used to gain experience into modelling and building context-aware software using the Fact Space Model for its coordination. We start this section by first giving an overview of the basic syntax of the language. Subsequently, some of the key points of the implementations design are highlighted, before presenting some typical examples using accumulation techniques borrowed from PROLOG.

[1]One may have notice that this rule is not strictly necessary, the same behaviour can also be achieved using the compensating action of `switch`.

5.2.1 Crime Syntax

This section describes the syntax used by Crime which resembles the syntax adopted by the logic programming language Prolog. The basic language constructs of the Crime language are *facts* and *rules*, for respectively representing and reasoning about context information. The remainder of this section is organised as follows: first of all, the syntax of the facts is presented, followed by those of rules.

Crime's Facts

A Crime fact consists of a *type* and one or more arguments. Facts represent contextual knowledge and reside in the fact space of the application. The type of a fact is used to specify the meaning of the fact, similar to the name of a class or method in class based object-oriented languages. Consider the fact shown on the first line of listing 5.1, the type of this fact is `userInfo`. This specifies that the information, expressed by the fact's arguments, represents user information. The second fact, `location(75773, Kitchen)`, represents contextual information of a person's location.

As is already mentioned in section 5.1.1, Crime allows facts to be ascribed to a specified *fact space*, called *quantified facts*. Whenever a fact is not quantified, it is considered to belong to the default fact space, which is *private*. Facts belonging to this *private fact space* are not exchanged between co-located devices. Consider the two facts shown in listing 5.1: the first fact isn't ascribed to any fact space so it belongs to the private one. The second fact is to be placed into the `public` fact space and is therefore shared with all applications residing on co-located devices. Note that this includes other applications running on the same device as the application that asserted the fact.

```
userInfo(Alice, 75773)
public -> location(75773, Kitchen)
```

Listing 5.1: Crime's facts

Crime's Rules

Crime provides *rules* in order to reason about the current context which is represented as facts maintained in a federated fact space. A rule consists of several prerequisites and consequences. A rule is *triggered* whenever all its prerequisites are met, when this occurs the rule's consequences are executed.

The prerequisites of a rule consist of facts possibly containing one or more variables. Variables are denoted as a string preceded with a question mark. Consider for example the rule starting on line 1 in listing 5.3. This rule has two prerequisites that must be fulfilled in order to execute the rule's consequences. The first prerequisite, `public` $-$ $>$ `location(myID, ?room)` represents that a fact with type `location` and with a first argument equal to `myID` should be part of the `public` fact space. There is no constraint on the fact's second argument. The second prerequisite of the rule describes that there should be a fact of type `room` in the private fact space. No other constraints are specified for this second fact. However, an implicit constraint is expressed by using the same variable `?room` in two prerequisites which state that the variable bindings in those prerequisites must be consistent.

The consequences of a rule consist of one or more actions, which can even be user-defined. A colon prefix is used to differentiate between *application-specific actions* and logic facts. Whenever the consequence is represented as a fact, the implicit action is to add the fact to its denoted fact space, when no fact space is specified it is added to the private fact space. A consequence can also be a user-specific action to be executed. For instance the consequence on the first line in listing 5.3 is a user-specific action that implements the `switch` operation on the user's cellular phone. The user-specific switch action from the rule denotes implicit method invocations on the `switch class`. This custom action switch is enforced to inherit from the abstract class `Action`. Subclassing from this class implies that the switch class must implement an `active` method which is invoked when the action is derived, as well as a `deactivate` method which describes a compensating action to be performed when the action is retracted. The implementation of this switch action is shown in listing 5.2. When the `activated` method is invoked it sets the profile of the mobile phone by making use of the arguments passed to it, whenever the `deactivate` method is called, it switches the profile back to the default profile.

```
public class Switch extends Action {
  public static MobilePhone phone;

  public void activated(Vector args) {
        Profile profile = new Profile(
        args.elementAt(AttributeValue.PROFILE));
        phone.switch(profile);
 }

 public void deactivate(Vector args) {
        phone.switch(Profile.DEFAULT);
 }
}
```

Listing 5.2: Crime: application-specific action

Consider the two rules shown in listing 5.3. The first rule, which starts on line 1, switches the profile of a user's cellular phone whenever the user has a preferred profile specified for the room where he's located at the moment. Furthermore, the second consequence of the rule publishes a fact of type `profile` in the user's private fact space. The second rule (line 6-7) is triggered when the user has no private fact `profile`. In that case the profile of his mobile phone is switched to the default profile.

```
1  :switch(?profile) ,
2  profile(?profile) :-
3          public -> location(myID, ?room),
        room(?room, ?profile).

:switch(default) :-
        not profile(?p).
```
Listing 5.3: CRIME: *switch* rule

5.2.2 Federated Fact Spaces

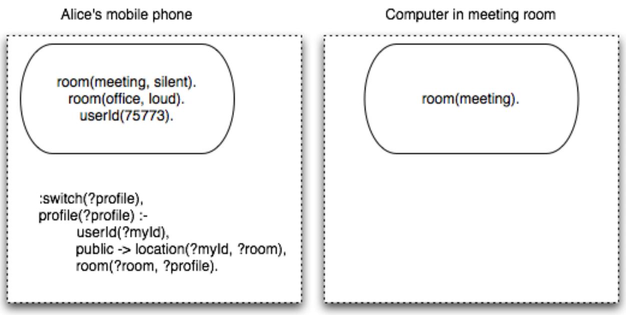
CRIME's implementation of the federated fact spaces that is described in section 5.1.1 is achieved by a distribution architecture similar to the one used by LIME, which is discussed in chapter 2. Recall that LIME uses three primitives as its interface tuple space, namely *read*, *in* and *out*. LIME's **read** operation reads a tuple from the tuple space, whereas its **in** operation also removes the read tuple from the tuple space. These operations are omitted by CRIME as every fact in the fact space is implicitly read when using prerequisites in a rule: this makes an explicit read operation in CRIME obsolete. The **out** operation of LIME publishes a tuple in the tuple space, while this behaviour is supported by CRIME's **assert** operation.

Example

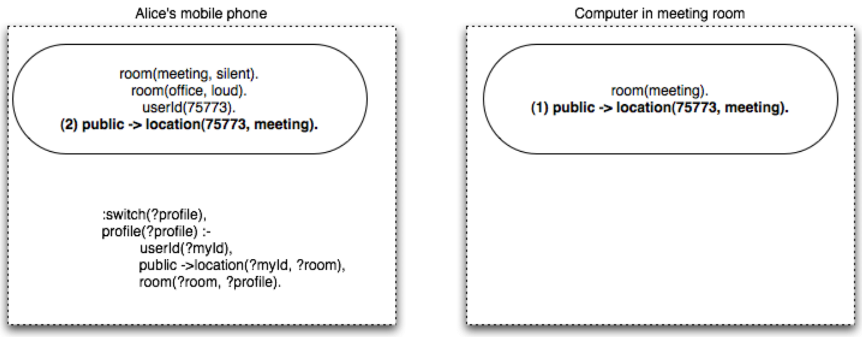
In this subsection we explain the use of the federated fact space in CRIME by means of an example. Consider an application where a user wants his mobile phone to be switched to his preferred profile when entering a certain room. Alice doesn't want to be disturbed when she's participating in a meeting, so the fact space residing on her cellphone contains a fact `room(meeting, silent)`. Furthermore, the fact space contains some other non-quantified facts, as other preferred profiles and her user identifier. This fact space is depicted in figure 5.4(a). Alice's mobile phone has a CRIME application running which performs the profile switching. The rule of this application is also depicted in the figure. The second part of the picture represents the fact space of a computer standing in the meeting room.

Suppose Alice enters that meeting room. Her presence is detected by a widget and the computer in the room publishes a fact dedicating that the person with user identifier `75773` is present: `public` $->$ `(location(75773, meeting)`. This is depicted as step `(1)` in figure 5.4(b). The detection of users is explained in more detail in section 5.3.1. As Alice's cellphone is co-located with the computer in the meeting room, all quantified facts are exchanged between the two devices. So, as step `(2)` represents, the published fact is added to the fact space residing on Alice's phone.

The addition of the fact `public` $->$ `location(75773, meeting)` triggers the rule of Alice's application. The rule's prerequisites are all met and the consequences are executed. First the application specific action `:switch` is executed, which changes the profile on Alice's mobile phone. The second action



(a) Initial configuration

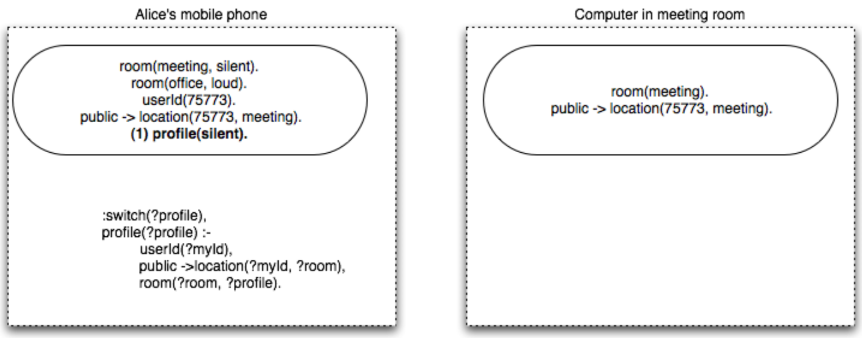


(b) Alice enters meeting room

Figure 5.4: Fact spaces of *switch profile* rule (1)

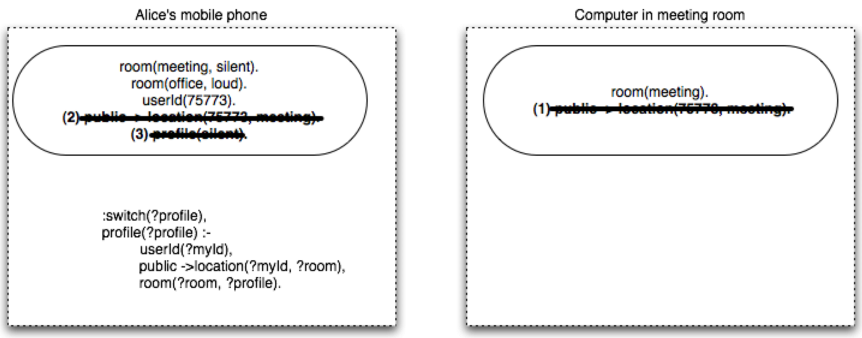
asserts a fact `profile(silent)` to the fact space residing on her phone. These steps are depicted in figure 5.5(a)

Now, when Alice leaves the room, the context widget detects her absence and the fact `public` $->$ `location(75773, meeting)` is *retracted* from the fact space of the computer in the room. As Alice's mobile phone and that computer are no longer in earshot, the exchanged facts are retracted which causes the retraction of that fact in the fact space residing on Alice's mobile phone (step `(2)` in figure 5.5(b)). The retraction of these facts is caused by the *disconnection* of the two devices, namely Alice's cellular phone and the computer. Furthermore, the prerequisites of the CRIME rule are no longer met and all consequences must be retracted. So, as step `(3)` in the picture indicates, the published fact `profile(silent)` is retracted from the fact space as well.



```
:switch(?profile),
profile(?profile) :-
        userId(?myId),
        public ->location(?myId, ?room),
        room(?room, ?profile).
```

(a) Profile is switched to silent



```
:switch(?profile),
profile(?profile) :-
        userId(?myId),
        public ->location(?myId, ?room),
        room(?room, ?profile).
```

(b) Alice leaves meeting room

Figure 5.5: Fact spaces of *switch profile* rule (2)

Persistent Facts

Persistent facts are facts which remain in other fact spaces, even in case of disconnections. Crime does not have native support for persistent facts, however a custom action `:persistent` can easily be written. The idea is that the activate method of this action inserts a private fact into the local fact space of the application. However the `deactivate` method of the persistent class does not retract this fact. This behaviour ensures that this private fact is not removed when the device that published the `public` fact disconnects.

```
:persistent(?args) :-
        public -> fact(?args).
```

Listing 5.4: Crime: *switch* rule implementing persistent fact

5.2.3 Logic Coordination Language

The Fact Space Model of Crime provides a logic coordination language for reasoning about context information that is represented as facts in a federated fact space. This logic language uses the forward chaining strategy for deriving new conclusions as this data-driven technique is very suitable for the event-driven nature of Crime. A first paragraph of this section discusses both derivation strategies and addresses the choice for using forward chaining for Crime's implementation. A second important implementation choice of this logic coordination language is maintaining the truth. Truth maintenance is very important in mobile environments as stable connections can not be taken for granted and disconnections are frequently occurring. When two devices go out of earshot, not all derived conclusions and exchanged facts are guaranteed to be true. The problem of maintaining the truth is discussed in a second paragraph of this section. A third paragraph introduces extra constructs borrowed from the declarative language Prolog.

Inference Engine

Backward chaining is a *resolution strategy* which can be used to allow applications to reason over a stable distributed knowledge base describing the information for one application. This restriction is due to the fact that taking into account new information implies that the inference engine must be restarted explicitly and the derivation process must be restarted from scratch. Restarting the derivation process everytime a new fact is asserted to the fact base is costly and time-consuming. Therefore, the use of backward chaining inference engines should be restricted to the case where the facts in the distributed knowledge base are related to a single application. In these cases, it is most likely that a change in the knowledge base is reflected in the outcome of the reasoning process.

The use of a *forward chainer* on the other hand is useful when applications need to reason over a fluctuating distributed knowledge base. Forward chaining is a data-driven reasoning strategy, this implies that, in contrast with the backward chaining strategy, the inference engine is triggered automatically when changes to the knowledge base are made. The main difference with the backward chaining strategy is the behaviour when new data becomes available. Instead of starting from scratch, the inference engine builds its proofs bottom-up, resulting in a rederivation of only those parts affected by the appearance of the new data. The benefit of using this strategy in a distributed context is that unrelevant changes to the knowledge base are filtered out in the first step of reasoning: Filtering is done by checking the prerequisites of the rules, if none of these prerequisites match with the new information, this can be dismissed.

Truth Maintenance System

The Fact Space Model provides a logic coordination language for reasoning about a perceived environment. As this environment is a mobile ad hoc network, devices can go out of range due to the transient connectivity of the network. Such disconnections result in the retraction of facts, namely the quantified facts of the other device. As the connectivity of a mobile ad hoc network is transient, it is very likely that those devices' communication is restored and all quantified facts need to be exchanged and reasserted. These changes to the fact spaces can not be delayed as context-aware applications depend on the facts residing in it.

In order to overcome this, CRIME adapts the behaviour of the RETE algorithm in order to more efficiently support retractions and reassertions of facts. The basic RETE algorithm, that is described in chapter 3, handles retractions of facts by propagating the negation of the token through the network – instead of inserting a positive token to the root node, a negative token is inserted and propagated. By propagating this dual token through the RETE network, the match operations of the join nodes, that perform consistency variable checks, need to be recomputed. This technique is rather costly as all previous computations need to be recalculated. Furthermore, when the fact is reasserted, those computations need to be recomputed twice. So, instead of deleting this information a second cache is introduced in the basic RETE network. This cache keeps track of all tokens derived from a fact which is no longer present in the fact space. Whenever the fact is reasserted, its corresponding tokens can be reinserted in the normal memories of the network. The dependencies between an asserted fact and its corresponding tokens is realised by using forward references. This technique is called *scaffolding* and is explained in more detail in chapter 8.

Extending Primitives

CRIME provides some extra statements which enable writing more complex rules. Those statements are borrowed from Prolog [62], namely *findall*, *bagof* and *length*. The remainder of this section highlights these constructs by means of an example rule.

The first PROLOG construct we introduce is *findall*. The findall statement consists of three arguments: a variable, a query and a variable for accumulating bindings. The first argument denotes the values of the query to be accumulated in the last argument. Consider the rule shown in listing 5.5, every occurrence of the variable `?person` in the query is accumulated in a list `?persons`. The rule determines all users that are located in an office.

```
present(?persons) :-
          findall(?person,
                  ( location(?person, office). ),
                  ?persons).
```

Listing 5.5: CRIME: rule using `findall`

Consider the facts shown in 5.6 which are part of CRIME's working memory. The *findall* rule of listing 5.5 adds a new fact to the working memory, namely `present([ 76539, 79783 ])`. The fact that is added to the working memory has one argument which is a list containing the user identifiers of all the persons that are located in an office.

```
location(75773, meeting).
location(76539, office).
location(72398, meeting).
location(79783, office).
location(83417, kitchen).
```

Listing 5.6: Facts in CRIME's working memory

Another useful PROLOG construct is *bagof*. Bagof has a syntax similar to the syntax of the findall statement. Whereas findall is able to collect occurrences of a certain variable into a list, bagof is able to collect these variable occurrences according to the unbound variables in the query. So, in the example rule of listing 5.7, the variable `?person` is accumulated according to the `?room` he is located.

```
present(?persons, ?room) :-
            bagof(?person,
                  ( location(?person,  ?room). ),
                  ?persons).
```

Listing 5.7: CRIME: rule using `bagof`

Reconsider the working memory shown in listing 5.6. The *bagof* rule adds new facts for each room, so three facts are added to the working memory: `present([75773, 72398], meeting)`, `present([76539, 79783], office)`, and `present([83417], kitchen)`. For each room the user identifiers are accumulated, these lists are the first argument of the `present`-facts that are added.

A last construct determines the *length* of the lists that are used by a findall or bagof statement. For instance, listing 5.8 shows an adapted version of the *bagof* rule. This rule determines the number of persons in a certain location.

```
present(?number, ?room) :-
            bagof(?person,
                  ( location(?person,  ?room). ),
                  ?persons),
            length(?persons, ?number).
```

Listing 5.8: CRIME: rule using `length`

The facts in the working memory that is shown in listing 5.6 cause the addition of three `present`-facts, namely `present(2, meeting)`, `present(2, office)`, and `present(1, kitchen)`.

The introduction of these PROLOG constructs enables CRIME to solve the multiple **rd** problem that is addressed in chapter 2. Recall the example where a user wants to retrieve all tuples of a customer with id 42: $< Leased, ?CarId, 42 >$. In LINDA an **rd** operation can block an entire program, namely when performing a *(n + 1)*th **rd** operation and there are only n

tuples of type `Leased` residing in the tuple space. CRIME doesn't suffer from this problem as the *findall* construct cashes all matched facts in the RETE network and returns all the occurrences of the variable `?CarId` in one accumulated list. Listing 5.9 shows a CRIME rule which retrieves all car identifiers leased by the customer with id 42.

```
leased(?carIds) :-
          findall(?carId,
               ( leased(?carId,  42). ),
               ?carIds).
```

Listing 5.9: CRIME: solution for multiple rd problem using `findall`

5.2.4 Crime's Contributions

CRIME is based upon the federated tuple spaces provided by LIME. These tuple spaces form a consistent way to share information which supports both space and time uncoupling, as is discussed in chapter 2. However, LIME does not provide direct support to react on a combination of tuples residing in the tuple space. Furthermore, the language doesn't support a mechanism to undo the reactions on tuples, when they are no longer present in the tuple space. CRIME overcomes these shortcomings as it uses a logical inference engine for reasoning about combinations of facts residing in the fact space. The use of a logical inference engine is also supported by other languages like for example GAIA that is described in chapter 4. As for GAIA, the rules of this logic language allow to keep track of causal links between the current context and possible reactions on that context information. However, there's no support for redoing the actions whenever the context is changed. Reacting and reversing the actions based on the sensed context is possible in CRIME as the retraction of a fact in the fact space can cause the prerequisites of a rule to fail and the consequences of that rule are made undone if it is specified.

5.3 Building Context-Aware Applications

This section gives an overview of the most representative context-aware applications developed in CRIME. As all these application rely on location based-information, first an overview is given how location detection is conceived in CRIME. This location detection system is used in the rest of the applications presented. Then two light-weight applications, an in/out board and a messenger service [19] are presented. Subsequently we explain the implementation of a slightly more involved application, namely a context-aware jukebox [49, 6].

5.3.1 Detecting a User's Location

Recently many new detection mechanisms have been proposed to derive a person's location. A common factor of many of these new technologies is that indoor location positioning is mostly realised by some sort of tracking device which individuals must carry in order to know their whereabouts. This device can be used to actively send through the location of the person in question or can be passively scanned when entering an area. Two popular technologies are infrared signals (as used in the Active Badge system [65]) and the recently developed RFID tags [45].

The choice of tracking technology for an industrial application can be crucial, as the accuracy and the power consumption of these technologies can be significantly different. However the impact on the development of these applications is rather minimal. We have therefore opted to use a rather conservative stance, by using the available bluetooth technology of the Mac mini (the development platform) in order to detect the users of the system by means of their bluetooth-equipped cellular phones.

Concretely, our setup consists of an event-driven application detecting all reachable bluetooth devices. Upon the detection of a new device, it asserts a private fact (e.g. `observed("Alice's Phone")`) by making use of the API of the CRIME engine. The application further uses a trivial rule in order to map these facts upon `location` facts, as we have used in previous examples in this chapter. The rule performing this mapping is shown in listing 5.11.

```
thisRoom(LivingRoom).

public -> location(?id, ?room) :-
          thisRoom(?room),
          observed(?id).
```

Listing 5.10: Location detection rule in CRIME

5.3.2 In-Out Board

The *in-out board* application developed in CRIME gives an overview of a person's location. The board also gives information about the phone number of the room where the person is located. This information could be used in a basic phone forwarding system.

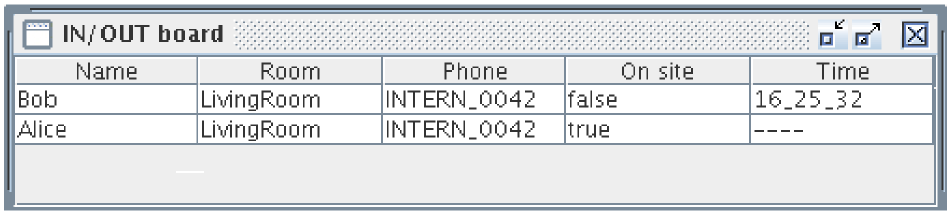
The application uses the public `location` facts that are asserted to the fact space by the location detector as is described in section 5.3.1. Beside this information, it also makes a mapping from a user identifier, `?id`, to the user's name. Furthermore, facts representing the mapping between rooms and their corresponding telephone number are also kept in the public fact space.

```
public -> person(76539, Bob, student).
public -> person(75773, Alice, student).
public -> phone(LivingRoom, INTERN_0042).

:SetBoard(?name, ?room, ?number) :-
          location(?id, ?room),
          private -> person(?id, ?name, ?occupation),
          private -> phone(?room, ?number).
```

Listing 5.11: In-out board application in CRIME

The user-defined action `SetBoard` refreshes the GUI shown in figure 5.6. In this screenshot we see that Alice is currently in the living room, whereas Bob's location is unknown at the moment. However, the in-out board provides information telling that Bob was last seen in the living room at 16h25. This is achieved by the compensation action of the `SetBoard` class as defined by the programmer. When this compensating action is executed, for instance when Bob left the living room, the compensating action records the time and updates the GUI.


IN/OUT board

Name	Room	Phone	On site	Time
Bob	LivingRoom	INTERN_0042	false	16_25_32
Alice	LivingRoom	INTERN_0042	true	----

Figure 5.6: In-out board in action

5.3.3 Messenger Application

The *messenger* application consists of viewing and sending messages exchanged between certain persons. Viewing the messages is performed by special boards which display the messages for the persons residing the room. Sending messages is realised by making use of a CRIME application. The rules steering this application are shown in listing 5.12. It makes use of special `message` facts which are inserted by making use of a GUI. When such a fact is asserted, a trivial mapping is made for attaching the sender's name to the message. The example shown in listing 5.12 consists of a fact base and the rule needed to attach the user's name to his messages. Given this fact base and the rule, a public fact `message(Bob,Alice,"Hello Alice")` is published.

```
private -> myName(Bob).
private -> message(Alice, "Hello Alice").

public-> message(?from , ?to, ?message) :-
    private -> message(?to, ?message),
    private -> myName(?from).
```

Listing 5.12: Messenger application in CRIME

In order to display the messages, a message board must detect those messages intended for a person located in the same room as the board. Furthermore, when a person leaves that room, the messages intended for this user must be removed from the board. Also when moving the board itself from one room to another room, the board of the messenger application must adapt itself to display the messages of the new room it is located in. The rule which assures all of the above requirements is shown in listing 5.13.

By making use of the location detection mechanism described in section 5.3.1, the board detects the room it is located in. As a direct consequence of the retraction mechanism of CRIME, the board adjusts itself when it is relocated. The rule specifies that in order to show a message, the receiver of the message must be located in the same room as the board displaying the message. This achieved by using the same variable name, namely `?room`. Making sure that the board retracts the messages when a user leaves the room is realised by executing a compensating action when the `location(?id,?room)` fact is retracted.

```
thisBoard(BOARD_IDX42).

:showMessage(?from, ?to, ?message) :-
      message(?from , ?to, ?message),
      location(?board, ?room),
      thisBoard(?board),
      person(?id, ?to, ?occupation),
      location(?id, ?room).
```

Listing 5.13: `SetBoard` application in CRIME

5.3.4 A Context-Aware Jukebox

Alice and Bob are two students at the *Vrije Universiteit Brussel* and they spend quite a lot of time together. When they are not studying, a great part of their life is all about music. In the students home, where they live during the week, an entertainment room is available. This room is used for relaxing and most of the time a jukebox is playing. Each person has a different taste when it comes to music: Alice, Bob and their friends each have a favourite genre. No arguing of who's in charge of the jukebox is needed, neither is reminding to turn off the jukebox when leaving the entertainment room as the last person. This is a direct consequence of the fact that the jukebox is in fact a small computer (a Mac Mini in our setup). The jukebox combines the location information it receives from the location detectors and the musical preference of Alice and Bob to construct an acceptable playlist for all users residing in the room. Moreover when friends come over their taste in music can be taken into account as well. Therefore the jukebox knows exactly what his users like and plays just the songs they prefer. When nobody is in the room, the jukebox even stops playing.

In order to implement the scenario presented above we have identified four important issues. A first important issue is the detection of the users currently located in the room. Secondly when friends come over they should be able to deploy the applications needed to inform the jukebox of their personal taste in music. Thirdly, when users walk into the entertainment room the jukebox must be able to perceive their musical preferences and possibly switch itself on. And finally, the jukebox must compile the playlist according to the various tastes of the users present in the room.

The first requirement, namely the detection of users in a room is implemented as already presented in section 5.3.1. We discuss the rest of the necessary building blocks for this setup in the remainder of this section.

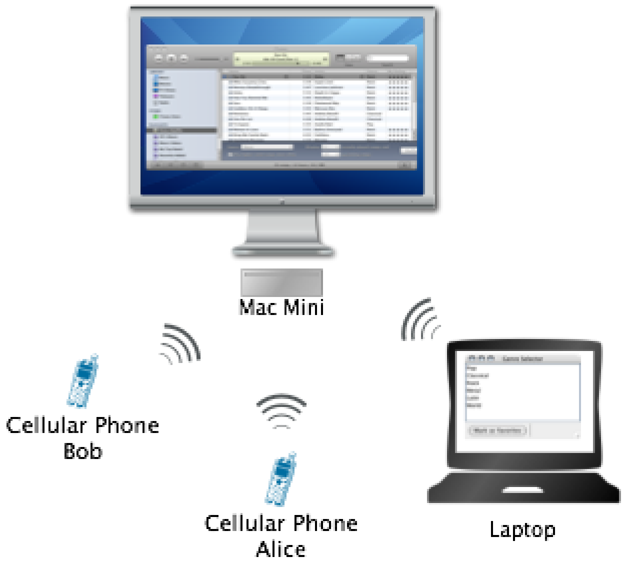


Figure 5.7: Components of jukebox application

In order to implement the scenario described above, several components are needed. These components are depicted in figure 5.7. The jukebox is simulated by the iTunes application running on a Mac Mini. Alice and Bob's cellular phone with bluetooth connectivity is needed for detecting when they are in communication range with the Mac Mini. A last component is a computer that can be used for specifying a user's taste of music: for instance, when friends come over, they can select their favourite kind of music on that computer in order to inform the iTunes application of their musical preference. This computer is not always needed, as friends can deploy a CRIME application which enables selecting their favourite music on their own devices.

Deploying Applications

One issue of the scenario described in the previous section is the deployment of applications, in this concrete scenario an application to specify one's musical preferences. In order to be able to provide this kind of service a minimum infrastructure is needed. Therefore a small application in CRIME is developed to keep track of the available applications depending on the users' location. The user which runs this application is informed of all the available applications he can deploy. Listing 5.8 illustrates how the GUI is updated by making use of the offer action. Downloading and running the applications when the user opts to deploy it is realised by the underlying application.

A remarkable property of this application is that it dynamically takes into

```
:offer(?name ?url) :-
  application(?name ?url).

application(jukebox, "http://prog.vub.ac.be/amop/crime/jukebox.zip").
```

Figure 5.8: Supporting the advertisement of available applications

account the available applications depending on the user's context. This is a direct consequence of using `public` facts in order to represent the available applications. As these public facts are shared according to the connectivity of the device offering them the user only perceives those applications offered in his approximation.

When a fact that represents an available application is retracted, the `deactivate` method of the offer action is invoked. In the current implementation this deactivate method removes the available application form the list. One possible extension of the deployment application could be to treat a previously-deployed application differently. This could be achieved by asserting a private fact everytime an application is deployed, hence the rule described previously could incorporate this fact in order to behave differently.

Music Selection

The use of a coordination language implies that the coordination and distribution aspect of an application can be strictly separated. This separation implies that by making use of a coordination language one can enable non-distributed applications to take into account contextual information, with the limitation that this non-distributed application must provide the adequate hooks to change it behaviour. This observation has led us to the adaptation of an existing jukebox application that takes into account the contextual information provided by Crime. The jukebox application adapted is the *iTunes* music player[2] which is an established software artifact providing the appropriate set of hooks in order to implement our scenario. The Crime rules in order to steer the jukebox application and determining which music to play are shown in listing 5.9.

The first rule triggers the context event handler `Toggle` when a user is detected in the room, which in turn starts the music player. Similarly, when no users are present in the room, the `deactivate` method of that application specific action ensures that the jukebox stops playing music. Again the detection mechanism used for this application is based upon the `location` facts which were generated by the location detecting application described in 5.3.1.

Besides detecting the users present in a room, the jukebox application must attribute the ratings of a particular genre depending on the current number of users in the room and the number of users preferring that specific taste of music. The second and third rules, shown in figure 5.9, calculate these values by

[2]Copyright 2000-2006 Apple Computer Inc.

making use of the *findall* and *bagof* constructs presented in section 5.2.3. The findall of the rule starting on the fourth line, accumulates all persons present in the room into the list `?persons`. Analogue to findall, the bagof construct used in the rule (line 10-15) accumulates all persons in the room according to their preferred genre. Finally the last rule uses the user-defined action `updateRating` to change the rating of a particular genre using the information provided by the two previous rules. The `updateRating` action uses this information in order to compile a playlist where highly rated music is featured more often. This is sufficiently to ensure that the music played by the jukebox is appreciated by most users present in the living room.

```
1   :toggle() :-
2     location(?person, "Living␣Room").
3
4   total(?quantity) :-
5     findall(?person, (
6       location(?person, "Living␣Room")),
7       ?persons),
8     length(?persons, ?quantity).
9
10  category(?genre, ?quantity) :-
11    bagof(?person, (
12      location(?person, "Living␣Room"),
13      prefers(?person, ?genre)),
14      ?persons),
15    length(?persons, ?quantity).
16
17  :updateRating(?genre, ?rating) :-
18    category(?genre, ?absolute),
19    total(?total),
20    rating is ?absolute / ?total.
```

Figure 5.9: Crime: rule set to customise jukebox playlist

5.4 Conclusion

This chapter describes the Fact Space Model and its experimental implementation Crime. This coordination language is based on the tuple spaces provided by Lime and introduces new mechanisms to solve the problems at hand in a mobile ad hoc network, namely disconnections. As the Fact Space Model treats the federated tuple spaces of Lime as a fact base for storing contextual information where every change is reflected in an appropriate action. When context becomes available by asserting facts, this assertion adapts the behaviour of the applications depending on this sensed context. When the facts are retracted, as context becomes unavailable, the context sensitive application's behaviour is adapted according to this loss of information. So, both the assertion and retraction of facts are important for the adaptation of the applications.

CRIME provides a logic language to reason about a combination of facts and their consequences. By keeping these causal links, context adaptations can be performed when context becomes available or unavailable. CRIME handles disconnections of context providers in an elegant way: when context information becomes unavailable, the corresponding facts are retracted from the federated fact space possibly resulting in the retraction of actions that were previously executed.

The next chapter describes the implementation of CRIME in more detail and highlights the design choices we have made

Chapter 6

Crime

This dissertation advocates the use of a distributed reasoning engine in combination with a truth maintenance system in order to ease the development of context-aware applications. While in the previous chapter we have discussed how applications can be written by using the Fact Space Model and subsequently in the experimental language CRIME, this chapter is more technical and gives an overview of the various aspects of the implementation of this language. CRIME is a direct translation of the Fact Space Model and incorporates an inference engine which is forward-chained by making use of a RETE network.

The language we have chosen for the implementation of CRIME is JAVA. The decision of JAVA is justified as it's one of the few languages which is widely supported by mobile devices. The aspects of the implementation which we highlight in this chapter can be divided into three large parts: A first important aspect of our implementation involves the translation of source code into JAVA objects. This translation is implemented by making use of the ANTLR tool. After this translation, the resulting parse tree is processed and transformed into a RETE network. A second important aspect of our implementation is the inference engine. This engine uses the RETE network that is built by the parser and propagates tokens trough this network in order to derive the effects of the assertion or retraction of a fact. The implementation of the various nodes and how they propagate tokens is highlighted in section 6.2. The third large part of our implementation is the network layer which ensures that facts are exchanged when two devices are in reach.

6.1 Parsing

Parsing is the process to translate a sequence of tokens into a datastructure for further processing by using the rules specified by a grammar. This process can be highly automated and various tools for realising this are freely available. The tool used for the implementation of CRIME is ANTLR, which transforms the input of a file into an *abstract syntax tree* given a grammar in an *Extended*

Backus-Naur form. Furthermore, the tool provides the means to traverse the tree in order to transform this abstract syntax tree into the various datastructures used for further processing, in the case of CRIME we translate this tree into a RETE network when parsing a rule.

6.1.1 Parsing Rules

A simplified version of the *Backus-Naur* form of the grammar of a CRIME rule is shown in listing 6.1. This grammar specifies that a **rule** consists of a number of actions followed by the inference symbol (*:-*) and several conditions. **Actions** consist of an action possibly followed by more actions, where an action can be either a user-defined action or a logical fact to be asserted, called *pattern.* Such a user-defined action is a pattern proceeded with a colon. **Conditions** consist of a pattern, a findall construct or a bagof statement possibly followed by more conditions and ending with a dot symbol. The **Patterns** expression consists of several patterns which all have a type and attributes surrounded by parenthesis. **Attributes** consist of several attributes, which can be a variable or a constant. **Variables** consist of a name proceeded with a question mark.

```
<rule>          ::= <actions> ':-' <conditions>
<actions>       ::= <action> ',' <actions> | <action>
<action>        ::= <user_action> | <pattern>
<user_action>   ::=  ':' <pattern>
<conditions>    ::= <pattern> ',' <conditions>
                  | 'findAll' '(' <variable> ( <patterns> ) <variable>')' ','
                    <conditions>
                  | 'bagof' '(' <variable> ( <patterns> ) <variable>')' ','
                    <conditions>
                  | '.'
<patterns>      ::= <pattern> ',' <patterns> | <pattern>
<pattern>       ::= <type> '(' <attributes> ')'
<attributes>    ::= <attribute> ',' <attributes> | <attribute>
<attribute>     ::= <variable> | <cte>
<variable>       ::=  '?' <name>
```

Listing 6.1: Syntax of the CRIME rules in Backus-Naur form

With the specification of the rule grammar shown in listing 6.1, ANTLR can parse and transform rules into a parse tree. Consider the example shown in listing 6.2, this rule collects the names of all persons residing in the kitchen and passes them to the user-defined action `LocatedInKitchen`.

```
:LocatedInKitchen(?persons) :-
      findall( ?name,
               ( location(?id, Kitchen),
                 userInfo(?name, ?id) ),
               ?persons).
```

Listing 6.2: CRIME: *located in kitchen* rule

The resulting syntax tree is shown in figure 6.1. Traversing the tree in a depth first manner and collecting all the leafs results in the original rule. The ANTLR tool provides the facilities to transform such a tree into JAVA objects, during this phase various checks are executed. For instance verifying whether all variables used in the *consequences* of a rule are provided by the body of the rule. This allows the inference engine to transform the resulting parse tree that is represented by JAVA objects (at that moment in time), into a **Rete** network without the need to perform these checks during that phase.

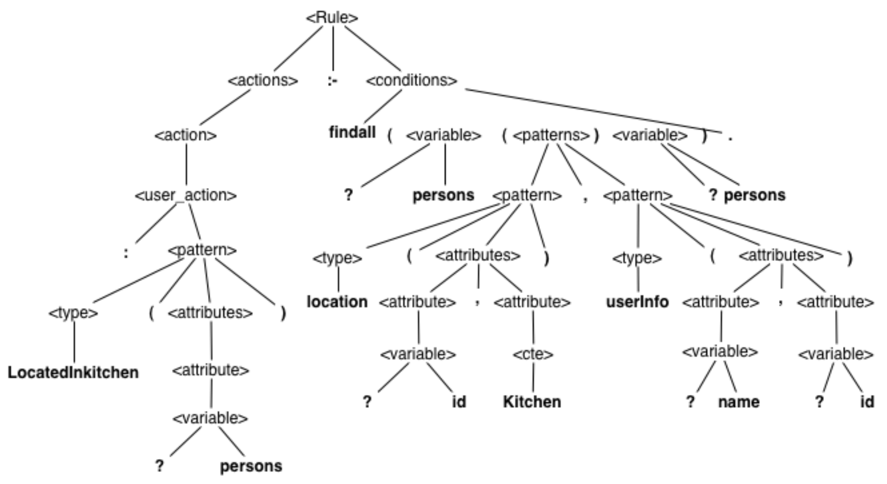


Figure 6.1: Parse tree of *located in kitchen* rule

6.1.2 Transforming the Parse Tree

A parse tree could be used in order to represent and reason about the rules in the system, however this representation is far from suitable. This section explains how this parse tree is further transformed in order to compile the parsed rules into a RETE network. As shown in the uml diagram depicted in 6.10, all expressions of the parse tree are modelled by extending from the abstract class `Expression`. This class ensures that all expressions in the system implement the `buildLayer` method. This method forms the core of how the RETE network is built. By incorporating the `buildLayer` methods in the expressions, new expressions can be added easily.

A `buildLayer` method receives three arguments: the reasoning engine, the last produced node by previous expressions, and all subsequent expressions. The reasoning engine is used in order to access transformations which are used by more than one expression form. The last produced node is needed to extend the RETE network built so far, with the nodes needed for representing the current expression. The third argument is provided as the previous expressions are needed in order to make consistency checks between several prerequisites in the body of the rule. Note that not all prerequisites contribute to the extension of a token during its propagation through the RETE network: for instance variables in a *not* expression are not bound with variables in subsequent conditions.

For example the body of a rule: `not foo(?a),bar(?a)`, does `not` express that the negation of the `foo` fact must have a same variable at position 1 as the subsequent `bar` fact. It only expresses that in the absence of a fact which binds with `foo(?a)` the rule is able to be triggered, therefore the variable `?a` in the precondition `bar(?a)` is free.

After the parsing phase by ANTLR, the rule represented in listing 6.2 is modelled by an object `Rule`. This object contains prerequisites and consequences, representing respectively the body and the head of the rule. This object is passed to the reasoning engine by invoking its `deploy` method which is shown in listing 6.3.

The deployment of a rule consists of iteratively calling the `buildLayer` methods of all the expressions residing in the prerequisite of the rule. During every iteration of this process, the *lastEnd* node, is set upon the result of this method call. Furthermore, the vector which contains all the previous expressions must be updated by the expression itself. By leaving this update to the expression itself, expressions that do not pass additional information, like for instance the *not* expression, can be implemented more conveniently. When all expressions of the prerequisites are handled, the consequences of the rule are initialised, this involves the lookup of the user-defined actions and transforming the facts in the consequences to an assert action provided by CRIME. When these consequences are initialised, a production node is made and attached to the end of the RETE network. In the rest of this section an overview of how expressions are converted in order to build a RETE network is given.

```
deploy(Rule rule) {
        ...
        for(Iterator preItr = prerequisite.iterator(); preItr.hasNext(); ) {
                        Expression pre = (Expression)preItr.next();
                        lastEnd = pre.buildLayer(this, lastEnd, lPrereq);
        }
        initaliseConsequences(rule.getConsequences(), lPrereq);
        ProductionNode pNode = NodesFactory.productionNode(rule, this);
        lastEnd.addChild(pNode);
}
```

Listing 6.3: Deployment of a rule by the reasoning engine

Prerequisite

Pseudo code for the `buildLayer` method of a parsed fact is shown in listing 6.4. For every prerequisite in the rule which is represented as a `Fact` object, its `buildLayer` function is responsible for filtering out those facts that have a wrong type by using a filter node. This is realised by invoking the *root node*'s `insertChild` method, which returns a filter node filtering on the right type. This root node only creates one filter node for each type so that these kind of filter nodes can be reused by different rules.

Another filter node is created to filter out constant checks and checks on *intra-elements* which is performed by calling the engine's `buildFilterNode` method that instantiates a filter node with the right filters for that prerequisite. This is accomplished by iterating over the attributes of that fact and creating an `AttributeFilter` for every constant. This method also creates filters to test on intra-elements. This is realised by making a `VariableFilter` for every variable which occurs more then once. This filter ensures that only those tokens are passed that have the same attributes on the corresponding place of the variables residing in that fact. This last created filter node is subsequently added as a child to the filter node responsible for checking the fact's type.

To finalise, a join node is instantiated which combines the last node built by previous prerequisites in combination with the one performed by transforming this prerequisite. The making of a join node requires that there are made join node filters for every variable occurring in the fact and in one of the prerequisites handled before. Subsequently, it adds itself to the prerequisites and returns the join node or the right end in case the prerequisites were empty.

```
class Fact {
    method ReteNode buildLayer(CrimeEngine engine, ReteNode
                               lastEnd, Vector prerequisites) {
        FilterNode filterNode = rootNode.insertChild(myType);
        ReteNode endNode = Engine.makeFilterNode(this);
        ReteNode rightEnd filterNode.addChild(endNode);
        JoinNode joinNode = makeJoinNode(this,lastEnd,rightEnd, prerequisites);
        prerequistes.add(this);
        return joinNode or RightEnd;
    }
}
```

Listing 6.4: `BuildLayer` method for `Fact` expression

To summarise, we recapitulate the different steps that must be performed for transforming a rule's prerequisite:

1. a filter node must be instantiated or updated as the root node's child in order to filter on the prerequisite's type

2. for all constant attributes of the fact a new `Filter` is created

3. group all those filters (needed for constant tests and intra-elements) together in a second filter node

4. this last filter node forms the child of the first one which is used for filtering on the fact's type

5. the previously built network – caused by building another prerequisite – must be joined with this last filter node

6. this join node is the last node of the Rete network built so far

Findall

Building the Rete network for a *findall* expression is very similar to that of building a prerequisite. However, as a findall statement has a list of expressions as its second argument, this subexpression contains a list of statements that must be built also. Pseudo code for the `buildLayer` method of the findall statement is shown in listing 6.5. As can be seen, first the Rete network of the expression list is built. Then the output of this newly built network is joined to the previously built network of the previous prerequisites. In case that the findall expression is the first prerequisite of the body, this step can be omitted as there are no previous prerequisites. The resulting `joinNode` or to the `rightEnd` in case that there where no previous prerequisites, is then extended with a findall node.

```
class Fact {
    method ReteNode buildLayer(CrimeEngine engine, ReteNode
                              lastEnd, Vector prerequisites) {
        ReteNode rightEnd = buildReteNetwork(this.expressionList);
        JoinNode joinNode = makeJoinNode(this.expressionList,
                                         lastEnd,rightEnd, prerequisites);
        prerequistes.add(this);
        ReteNode findall = makeFindallNode(accumulator,makeAttributePicker());
        attach findall to the joinNode or to rightEnd
        return findall;
    }
}
```

Listing 6.5: `BuildLayer` method of findall expression

Recapitulate the different stages for transforming a findall statement:

1. transform the query which is the findall's second argument
2. join the resulting network of step one with the Rete network that was built by transforming previous expressions
3. the last node of the current Rete network is then is attached with a findall node

The transformation of a *bagof* expression only differs from the one that was just described as a bagof node is attached in the last step of the `buildLayer` method instead of a findall node.

Example

Recall the example rule from listing 6.2, its transformation into a parse tree is already shown in section 6.1.1. Here we show how this parsed rule is transformed into a Rete network.

The first step of the process invokes the engine's `deploy` method with the parsed *located in kitchen* rule. The prerequisites of this rule consist of only one expression, namely a findall expression for which the engine calls its `buildLayer` method . Transforming a findall statement starts by building the network of the subexpressions residing in it. In this example there are two facts, namely `location(?id, Kitchen)` and `userInfo(?name, ?id)`. Executing the `buildLayer` method of this first prerequisite results in the addition of one filter node checking on the type `location`.

Thereafter the `buildLayer` of the second fact is invoked which first creates a filter node checking the type `userInfo`. Moreover, this method ensures that that the filter node created by the previous expression, is joined with this newly created filter node. Recall that the previous prerequisites are needed for keeping track of used variables, for which variable consistency checks might be performed. When the second fact creates a join node to connect the filter node `location` and the filter node `userInfo`, these previous prerequisites are searched for variable bindings occurring also in the second fact. As the previous prerequisite `location(?id, Kitchen)` already contains a variable `?id` the join node's responsibility is to check this variable binding. This join node is returned for the building process of the findall which simply adds a findall node to this join node.

To finalise, the engine builds a *production node* from the rule's consequences and attaches this to the returned findall node. The various steps of this process are depicted in figure 6.1.2.

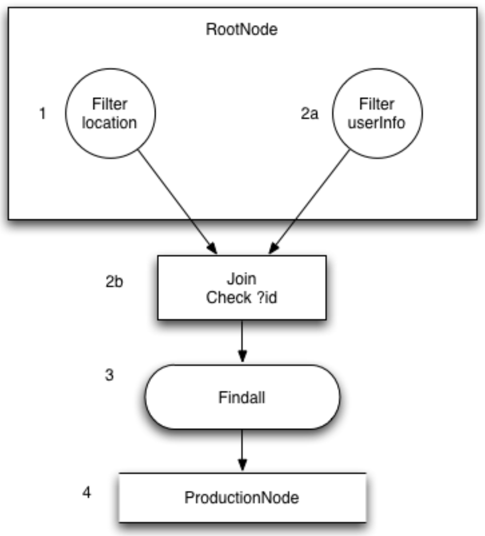


Figure 6.2: Building Rete network for the *located in kitchen* rule

6.2 Inference Engine

6.2.1 Assertion and Retraction of Facts

Whenever a fact is *asserted*, it is added to the working memory of the reasoning engine. Furthermore, the fact is converted to a corresponding token and propagated into the Rete network. In case of a *retract* operation, a corresponding negative token is created and propagated through the network. The propagation of tokens into the network is implemented by the `insert` method implemented by all nodes in the network as shown in figure 6.3. This method has one argument, namely the token to be propagated, and every node in the network performs a specific test in order to determine whether the token should be propagated to its children or not.

The propagation begins in the root node which recursively passes the token to its children nodes until the token is inserted into a production node or one of the nodes decides to stop the propagation. In the next section we give a detailed overview of how this propagation works for the various nodes in the network.

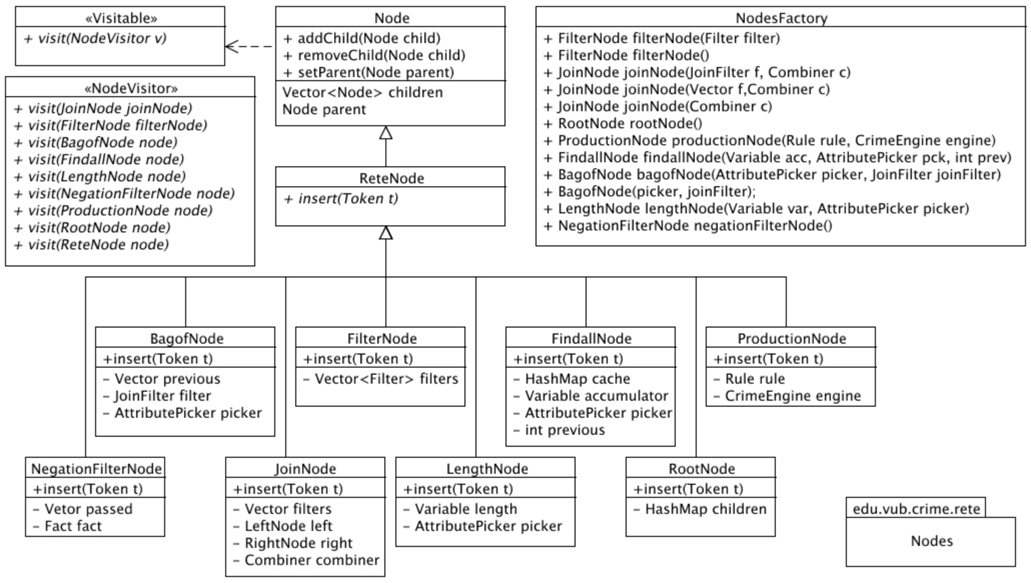


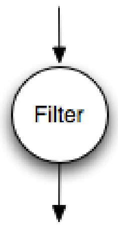
Figure 6.3: Uml diagram of the Rete nodes' structure

6.2.2 Functionality of the Nodes

An overview of the various nodes in the network is shown in figure 6.3, all these nodes derive from the class `ReteNode` which in turn derives from the class

`Node`. The `Node` class implements the general functionalities of a node having one parent, e.g. accessing and mutating its children. The `ReteNode` class has one abstract method, `insert` which must be implemented by all its subclasses. This section gives an overview of how this propagation method is implemented for the various nodes.

Filter Node



Recall that *filter nodes* form the *alpha* part of the RETE network, as is described in chapter 3. There are two different kind of filter nodes: the first one performs checks on the type of fact whereas a second one performs tests to check everything else, like for example constant tests and *intra-element* tests (variables occurring more than once in the same condition).

The difference between these kind of tests is reflected by using different `Filters`. An uml diagram of the various filters is shown in figure 6.4. All filters derive from the abstract class `Filter` and must implement the `pass` method. This method expects a token as argument and returns a boolean indicating that the filter allows the token to be *passed* or not.

Each filter node contains one or more filters performing the needed tests. The filter nodes that check the type of fact only have one filter, `FilterKind`, which only passes tokens containing a fact of the right type. Note that the type of a fact not only consists of the actual type name of the fact, but also the arity of the fact is taken into account. For example the type of the fact `type(?varOne,?varTwo)` is `type2` and not `type`. Using the arity as well allows the programmer to use facts having the same type name but with a different arity. The reason to split up the test for type checking is done in order to reuse the nodes checking for the same type.

Filter nodes that perform all other tests can have several filters: one for each test to be performed. The current implementation of CRIME supports three kinds of filters to accomplish these tests. Testing that a certain attribute of a token equals a constant is achieved by making use of the `FilterAttribute` filter. A special kind of filter is responsible to filter out only those facts from a specific fact space, namely the `FilterSpace`. A last filter is used to perform the intra-elements tests and is called `FilterVariables`. This filter uses a variable checker for storing the indices of the attributes that must be compared, and performs this comparison by using this stored information. The `VarChecker` class is explained in more detail in the paragraph explaining the functionality of the join node.

The `propagate` method of a filter node applies all filters to the propagated token, when all filters pass the token, the filter node propagates the token to all its children. Filter nodes do not behave differently for a positive or negative token.

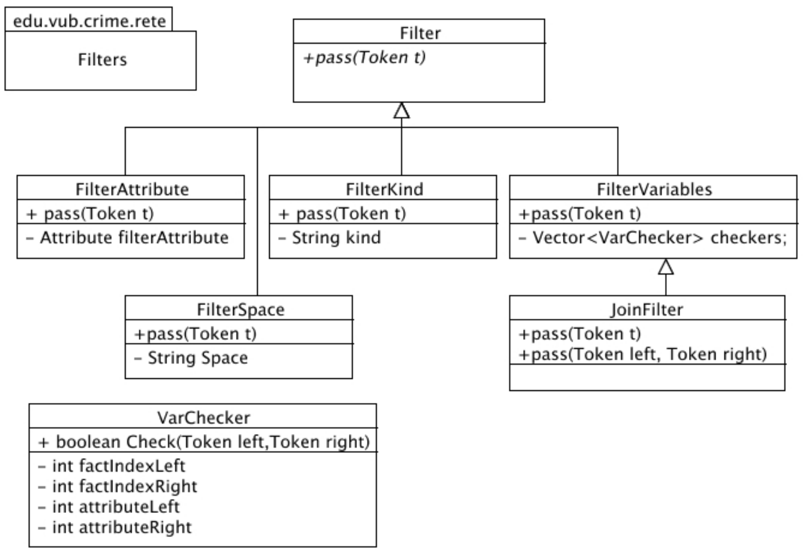
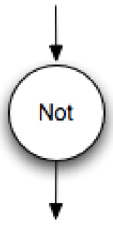


Figure 6.4: Uml diagram of the filters

Negated Filter Node



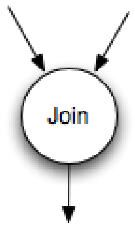
This node implements the filter process of a rule whose first fact is negated, as for instance the rule in listing 6.6. This rule is triggered whenever a new person – someone whose detailed information is not known – enters the campus. In that case, the person must subscribe himself.

The first prerequisite of the rule, `not userInfo(?name, ?id)`, is negated and hence the first node of the RETE network for this rule is a negated filter node. When no tokens are added to this node, a dummy token is passed to its child, a join node in the example rule below. From the moment that at least one token is inserted in this node, the dummy token is removed from its child and no token is passed. These passed tokens are kept in order to detect their retractions by the propagation of a negative token. When such a negative token is inserted the corresponding positive token kept in a cash is deleted, when this cash is empty the dummy token is propagated again. In order to avoid floundering we have opted to not allow free variables in negated facts. As variables can only be bound by previous facts, the first prerequistes in a rule can not have variables. Therefore, we can use a simple negated filter node for these prerequisites, for other negated prerequistes not at the first place in a rule we must use a join node with a special filter as seen in the next section.

```
:Subscribe(?id)  :-
        not userInfo(?name, ?id),
        location(?id, campus).
```

Listing 6.6: Rule with first fact negated

Join Node



A *join node* is a special kind of RETE node as it combines two nodes by performing variable consistency checks. The join node is implemented by using two inner classes, namely a *left node* and a *right node*. Those nodes each have their own memory for storing the tokens retrieved from the corresponding parent node.

An outline of this JAVA class is presented in listing 6.7. By introducing these two inner classes, a join node may never be attached directly to another node, but only through its left or right node, which ensures that the join node knows if it is activated from right or left.

```
public class JoinNode extends ReteNode {
        private Vector filters_;
        private LeftNode left_;
        private RightNode right_;
        private Combiner combiner_;

        private class LeftNode extends ReteNode {
                private MemoryTable memory_;

                public void insert(Token tkn) {
                        ...
                }
        }

        private class RightNode extends ReteNode {
                private MemoryTable memory_;

                public void insert(Token tkn) {
                        ...
                }
21          }
22
23          public void insert(Token tkn) {
24                  throw new RuntimeException("...");
25          }
26  }
```

Listing 6.7: Outline `JoinNode` class

Whenever a token is inserted to the left or right node, this token is inserted in the corresponding memory and a match operation must be performed by the join node. This matching process is performed by the join node's *combiner* (line 5 of the code excerpt in listing 6.7). Depending on whether the second fact is negated or not, a `JoinCombiner` or `NegationCombiner` is used.

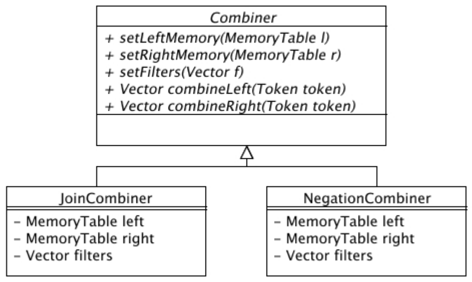


Figure 6.5: Uml diagram of the combiners

A join combiner is used for combining a new inserted token with the tokens in the other memory which have a consistent variable binding. In case of a *left activation* the tokens in the right memory are all checked in order to test their consistency: when a right token satisfies all the filters of the join node, the token is said to *match* with the newly inserted token, and the combination of both tokens is sent to the join node's children. Same reasoning holds for *right activations*.

A join node uses a special kind of filters for testing consistency of variable bindings, namely `JoinFilters`. These filters are used to check that some attribute of a fact of a certain token is the same as an attribute of another token's fact. So, for each variable check that needs to be performed, a new join filter is added to the join node.

A variable checker keeps track of four indices as is illustrated in table 6.1. For each fact the index of the attribute that must be checked is kept, as well as the position index of that fact in the token. The second column of the table represent a left and right token whose variable consistency check is satisfied and their combination may be passed to the children of the join

index	**pass**	**not pass**

factLeft = 0
attributeLeft = 1
	location(75773, Kitchen)
	Kitchen
		location(75773, Kitchen)
		Kitchen

factRight = 0
attributeRight = 1
	location(76539, Kitchen)
	Kitchen
		location(76539, Living room)
		Living room

Table 6.1: Indices used by a variable checker

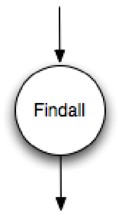
node. The variable check fails for the tokens represented by the last column of the table, as the value in the right token equals `Kitchen` whereas those of the left token is `Living room`. Hence, these tokens do not match and are not passed.

Whenever a rule contains a negated prerequisite that isn't the first condition of that rule, a join node with a *negation combiner* is needed for combining the tokens of that join node in a special way. An example of such a rule is given in listing 6.8, which is a modification of the previous rule as both prerequisites are exchanged. The join node that combines these two prerequisites uses a negated combiner. A negation combiner works by checking the matches of the left memory with those of the right memory. The token residing in the left memory is passed to the children when there are no matches. Because it's possible for a token from the left memory to have multiple matches with tokens in the right memory, every token keeps a number indicating how many matches it has. When a token is inserted into the left memory – by a left activation –, all tokens in the right memory are checked and for every match the number in the token is increased. When a token from the right is inserted, the tokens in the left memory are checked and all matches from that memory are increased by one. Removing a token from the right memory decreases every match with the tokens residing in the left memory.

```
:Subscribe(?id)  :-
        location(?id, campus),
        not userInfo(?name, ?id).
```

Listing 6.8: Rule with a fact negated

Findall Node



A findall token accumulates all occurrences of a certain variable in one or more prerequisites. Consider the *located in kitchen* rule given in listing 6.2: in this case the names of persons located in the kitchen are accumulated in a list, the variable `?persons`.

A *findall node* contains a cache which stores these values according to the variable bindings of preceding prerequisites. In the example rule, there's only a findall statement, hence there are no such variable bindings.

When a new token is inserted to the findall node, the needed attribute can be selected by using the known variable picker. The cache of the node needs to be updated with this new attribute: in case of a positive token, the corresponding entry must be extended with this argument whereas the list is shortened in case of a negative token. The correct entry of the cache is found by shortening the inserted token such that the token no longer contains the variable bindings of the findall statement itself.

Whenever the cache is updated, the previously sent token to the children must be retracted. For instance, consider a working memory that contains the facts that both Alice and Bob are located in the kitchen. Whenever Carol enters the kitchen as well, a new token `< + location(72672, Kitchen) >` is inserted into the findall node, and two tokens must be sent to the children: namely the tokens `< - locatedInKitchen( [Alice, Bob] )` and `< + locatedInKitchen( [Alice, Bob, Carol] )`. The retraction of this first token is needed as the accumulated list is modified and truth must be maintained. The insertion of a negative token is handled similarly.

```
locatedInKitchen(?persons) :-
      findall( ?name,
             ( location(?id , Kitchen),
               userInfo(?name, ?id). ),
             ?persons).
```

Listing 6.9: CRIME: rule using a findall construct

Bagof Node

A *bagof node* acts very similar to a findall node, only the structure of the cache is different. As a findall node groups his accumulations by previous variable bindings, a bagof construct accumulates them according to the free variables in the query of the bagof statement.

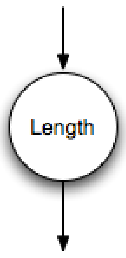
Conceptually the cash used by the bagof node maps a group of unbound variables onto `TokenValues` objects. This class acts as a buffer for keeping track of the accumulated variables of a certain group of tokens. When a positive token is propagated, the bagof node looks up the corresponding `TokenValues` object and first propagates a negative token containing the accumulated variables of the tokens residing in that object. Subsequently, it adds the attribute that must be accumulated to the `TokenValues` object and propagates a positive token containing the additional variable. For the propagation of a negative token the procedure is similar, except that the token is removed from the `TokenValues` object instead of added.

Consider the example shown in listing 6.10. When the token `< + location(76539,kitchen), userInfo(Bob, 76539) >` is added to the bagof node, it first propagates a negative token with the current accumulated attributes for persons present in the kitchen. Then it propagates the token `< + ..., persons( [Bob, ...]  )  >` where Bob is added to the list of persons.

```
locatedInRoom(?persons,?room) :-
      bagof( ?name,
             ( location(?id, ?room),
               userInfo(?name, ?id). ),
             ?persons).
```

Listing 6.10: Crime: rule using a bagof construct

Length Node



A *length node* is responsible for calculating the length of a list, for instance an accumulated list returned by a findall statement. A length node has a variable picker for selecting this list from the inserted token. Furthermore, the node consists of a string representing the name of the new fact that must be added to the token.

Consider a slightly modified rule represented in listing 6.11. The length statement determines the length of the list, the variable `?persons`, and adds a new fact with type equal to `number` to the inserted token. For instance, when the token that is inserted in this length node equals `< + persons( [Alice, Bob] ) >`, the token that is sent to the children of the length node is `< + persons( [Alice, Bob] ), number(2) >`.

```
locatedInKitchen(?number) :-
      findall( ?name,
              ( location(?id, Kitchen),
                userInfo(?name, ?id). ),
             ?persons),
      length(?persons, ?number).
```

Listing 6.11: Crime: rule using a length construct

Production Node

A production node forms the last node built by parsing a single rule. This node is responsible for executing the corresponding consequences of that rule. The execution of these actions is explained in more detail in section 6.2.3.

6.2.3 Executing Consequences of Rules

All consequences of a single rule are stored in an *activation* which can be exchanged between several devices. When a new activation is added by the production node, it is kept in an *agenda* which executes the stored activations by using some *conflict resolution strategy*.

Whenever an activation is added, the agenda first checks if this activation is useful: this means that no conflicting activation is present. Two activations are said to be conflicting when the execution of both activations has no effect and could as well be neglected. When such a conflicting activation is already added to the agenda, this one is removed from the agenda and the new activation is not added.

Each agenda is initialised with a *scheduler* for determining the order in which the activations must be executed. For instance, an agenda using a `FifoScheduler` executes the activations in order of their addition. As a priority can be ascribed to the activations, another possible scheduler could execute activations in order of decreasing priority.

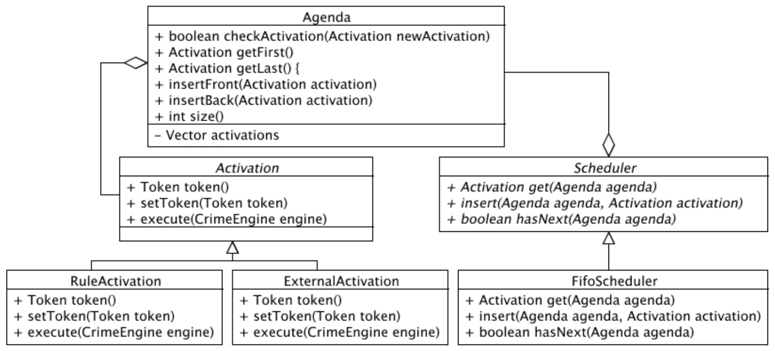

Agenda
+ boolean checkActivation(Activation newActivation)
+ Activation getFirst()
+ Activation getLast() {
+ insertFront(Activation activation)
+ insertBack(Activation activation)
+ int size()
– Vector activations
Activation
+ Token token()
+ setToken(Token token)
+ execute(CrimeEngine engine)
Scheduler
+ Activation get(Agenda agenda)
+ insert(Agenda agenda, Activation activation)
+ boolean hasNext(Agenda agenda)
RuleActivation
+ Token token()
+ setToken(Token token)
+ execute(CrimeEngine engine)
ExternalActivation
+ Token token()
+ setToken(Token token)
+ execute(CrimeEngine engine)
FifoScheduler
+ Activation get(Agenda agenda)
+ insert(Agenda agenda, Activation activation)
+ boolean hasNext(Agenda agenda)

Figure 6.6: Uml diagram of agenda classes

6.3 Federated Fact Spaces

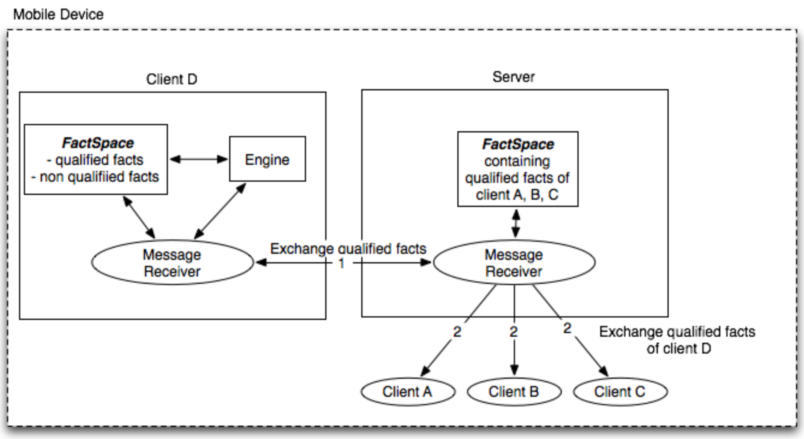
This section describes the communication necessary for the exchanging of facts residing in fact spaces of several devices. The previous section presented the working of an inference engine, in order to understand how applications work internally. Here we explain how such an inference engine can be extended in order to co-operate with other applications. We based the implementation of this distribution on the client-server architecture. Each application is represented by a client and must be connected with a local server. This server is introduced to allow the transiently sharing of facts between both clients residing on the same device and clients residing on different devices. The remainder of this section highlights the implementation of the communication layer implementing the federated fact space provided by CRIME.

6.3.1 Exchanging of Facts

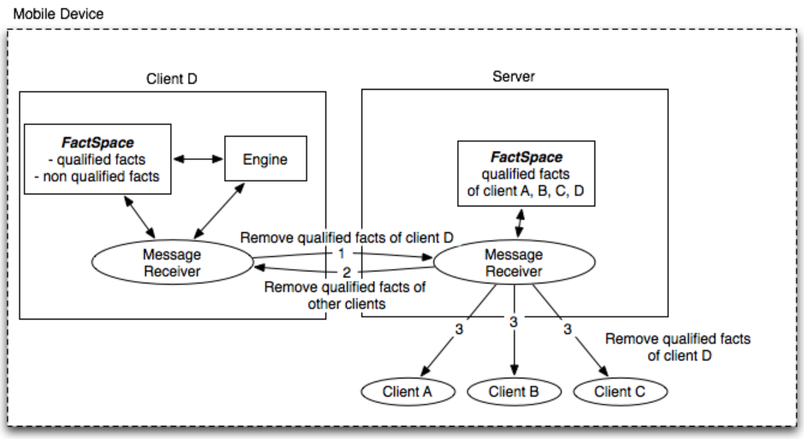
Using One Server

As is already mentioned, all *quantified facts* are exchanged between CRIME applications. Consider an example configuration shown in figure 6.7(a): a client application and server are running on the same device. When they connect, all quantified facts residing in the client's fact space are sent in one activation to the server. The *message receiver* of the server is responsible for receiving this activation and distributing it to other connected clients, which are kept in a *client pool*. While the message receiver is taking care of receiving sent messages, the server sends an activation to the newly connected server which contains all quantified facts that are residing in the fact space before the connection is established. So, in the example setting depicted in the figure, the server's fact space contains all quantified facts of clients A, B and C. These quantified facts are sent in an activation to client D in order to be asserted to its local fact space. The messages receiver of client D receives this activation and adds it to the agenda of that client.

When client D and the server disconnect, the exchanged facts must be retracted as is depicted in figure 6.7(b). First of all, all quantified facts of client D must be retracted as this exchanged context information is no longer guaranteed to be true. This retraction must take place in the fact space residing on the server, as well as on the fact spaces of all connected clients. The message receiver receives this messages and adds a new activation representing the retraction of these facts in each of their agendas. Furthermore, the quantified facts exchanged with the server after connecting must be retracted from client D. This are the facts of the other connected clients A, B and C. Performing these steps guarantees the truth in the context information represented by facts residing in the fact space.



(a) Client D connects with server



(b) Disconnection between client D and server

Figure 6.7: Communication with one server

Between Several Servers

The previous paragraph discussed the communication between an activation and a server. When moving towards a distributed environment, several servers are needed as at least one server must be running on each device. This paragraph discusses the communication and facts that need to be exchanged between two servers, assuming that the client-server communication is performed the same way as described in the previous paragraph.

Consider two servers possibly residing on the same device as is illustrated in figure 6.8(a). This figure presents the communication performed when two servers connect. Note that the connection mechanism itself is not important for understanding the communication, the mechanism itself is discussed in section 6.4. When server A and server B connect, the quantified facts residing in their fact spaces must be exchanged. We explain the several communication steps only in one way, namely for server A receiving the quantified facts of server B. The same reasoning holds for server B.

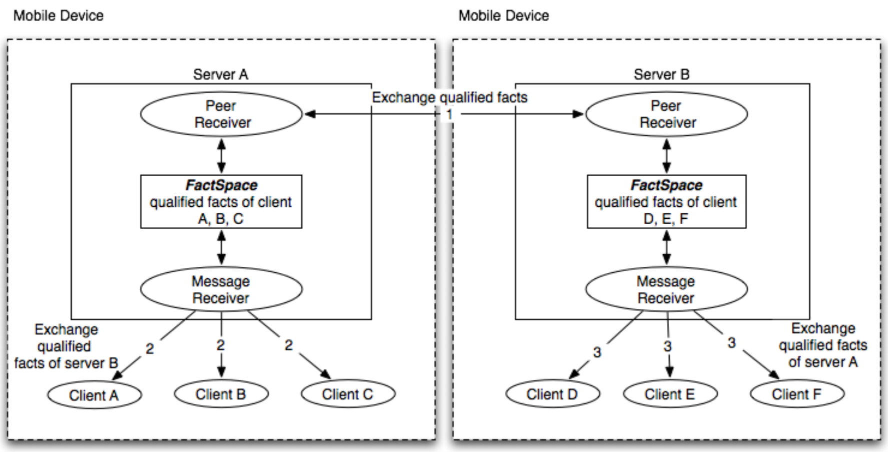
First of all, the message receiver of server A receives a new activation from server B. This activation contains all quantified facts of server B that must be asserted to the fact space of server A. Hence, the peer receiver of server A instantiates a new activation and adds it to the agenda. When this activation is executed by the engine residing on server A, this activation is propagated to all clients and servers residing in the server's client pool. This client pool contains the following connections: a connection to its connected clients A, B and C, as well as a connection with server B. Hence, the activation is sent to all three clients and the connected server B. As server B is the server who originally sent the activation, it won't add this activation to its agenda in order to prevent looping. Server B is able to detect and prevent looping as the name of the server that has sent an activation is sent together with the activation.

The performed communication when two servers disconnect is presented in figure 6.8(b). Here again, we do not discuss how the disconnection of two servers is detected as this is explained in the next section. We only consider server A, as the communication is identical for server B.

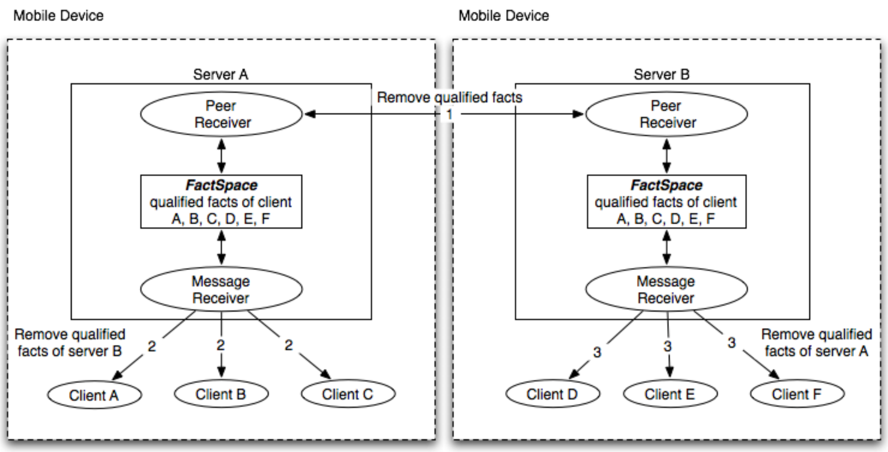
When the two depicted servers detect their disconnection, all exchanged facts must be retracted in order to maintain the truth as is explained in chapter 5. First of all, server A retracts all quantified facts from its fact space. Subsequently, an activation is sent to all connected clients which performs the retraction of the quantified facts of clients D, E and F.

6.3.2 Network Layer

The classes implementing the exchange of activations are shown in figure 6.9. As receiving messages on the client side and the servers side are very similar, we have opted to combine the common functionality in a `Receiver` class. When inventing new common functionalities it is sufficient to implement these in the `Receiver` class instead of implementing this two times for the server and the client. As the communication between two servers is slightly different than between a server and a client, a `ReceiveMessagePeer` is implemented which is used to only receive messages from a connected server. The `Server` class keeps track of all connected clients and servers by making use of the `ClientPool` class. The `Client` class has private field `engine` in order to propagate the received activations to the `CrimeEngine`.



(a) Connection established between server A and server B



(b) Disconnection between the servers

Figure 6.8: Communication with several servers

6.4 Service Discovery

Crime applications need to start a server in order to be able to communicate with eachother. Every Crime application residing on a device connects with this local server and the connection with this server is assumed stable. This assumption is not in contradiction with the ambient environment because it does not involve other devices. As shown in the previous section, the quantified facts of an application are sent to this server and transiently shared with other devices in the approximation. In this section we zoom in how client-server and

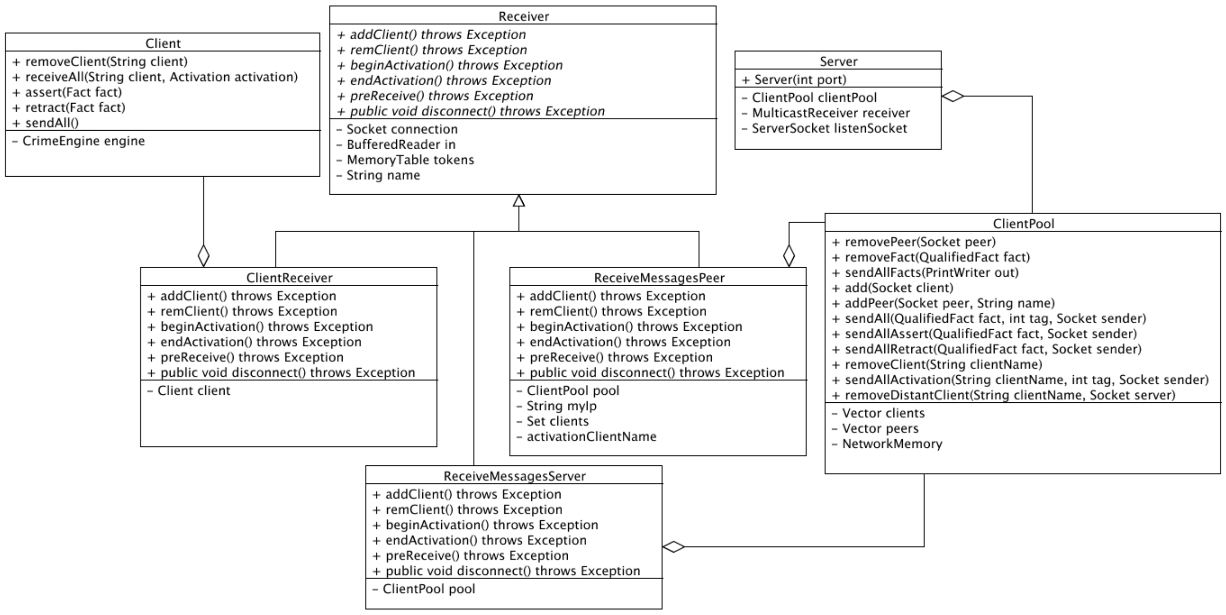

Client
+ removeClient(String client)
+ receiveAll(String client, Activation activation)
+ assert(Fact fact)
+ retract(Fact fact)
+ sendAll()
– CrimeEngine engine
Receiver
+ addClient() throws Exception
+ remClient() throws Exception
+ beginActivation() throws Exception
+ endActivation() throws Exception
+ preReceive() throws Exception
+ public void disconnect() throws Exception
– Socket connection
– BufferedReader in
– MemoryTable tokens
– String name
Server
+ Server(int port)
– ClientPool clientPool
– MulticastReceiver receiver
– ServerSocket listenSocket
ClientReceiver
+ addClient() throws Exception
+ remClient() throws Exception
+ beginActivation() throws Exception
+ endActivation() throws Exception
+ preReceive() throws Exception
+ public void disconnect() throws Exception
– Client client
ReceiveMessagesPeer
+ addClient() throws Exception
+ remClient() throws Exception
+ beginActivation() throws Exception
+ endActivation() throws Exception
+ preReceive() throws Exception
+ public void disconnect() throws Exception
– ClientPool pool
– String myIp
– Set clients
– activationClientName
ClientPool
+ removePeer(Socket peer)
+ removeFact(QualifiedFact fact)
+ sendAllFacts(PrintWriter out)
+ add(Socket client)
+ addPeer(Socket peer, String name)
+ sendAll(QualifiedFact fact, int tag, Socket sender)
+ sendAllAssert(QualifiedFact fact, Socket sender)
+ sendAllRetract(QualifiedFact fact, Socket sender)
+ removeClient(String clientName)
+ sendAllActivation(String clientName, int tag, Socket sender)
+ removeDistantClient(String clientName, Socket server)
– Vector clients
– Vector peers
– NetworkMemory
ReceiveMessagesServer
+ addClient() throws Exception
+ remClient() throws Exception
+ beginActivation() throws Exception
+ endActivation() throws Exception
+ preReceive() throws Exception
+ public void disconnect() throws Exception
– ClientPool pool

Figure 6.9: Uml diagram of network layer

server-server connections are established in the current implementation.

6.4.1 Client-Server Connection

The implementation of Crime uses sockets in order to perform communication between the client and the server. In order to facilitate the startup routine of the server a dedicated port is foreseen to establish a connection, which is *2040* in the current implementation. This port has been chosen randomly and can be changed by supplying an other *port number* when starting the server application.

The client in his turn is also aware of the default port to connect with, again this port number can be changed by supplying another port number on startup. When the client is started, it tries to connect to the server by using a socket. When the server is not reachable, it starts to poll this port until the server is reachable. One reason for the server not to be reachable is that the server application has not yet been started. The *polling* mechanism has been introduced in order to be able to start the server after the client application has been started. This mechanism has proven to be extremely handy when debugging.

6.4.2 Server-Server Connection

In order to provide CRIME applications with the necessary means to cope with the ambient environment, servers within the approximation connect automatically with eachother. As the used wireless technology is *wifi* a logical step was to use the *multicasting* facilities provided by this technology in order to detect other CRIME servers. Therefore we ascribe each server with a multicast object which sends out small packets containing the server's port number to all co-located servers, the *ip* address of the server can be derived from the packet received.

Detecting and receiving these multicasted packages is handled by the `MulticastReceiver`, which on the detection of a *new* package starts a connection with the associated server, in case that combination of the *ip* address and *port number* is lower than its own. Note the necessity of a mechanism in order to determine which server makes the connection. The `MulticastReceiver` keeps track of all connected servers in order to avoid connecting multiple times with the same server. In the case that they both connect to eachother this would result in unnecessary traffic.

6.5 Example Applications

6.5.1 Implementing an Application

In order to ease the implementation of applications with CRIME, some helper classes are facilitated. A first class is the `Main` class which implements a main method for starting up a CRIME client and initialising an inference engine. Furthermore, this class provides methods to assert and retract facts.

A second important class is the abstract `Action` class which must be used in order to write user-defined actions. This class has two abstract methods, namely `activated` and `deactivated` which both expect a vector of arguments. Consider an example to create a user-defined action `MyAction` which prints *Hello World!* when activated and *Goodbye!* when deactivated. The full implementation of this rule is provided in listing 6.12. Note that this action does not expect any arguments, hence the argument vector is not used. More complex actions use this argumentlist in order to provide more complex behaviour, for instance a simple use of these arguments could be to display the users residing in a room.

```
public  class MyAction extends Action {
    public void activated(Vector arguments){
        System.out.println("Hello World!");
    }
    public void deactivate(Vector arguments){
        System.out.println("Goodbye");
    }
}
```

Listing 6.12: Hello world action in CRIME

An example rule of the use of this newly defined `MyAction` action is shown in 6.13. In order to use this action in CRIME rules, it is sufficient to compile this class. When the user-defined action is detected during the parsing phase, it is *dynamically loaded* into the CRIME application and triggered when needed. Note that although the implementation of the action does not use the arguments in the example rule, the `?bar` variable is passed as an argument. We deliberately chose not to test upon the number of arguments that an action expects. This leaves the door open to implement actions which receive a variable size of arguments.

```
:MyAction(?bar) :-
        foo(?bar).
```

Listing 6.13: Using the hello world action

6.5.2 Bluetooth Detection

In the application presented in the previous chapter, we have used bluetooth in order to detect the users in the system. This bluetooth detection mechanism has been implemented by a CRIME application, it constantly scans for bluetooth devices in the approximation by making use of a PYTHON script. Pseudo code for this implementation is shown in listing 6.14. The detection widget simply finds all new devices and asserts these devices as facts to the engine. It also finds all devices which where lost and retracts these devices from the engine.

```
while(true) {
        Main.assert(detectNewDevices());
        Main.retract(lostDevices());
}
```

Listing 6.14: Bluetooth detection

6.6 Conclusion

In this chapter we have presented the most important aspects of the implementation. In a first step we have shown how rules are transformed into a parse tree by making use of ANTLR. A second step involved the transformation of this parse tree into a RETE network. We have ensured that the addition of new expressions is easy by placing the responsibility of building the RETE network by the parsed expressions. The only requirement for adding new expressions is to extend from a provided class `Expression` and adjust the parser for that new expression. Subsequently, we have explained how the nodes of this RETE network propagate their tokens through this network. Again these node can be easily extended by deriving from the `ReteNode` class and implementing the `propagate` method. A third section showed how federated fact spaces are implemented and how the communication between servers and clients is conceived. By implementing this layer in a dedicated `NetworkEngine` which derives from the normal `CrimeEngine`, we can reuse the `CrimeEngine` when porting our code to devices which make use of another communication technology. In a last section we discussed which classes provide the functionality in order to extend the Crime language with user-defined actions. The overall structure of our code is made extensible and applications-specific actions can be written easily. The implementation of context providers is eased by the `Main` class which provides the programmer with abstraction for publishing and retracting facts easily. In the following chapters we present specific extensions to the basic Crime language to improve the ability to cope with the mobile environment.

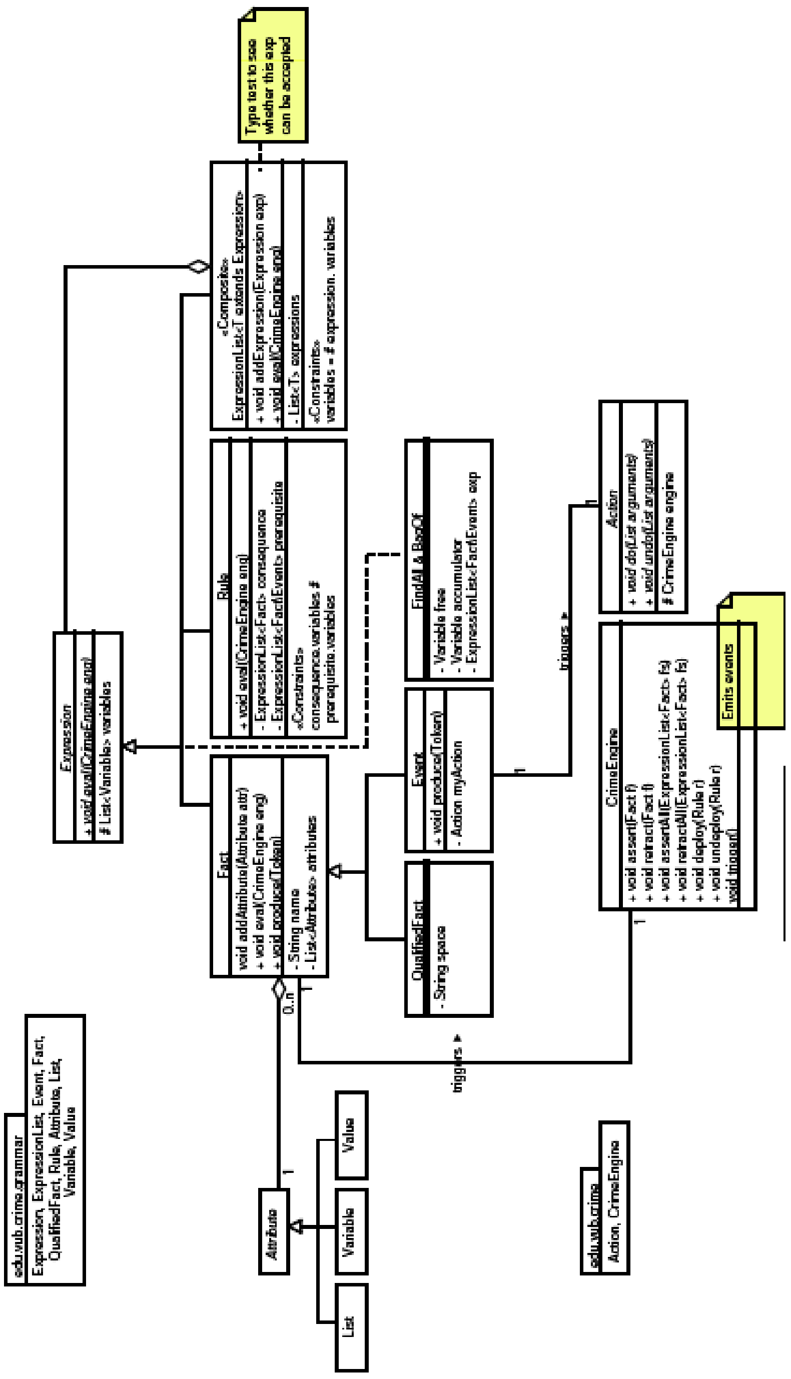

Type test to see whether this exp can be accepted
«Composite»
ExpressionList<T extends Expression>
+ void addExpression(Expression exp)
+ void eval(CrimeEngine eng)
- List<T> expressions
«Constraints»
variables = # expression. variables
Rule
+ void eval(CrimeEngine eng)
- ExpressionList<Fact> consequence
- ExpressionList<Fact|Event> prerequisite
«Constraints»
consequence.variables # prerequisite.variables
Expression
+ void eval(CrimeEngine eng)
# List<Variable> variables
FindAll & BagOf
- Variable free
- Variable accumulator
- ExpressionList<Fact|Event> exp
Action
+ void do(List arguments)
+ void undo(List arguments)
# CrimeEngine engine
triggers ▸
Emits events
Event
+ void produce(Token)
- Action myAction
CrimeEngine
+ void assert(Fact f)
+ void retract(Fact f)
+ void assertAll(ExpressionList<Fact> fs)
+ void retractAll(ExpressionList<Fact> fs)
+ void deploy(Rule r)
+ void undeploy(Rule r)
void trigger()
Fact
void addAttribute(Attribute attr)
+ void eval(CrimeEngine eng)
+ void produce(Token)
- String name
- List<Attribute> attributes
QualifiedFact
- String space
0..n
1
1
1
1
triggers ▸
edu.vub.crime.grammar
Expression, ExpressionList, Event, Fact, QualifiedFact, Rule, Attribute, List, Variable, Value
Attribute
Value
Variable
List
edu.vub.crime
Action, CrimeEngine

Figure 6.10: Uml diagram of the classes making up the parse tree

Chapter 7

Crime Time

This dissertation advocates the use of a distributed reasoning engine in combination with a truth maintenance system in order to ease the development of context-aware applications. In the previous chapter we have introduced the CRIME language which implements the Fact Space Model for reasoning about context in a distributed environment where stable connections can not be taken for granted and disconnections happen frequently. A limitation of CRIME is that it only allows reasoning about the *current* context, whereas past events may contribute useful information to make decisions about the present situation. The extension to CRIME presented in this chapter borrows techniques from HALO, an aspect-oriented extension to CLOS which provides support to reason about the execution history of a program, to permit CRIME programs to reason about past context information. The extension of CRIME, enriched with HALO's temporal operators is dubbed CRIME TIME.

The first section of this chapter introduces an additional feature to the jukebox example, used in the previous chapter to explain CRIME, that motivates the need to be able to reason about past events. Subsequently, we discuss existing approaches from aspect-oriented programming that incorporate such support. First of all REFLEX is presented: this is a framework that allows implementing context-aware aspects. These implement functionality expressed in terms of the past program context. Next HALO is presented which is a logic-based aspect language enriched with temporal operators enabling a declarative implementation of context-aware aspects. Since HALO builds upon a RETE network and CRIME has a similar overall structure, the introduction of a subset of its temporal operators facilitates CRIME with the same expressive power HALO exploits concerning these temporal operators. To conclude an overview of future work is given.

7.1 Motivating Example

Reconsider the jukebox application described in chapter 5. The jukebox in the apartment of Alice, Bob and Carol has a playlist containing the musical preferences of all present persons. The jukebox is smart enough to arrange the playlist so that songs acceptable for all three students are played. In order to avoid playing the same songs over and over again, the jukebox should remove tracks that persons have heard while they were previously in the room.

This extra functionality of the jukebox example can be written in Crime by keeping track of past events manually. In order to eject previously heard songs, three extra rules are needed. First of all, for each person in the living room the played songs should be stored in order to be able to delete them from the playlist whenever the user enters the room again. This can be realised by adding facts `played(?person, ?song)` to the working memory for every song that is played during the time a person is present in the living room. The Crime rule shown in listing 7.1 implements this behaviour: the application-specific action `UpdatePlayed` is responsible for asserting these facts. Note that a user-defined action is needed as those asserted fact may not be retracted when the user leaves the living room. So, when a fact `location(?person, Living room)` is retracted, the asserted fact `played(?person, ?song)` must reside in the working memory as this fact contains information that contributes to the composition of the playlist when that person enters the room again. This can be realised by not implementing the body of the `deactivate` method of the application-specific action.

```
:UpdatePlayed(?person, ?song) :-
            playSong(?song),
            location(?person, Living room).
```

Listing 7.1: Crime: rule for keeping track of heard songs

A second rule is needed in order to transform these asserted facts to `doNotPlay(?person, ?song)`. This transformation is needed for two reasons: a first one relies in the fact that the jukebox must be informed that those songs may not be played. Furthermore, the songs that were played when the person was previously in the room should not be distinguished with the one that are played when he has re-entered the room. Introducing this transformation resolves these problems as the previously played songs are represented by `doNotPlay` facts – these songs may be played when the user enters the room again – whereas the currently played songs that must be removed the next time the person enters the room are kept in `played` facts.

Listing 7.2 contains a Crime rule responsible for performing this transformation. The `activate` method of the user-defined action `DoNotPlay` asserts `doNotPlay` facts and retracts the corresponding `played` facts from the working memory. This application-specific action also destructively removes those songs from the playlist when the user enters the living room.

```
:DoNotPlay(?person, ?song) :-
```

```
          played(?person, ?song),
          entering(?person, Living room).
```

Listing 7.2: Crime: rule for transforming facts

The songs that were deleted from the playlist may be played when the user enters the living room a next time. So, a third rule is needed to retract the `doNotPlay` facts from the working memory. This retraction is performed whenever the user leaves the living room. Here again, an application-specific action `RemoveNotPlay` is needed as only its `activate` method may be implemented. Listing 7.3 shows the Crime rule for this last constraint.

```
:RemoveNotPlay(?person, ?song) :-
          doNotPlay(?person, ?song),
          leaving(?person, Living room).
```

Listing 7.3: Crime rule for retracting removed songs

Implementing this extra functionality in Crime is not straight-forward as keeping tracks of the heard songs must be implemented manually: an explicit rule is needed to assert facts for every song a person has heard while he is in the living room. Furthermore, those facts must be transformed to avoid confusion about songs that are currently deleted, and hence may be played again the next time, and those that are just played and may not be heard the next time the user enters the room. Extra care must be taken as application specific actions are needed for asserting facts, as a normal assert would be retracted when the prerequisites of the rule are no longer met, so for instance when the user is no longer located in the living room.

In order to implement these functionalities in a more expressive way, new primitives are added to the basic implementation of Crime. Before delving into the extension of the language itself, we describe the approaches in aspect-oriented programming we have based Crime Time on.

7.2 Related Work

The need to reason about past program state in order to correctly handle events does not only manifest itself in distributed context-aware systems. An interesting parallel can be made with the definition of business rules using aspect-oriented languages. An example of such a business rule is a web-shop application where the discounts a customer receives upon checkout should depend on whether a discount was active when the user added the item to its shopping cart. This strategy is to be preferred over taking into account discounts active at the checkout, since customers respond badly when they notice that items they have selected when a discount was active, have become more expensive [64, 31].

Similar to the Crime language explained in chapter 6, aspect-oriented pointcut languages allow responding to current active events (in this case in the program execution) using a combination of `execution` and `cflow` predicates. However, they fall short when events are considered relevant which are no longer active (i.e. they are no longer on the dynamic call stack). The remainder of this section presents context-aware aspects, a framework to allow pointcut expressions to be enriched with past context information. Subsequently we present HALO, an event-based aspect language which builds upon the idea of context-aware aspect, yet introduces them in a logic-based pointcut language to enable a declarative programming style.

7.2.1 Context-aware Aspects

Context-aware aspects introduce an extensible pointcut language where context information can be aggregated into a *context snapshot*. These snapshots can be used to determine whether a pointcut occurs in a conceptual context which is no longer necessarily tied to the dynamic call stack [64]. Whereas the framework described by Tanter *et al.* seems very general, its imperative style is in contrast with the declarative rules adopted in CRIME. For instance, the need to manually snapshot context at certain points in time imposes an imperative style where programmers are actively considering how reasoning about past events should be facilitated, whereas CRIME promotes a declarative style to describe context dependencies. In fact, one of the reason for its conception was precisely to untangle the complex imperative code typically found in event handlers of context-aware systems.

Consider the example of switching off the sound of a mobile phone while being in a silent context. The implementation as a context-aware aspect is shown in listing 7.4. In this code, a pointcut is defined which describes the join points when the `getSoundLevel` method of a mobile phone is invoked and the `SilentCtx` is active. In addition, an `around` advice is defined which simply bypasses the method invocation and returns *0* instead. As defined by the pointcut, only when the `silentCtx` is active the `around` advice is executed making sure the cellphone remains silent.

```
aspect SilentMode {
  pointcut call(): execution(int MobilePhone.getSoundLevel())
                         && inContext(SilentCtx);
  int around() : call() {
     return 0;
  }
}
```

Listing 7.4: REFLEX: `SilentMode` aspect

Note that being in a particular context does not bear any direct relation to any event on the dynamic callstack. Such a context can be activated simply by an event handler that activates a context. Despite the fact that the pointcut described in listing 7.4 does not occur in the cflow of the activation the pointcut is still influenced by it, as are all pointcuts until the context is explicitly deactivated.

Context aware aspects include the ability to take into account context information from the past. Consider the example to calculate the price of a phone call conducted during "happy minute". Phonecalls *initiated* during this promotional time are half-price. The code implementing the `HappyMinute` aspect using context-aware aspects is shown in listing 7.5. An aspect charge is defined which describes the join points where the `getPrice` method of the `PhoneConnection` class is called and the instantiated object was created when the `HappyMinute` context was active. The defined `around` advice specifies to first invoke the `getPrice` method, and then return only half of the price received from the `getPrice` method.

```
aspect HappyMinute {
  pointcut charge(): execution(double PhoneConnection.getPrice())
                              && createdInCtx(HappyMinuteCtx);
  double around() : charge() {
     return proceed() / 2;
  }
}
```

Listing 7.5: Reflex: `HappyMinute` aspect

In this example the context `HappyMinuteCtx` no longer needs to be active when the price is calculated, but instead it suffices that the connection was created in this particular context. A downside of this mechanism is that all context information needs to be snapshotted manually. For the above example every time a new phone connection is made, a snapshot needs to be taken. The framework provided by the context-aware aspects facilitate the programmer with an aspect-oriented style for taking these snapshots as can be seen in listing 7.6.

```
CtxActive createdInHappyMinute = new CreatedInCtx(new HappyMinuteCtx(), happyMinute);
happyMinute.addActivation(createdInHappyMinute);
```

Listing 7.6: Reflex: snapshotting HappyMinute context

7.2.2 HALO

HALO is an event-based aspect language which build upon the idea of context-aware aspects, yet introduces them in the context of a logic-based pointcut language [31]. HALO uses logical facts to represent the context state of the program and pointcuts are specified by making use of logical queries. Everytime a join point occurs, facts representing this join point are asserted to the fact base.

The logical rules representing the pointcuts are then checked against the fact base. When a pointcut has a solution based on the available join points, the advice body of the pointcut is executed. A pointcut can use logical variables in their advice as can be seen in the pointcut shown in listing 7.7. This pointcut intercepts invocations of the function `call` and prints out that someone called.

```
(at ((gf-call 'call ?caller))
    (format t ''someone called ~s'' ?caller)
```

Listing 7.7: HALO: Piece of advice for logging phone calls

Instead of forcing an imperative programming style in order to keep track of the contextual snapshots, HALO implicitly accumulates a history of join points. This implies that the fact base consists not only of the current join points but also records all past join points. Of course keeping these facts in the fact base is not enough: HALO also facilitates primitives to query these facts. These primitives, borrowed from metric temporal logic [11], were integrated into the pointcut language to allow aspects to reason about past events.

HALO incorporates the following subset of temporal operators:

- **all-past** operator
  syntax: `outer-pointcut (all-past inner-pointcut)`
  The functionality of the all-past operator is to collect *all* join points matched with the `inner-pointcut`. Furthermore, these join points are tested to have a timestamp smaller than the join points matching the `outer-pointcut`. For instance the example shown in listing 7.8 collects all `buy` calls of a user with a certain article as an argument that happened before a `buy` call of a user for the same article.

```
((gf-call 'buy (?user1 ?article))
 (all-past (gf-call 'buy (?user2 ?article))))
```

  Listing 7.8: HALO: example using all-past operator [31]

- **most-recent** operator
  syntax: `outer-pointcut (most-recent inner-pointcut)`
  The functionality of the most-recent operator is to accumulate all join points matched with the `inner-pointcut`. However, out of these accumulated match points only those join points are returned having the highest timestamp smaller than the join point of the outer pointcut. For instance the example shown in listing 7.9 captures the last `buy` call of a user that took place before a `checkout` call.

```
((gf-call 'checkout ?argsA)
 (most-recent (gf-call 'buy ?argsB)))
```

  Listing 7.9: HALO: example using most-recent operator [31]

- **since** operator
  syntax: `outer-pointcut (since inner-pointcut1 inner-pointcut2)`
  The first inner pointcut is evaluated against the join points matched by the outer pointcut. Moreover, the second inner pointcut is tested against the join points *in-between* the other join points – the one captured by `inner-pointcut1` and `outer-pointcut`. For instance, the code excerpt show in listing 7.10 determines all articles a user has watched between the time the user is logged in and the time the user buys an article.

```
((gf-call buy ?user)
 (since
      (most-recent
      (gf-call login ?user))
  (all-past
      (gf-call watch ?user ?article))))
```

Listing 7.10: HALO: example using since operator [31]

The temporal operators provided by HALO allows aspects to be written which depend on a certain context as seen in context aware aspects. The code excerpt in listing 7.11 is the implementation of the *happy-minute* example that is also used in the previous section.

```
(at ((end-gf-call getPrice ?phoneConnection ?result)
      (most-recent (create PhoneConnection ?phoneConnection)
                           (happy-minute-context-active)))
      (/ ?result 2))
```

Listing 7.11: HALO: `HappyMinute`

The difference with the context-aware aspects proposal is that HALO automatically provides mechanisms to keep track of the creation of the connection rather than requiring the programmer to keep track of this manually.

Although HALO's main concern are join points and aspects, HALO is highly related to the context model of CRIME where the context is modelled by facts and reactions are written in a declarative way. One of the contributions of HALO is the interpretation of temporal operators into a RETE network. The implementation strategies for the different temporal operators available in HALO served as a base to extend the derivation engine of CRIME to allow reactions on contextual changes happened in the past, as is presented in the following sections.

7.3 Crime Time

CRIME TIME is an extension to the CRIME language introducing temporal operators to reason about past context. The motivating example, given in section 7.1, can also be implemented more expressively by introducing these temporal operators. The remainder of this section is organised as follows: first a discussion about timestamping facts in a distributed environment is given. Subsequently, each of the temporal operators are highlighted, as well as their implementation in CRIME TIME. The last part of this section discusses the implementation of the modified jukebox application in more detail.

7.3.1 Time Model

CRIME is sculpted for operating in a mobile environment where applications on co-located devices contribute to the shared view of the environment by asserting facts into a federated fact space. A formalism for modelling time in a distributed system needs to be described as the introduction of HALO's temporal operators in CRIME requires that facts are tagged with a timestamp. This section discusses how time can be modelled in a distributed system and explains the implementation of the timestamping mechanism used in Crime Time.

Time in a Distributed Environment

At first it seems obvious to use the hardware clocks that are available in every device in a distributed environment for timestamping. However, synchronising several computers in a distributed setting can not be realised based on their hardware clocks. First of all, the synchronisation process itself takes an undeterministic period of time as the distance between the computers must be bridged. Moreover, this distance between the devices is generally unknown and causes delays between the two communicating devices. Furthermore, hardware clocks in a distributed system suffer from a phenomenon called *clock drift*. Clock drift can be caused by a difference in speed, temperature or even earth gravitation. When the hardware clock of a certain computer does not run in exactly the same speed compared to another computer clock, this difference in speed is accumulated and those clocks should be resynchronised. As is already mentioned, this synchronisation process is impossible in a distributed environment.

Even in case of one single computer, using the hardware clock of a computer to determine the order in which events took place doesn't guarantee a monotonic increasing relation between all events. The time in a computer system is updated after a predefined number of instructions, so it is possible that two events are executed at the same time-interval and hence tagged with an equal timestamp. As hardware clocks are not reliable for defining the order between several events, another technique for timestamping is obtained.

A second approach for determining a chronological order is based on counters and semaphores. Each computer has its own counter which is increased when an event is executed. As this counter is a shared resource between several applications on a single computer, semaphores are needed to overcome wrong counter updates.

For instance, consider the example shown in listing 7.12 where two processes want to attribute their events with a timestamp. Those processes are running on the same computer and retrieve the value of the global counter (line 6-7). The value of both local variables `ctrA` and `ctrB` equal `42` as at the moment `processB` retrieves the value of the shared resource, `processA` hasn't updated it yet. This can be explained as the computer switches between several processes. When the next instruction of `processA` is executed, the value of the global counter is updated to `43` and `eventA` receives this timestamp (line 8-9). Hence, after timestamp ten, the global counter has a value equal to `43` instead of 44. Furthermore, two events are ascribed with the same timestamp.

By introducing semaphores [20], this shared resource is dedicated to one process and other processes can only start using the resource when the first process has finished.

```
1  int counter = 42;
2
```

```
process A                                              process B
-----------                                            -----------
int ctrA;                                              int ctrB;
ctrA = read(counter);
                                                       ctrB = read(counter);
counter = ++ctrA;
eventA.setTime(ctrA);
                                                       counter = ++ctrB;
                                                       eventB.setTime(ctrB);
```

Listing 7.12: Problems using a shared resource

In a distributed environment, using these counters is not sufficient as events on different computers can be ascribed with the same timestamp. This problem is recognised and solved by Lamport [40] who introduces a technique to overcome these problems in a distributed system. *Lamport clocks* can be attributed to events by keeping in mind the following rules:

1. Ascribing timestamps to events of independent processes doesn't cause any problems as these timestamps are assigned concurrently and they're not exchanged between the processes.

2. The timestamping mechanism on one single computer by making use of the technique described above is correct as this process is executed sequentially.

3. When exchanging events between several processes, the event that is received by the process should ascribe a timestamp which is the maximum value of the counters on both processes. By taking this maximum value guarantees that a relationship is determined between the events on the computers and between the dependent processes, hence the order of the exchanged and local events can be rederived if desired.

   For instance, consider the timelines depicted in figure 7.1 which represent the values of the counters for each process. As we can see the first process, `P1`, and the second one are dependent as `P2` receives events of the first process. At the moment on which `P2` receives the first event `e1`, the value of its counter equals zero. So, this new event should be attached a timestamp equal to one. However, as this event is sent by the first process on timestamp `3`, Lamport indicates that this event must be ascribed the value four. This value describes the relationship that the event took place after the event timestamped with the value three on `P1`.

   For the second event that is exchanged between the first and second process, the timestamp used on `P1` is smaller than the one of `P2`, so the timestamp that is attributed to `e2` on the second process is the value of its counter increased, namely seven.

   Using Lamport clocks the values of the timestamps of the events on the same process are discontinu, however this causes no problem as there's still a relation between these events and their order can be determined.

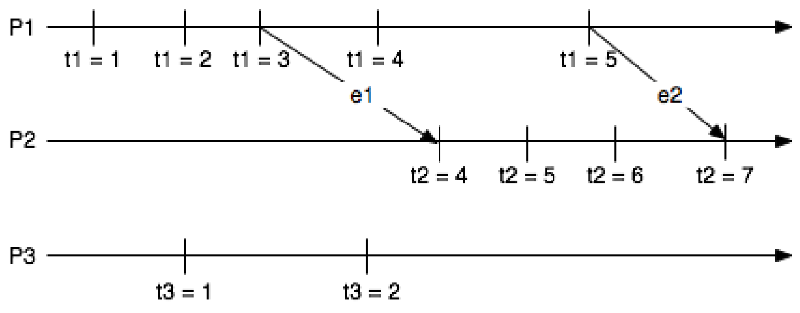


Figure 7.1: Lamport timestamps for three processes

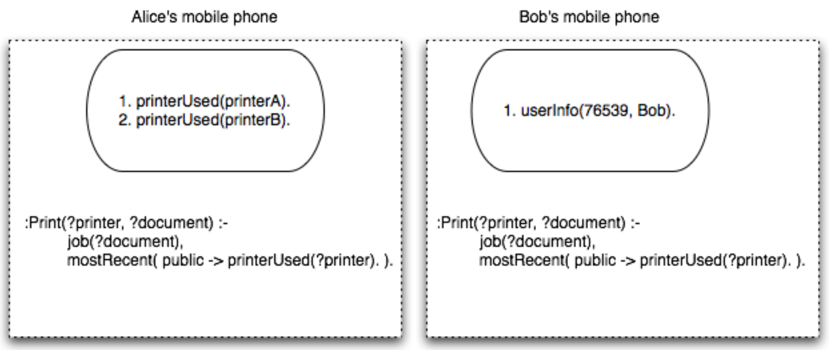
Notion of Time in Crime Time

As Crime operates in a mobile environment where devices share contextual information, introducing timestamps is based on Lamport. However, timestamping in Crime Time is more complicated as several actions can be exchanged in one single activation. This way of transferring suites of facts between several devices loses important information, namely the order in which those actions must be executed. Hence, Lamport timestamping can be used for ascribing a timestamp to the exchanged activations.

Crime Time increases its timestamp whenever a new activation is executed, as this activation is considered as a new event even in case it was retrieved from another device. All actions performed during that activation are attributed with the same timestamp.

Recall that the exchanging of all qualified facts must be performed in one single atomic operation, which is realised by giving all actions of that activation the same timestamp. However, this way of ascribing timestamps introduces some problems as the semantics of rules residing on different devices are no longer guaranteed to be the same.

Consider an example which is depicted in figure 7.2. The first part of this figure 7.2(a) depicts the fact spaces on two mobile phones and an application that is running on each of them. This application has a Crime rule which determines the printer that is last used by a person for printing his requested job. When the two devices connect, the qualified facts are exchanged, so in this example two `public` facts are received by the fact space residing on Bob's mobile phone. As the assertion of these facts is exchanged by using the same activation, these facts are ascribed with the same timestamp `3`. Figure 7.2(b) depicts both devices after they've connected. Now, suppose Bob requests a printing job by asserting the fact `job(‘‘document.pdf’’)` at timestamp 4. The prerequisites of the rule are all met and the rule is triggered. However, as both `printerUsed` facts that were exchanged have the same timestamp, the



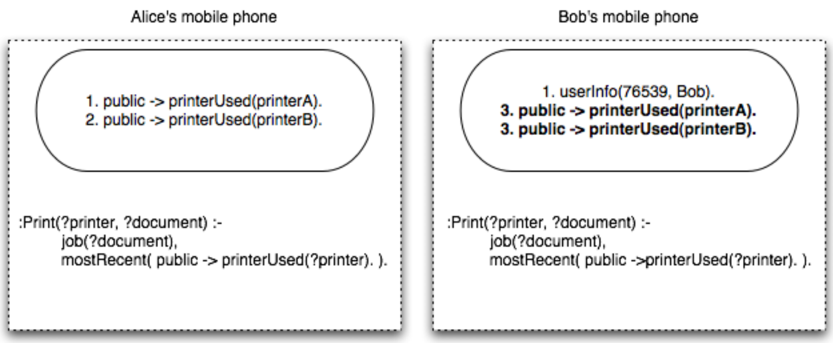
```
1. printerUsed(printerA).
2. printerUsed(printerB).

:Print(?printer, ?document) :-
        job(?document),
        mostRecent( public -> printerUsed(?printer). ).
```

```
1. userInfo(76539, Bob).

:Print(?printer, ?document) :-
        job(?document),
        mostRecent( public -> printerUsed(?printer). ).
```

(a) Initial configuration



```
1. public -> printerUsed(printerA).
2. public -> printerUsed(printerB).

:Print(?printer, ?document) :-
        job(?document),
        mostRecent( public -> printerUsed(?printer). ).
```

```
1. userInfo(76539, Bob).
3. public -> printerUsed(printerA).
3. public -> printerUsed(printerB).

:Print(?printer, ?document) :-
        job(?document),
        mostRecent( public ->printerUsed(?printer). ).
```

(b) Alice's mobile phone and Bob's mobile phone are connected

Figure 7.2: Different semantics caused by ascribing same actions of activations with the same timestamp

behaviour of the rule is not predictable.

This difference between semantics of rules residing on different devices is a problem which is not resolved in the current implementation of Crime Time. This problem only arises for facts that were asserted to the federated fact space before devices connect, after connection the semantics on both devices is guaranteed to be the same, as activations are immediately exchanged between co-located devices. However, this problem could be solved by ensuring to maintain the order of the actions of an activation when these are exchanged. Instead of ascribing the timestamp when the activation is executed, the timestamping process should then take place at the moment an action is performed. The communication part of Crime Time responsible for the exchanging of activations must be extended in order to overcome the problem that is described in this section.

Now that the timestamping mechanism used by Crime Time is explained, we can delve into the extension by introducing HALO's temporal operators in Crime. Crime Time introduces the following set of temporal operators: *sometime-past*, *most-recent* and *since*. Those operators can be used in the prerequisites of Crime Time rules. To support reasoning about past context, Crime Time attributes every fact with a *timestamp* which is assigned whenever the Rete engine executes an activation. As a consequence, several facts can have the same timestamp.

Note that the temporal operators are always implicitly parametrised by the fact that precedes them. That is to say, they have implicit access to the timestamp at which this fact was triggered. For instance, `f`$_1$`, timeOperator(f`$_2$`)` has one explicit argument f$_2$, and one implicit argument, namely the fact f$_1$ that preceded it.

7.3.2 Sometime-past Operator

The *sometime-past* operator takes one explicit argument that is timestamped with t$_2$. $f_1, sometimePast(f_2)$ matches facts so that t$_2$ < t$_1$ where t_1 is the implicit timestamp at which the fact f_1 is triggered.

Example

Consider the *printer* example rule shown in listing 7.13 that uses the `sometimePast` operator. This rule is used for determining a *default printer*, based on the number of times a user has used that printer before. This default printer is proposed to the user for printing his new job. When the user rejects this proposal, an other available printer is proposed.

The rule in listing 7.13 is triggered whenever a new printing job is scheduled. For instance when Alice wants to print a certain paper, this results in the addition of the fact `job(Alice, paper.pdf)`, triggering the rule below.

```
1  :Print(?file, ?usedPrinters, ?availablePrinters) :-
2       job(?person, ?file),
3       findall( ?usedPrinter,
4                ( sometimePast( printerUsed(?usedPrinter, ?person). ). ),
5                ?usedPrinters),
6       findall( ?availablePrinter,
7                ( printer(?availablePrinter). ),
8                ?availablePrinters).
```

Listing 7.13: Rule using `sometimePast` operator

The first `findall` statement on line 3 in the code excerpt, collects all printers that were previously used by Alice in the variable `?usedPrinters`. Determining which printers Alice has used in the past is realised by the `sometimePast` operator (line 4). Recall that a temporal operator takes one implicit argument, the fact `job(Alice, paper.pdf)` in this example, which is the point at which the job was scheduled and acts like an upper bound for the points at which the printers were used.

The `findall` statement starting on line 6 calculates all the available printers. In case Alice hasn't used any printer before, she can pick her favourite one from this set of printers. The implementation of that choice of printer is implemented by the user-defined action `Print`. This action proposes Alice's default printer, namely the one she has been using the most. This printer can be determined from the list of `usedPrinters`. When Alice agrees to print her file on a printer, for instance `printerA`, this action also adds a new fact `printerUsed(printerA, Alice)` to the working memory.

Consider the concrete scenario of the *printer* example that is depicted in figure 7.3. This figure represents the facts that are added at a certain time to the working memory. At timestamp one, the fact `job(Alice, document1)` is added to the working memory. At that moment in time Alice hasn't used any printer before, as can be seen in table 7.1. So, the printer application can't propose any default printer and Alice chooses one of the available printers, for instance `printerB`. This results in the addition of a new fact `printerUsed(Alice, printerB)` at timestamp two to the working memory, as is depicted on the timeline in figure 7.3.

When Alice requests a new printing job at time two, she already used `printerB` which is proposed to her as the default printer. However, Alice rejects this proposal and choses `printerC` for printing her second document which results in the addition of the fact `printerUsed(Alice, printerC)` to the working memory.

The next time Alice requests a printing job (timestamp three), the default printer is randomly chosen between the printers she has used before, because they both have been used once. This behaviour is implemented by the application-specific action `Print`. Another possibility could be to propose the printer she has last used in case of a tie. At timestamp nine Alice wants another document to be printed and her preferred printer is `printerB` which she has used thrice, however the default printer the application proposed to her is `printerC` although she has used that printer only once. This printer is proposed to her because `printerB` is no longer available, for example due to a defect.

timestamp	used printers	available printers	default printer
1	{}	{A, B, C}	none
3	{B}	{A, B, C}	B
5	{B, C}	{A, B, C}	B
9	{B, C, B, B}	{A, C}	C

Table 7.1: Used printers, available printers and default printer for *printer* example with `sometimePast` operator

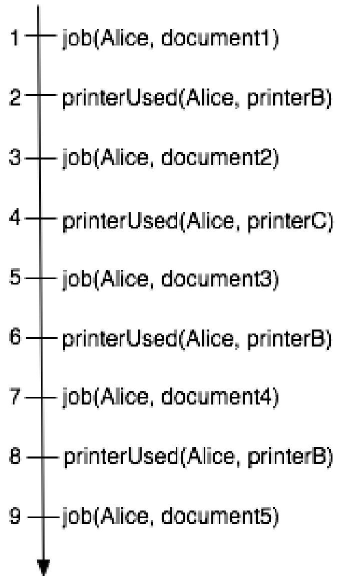

1 job(Alice, document1)
2 printerUsed(Alice, printerB)
3 job(Alice, document2)
4 printerUsed(Alice, printerC)
5 job(Alice, document3)
6 printerUsed(Alice, printerB)
7 job(Alice, document4)
8 printerUsed(Alice, printerB)
9 job(Alice, document5)

Figure 7.3: Timeline for *printer* example

Implementation in Crime Time

The network for the rule in listing 7.13 is depicted in figure 7.4. The `sometimePast` node in the network acts like a join node with an extra filter. First of all, the sometime-past node checks the variable bindings for the variables that occur in both the left and right prerequisites, `?person` in this example. In addition to the consistency check, the sometime-past node also filters based on the timestamps of the facts in the left and right memory of the sometime-past node. Recall that these timestamps are assigned to the facts whenever an activation is executed by the Rete engine.

Based on the network construction algorithm described in chapter 6, the sometime-past construct is compiled to a join node with the past context events stored in its right memory. Unlike a normal join node, elements in the right memory are not removed when the corresponding fact is retracted. This allows the right memory to serve as a cache of precisely those past events which may still be used in future derivations. Extending the join node to keep track of past context alone is not sufficient, since an additional check needs to be performed to guarantee that the timing constraints are upheld.

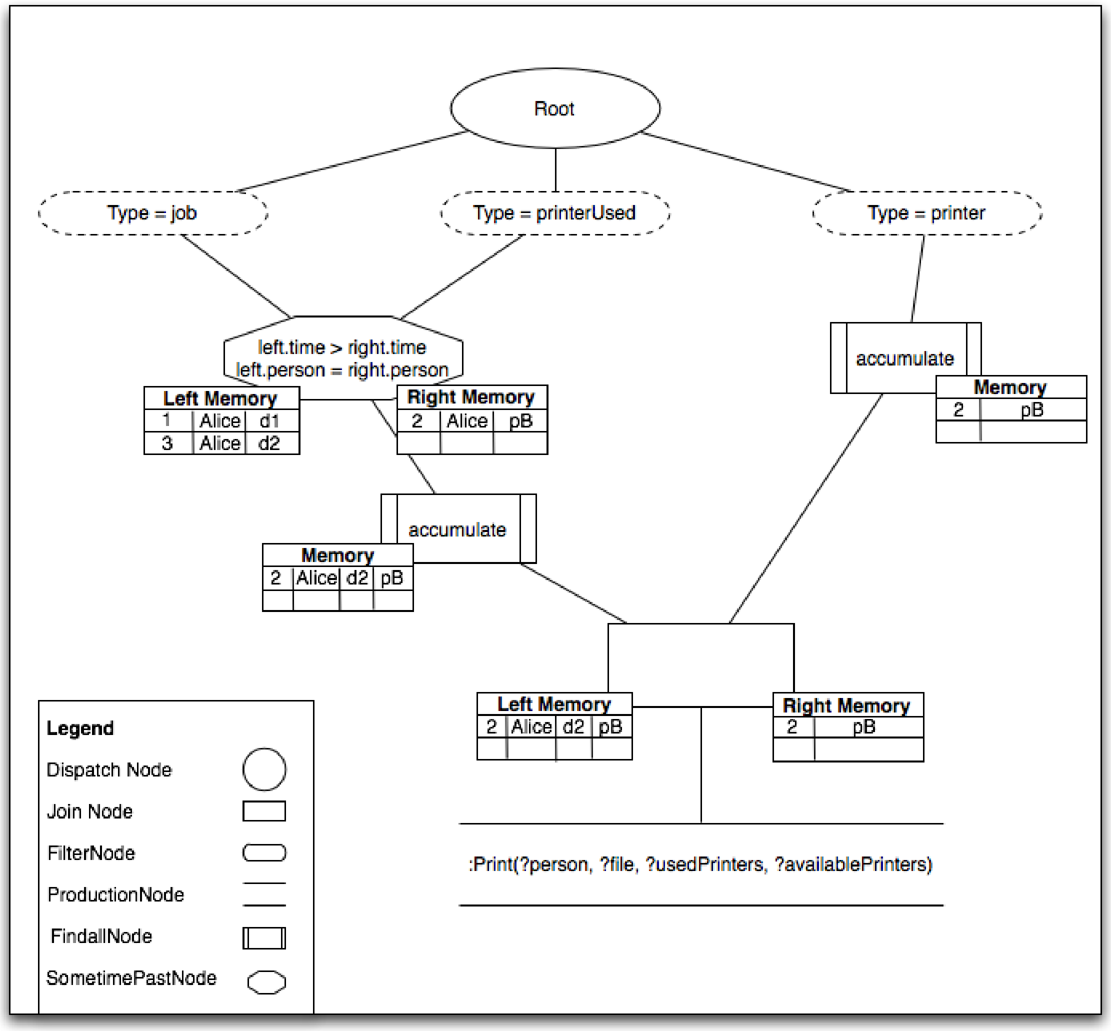


Figure 7.4: Network for printer rule with `sometimePast` operator

7.3.3 Most-recent Operator

This operator has similar semantics as the sometime-past operator discussed in the previous section. A *most-recent* operator has an explicit restriction that only the most recent matching fact can be returned.

Example

A modified version of the *printer* example is shown in listing 7.14. The default printer this rule proposes is the last one a person has used. This printer is determined by the `mostRecent` operator on the third line of the code. Note that there's no need for a findall statement as the most-recent operator only returns one single fact.

```
:Print(?file, ?printer, ?availablePrinters) :-
      job(?person, ?file),
      mostRecent( printerUsed(?printer, ?person). ),
      findall( ?availablePrinter,
               ( printer(?availablePrinter). ),
               ?availablePrinters).
```

Listing 7.14: Rule using `mostRecent` operator

Reconsider the printing example where Alice requests several printing jobs. When the application uses the rule with the most-recent operator shown in listing 7.14, the default printers it proposes are changed. The proposed printers are given in the fourth column of table 7.2. As the most-recent operator only returns one fact, the default printer is always the printer Alice has last used. However, at timestamp 9 there's no default printer proposed by the application as the last used printer, `printerB`, is no longer available. Hence the application-specific action `:Print` doesn't propose a default printer and lets Alice chose an available printer for her printing request.

timestamp	used printers	last used printer	available printers	default printer
1	{}		{A, B, C}	none
3	{B}	B	{A, B, C}	B
5	{B, C}	C	{A, B, C}	C
7	{B, C, B}	B	{A, B, C}	B
9	{B, C, B, B}	B	{A, C}	none

Table 7.2: Used printers, last used printer, available printers and default printer for *printer* example with `mostRecent` operator

Implementation in Crime Time

The network of the rule in listing 7.14 resembles the one depicted in figure 7.4 as the sometime-past and most-recent operator have similar semantics. However, there's a small difference in the way tokens in the right memory of the node are handled, as only the most recent one is passed to the children of this node. Furthermore, this network hasn't a findall node as the rule in listing 7.14 doesn't need a findall statement for accumulating all facts.

7.3.4 Since Operator

The syntax of the *since* can be written as `fact2 since(fact1, fact3)`. This operator takes two explicit arguments, f_2 and f_3, and matches events f_3 that occurred between f_2 and f_1 – with f_2 the fact that is implicitly taken as an argument.

Example

Consider an example application where an employee wants to request calls he has missed during the last time he wasn't present in his office. The rule shown in listing 7.15 determines those missed calls by using the `since` operator.

The rule is triggered whenever a person performs a requests for his missed calls. At this point in time, the system recalls the last time when the person entered his office (line 3 of the code excerpt). Recall that this line forms the implicit stop condition of the since operator. The timestamp of this entering-fact is used as the end of the since interval (line 4) of which the starting point is the last time the user left his office (line 5). The facts being sought are the telephone numbers which are accumulated by the `findall` statement of line 5.

```
1  :HandleCalls(?numbers) :-
2      request(?person, ''missed calls''),
3      mostRecent( enter(?person, ''office''). ),
4      since(
5            mostRecent( away(?person, ''office''). ),
6            findall( ?number,
7                     ( sometimePast( call(?number, ?person). ). ),
8                     ?numbers). ).
```

Listing 7.15: Rule using `since` operator

To exemplify the use of the rule in listing 7.15, consider the example below. The timeline depicted in figure 7.5 represents the facts that are added to the working memory and their corresponding timestamp. When Alice performs a requests for her missed calls, the fact `request(Alice, ''missed calls'')` is added to the working memory at timestamp 11. This addition results in the triggering of the rule, which code excerpt is shown in listing 7.15, and the `findall` statement determines all calls Alice has missed while she was absent. The telephone numbers that are passed to the user-defined function `HandleCalls` are `158` and `083`.

Implementation in Crime Time

The network of the rule given in listing 7.15 is depicted in figure 7.6. The *since node* resembles the join node of the beta part of the Rete network which combines two tokens. Just as the sometime-past and most-recent node, the since node performs an extra test on the timestamps of those tokens. The node verifies whether the timestamps of the `call`-facts is greater than the timestamp of the `away`-fact. As can be seen in the figure, the filter node that filters out the `enter` facts is used twice, once in combination with the first argument of

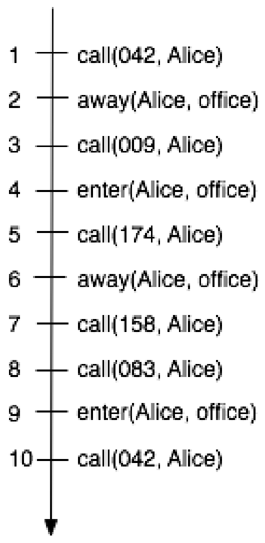


Figure 7.5: Timeline for *missed calls* example

the since statement, and a second time for the last argument of that temporal operator.

7.3.5 Implementing the Motivating Example

Recapitulate the jukebox application discussed in section 7.1 where users want to avoid the repetitive behaviour of the jukebox' song selection. This example application can be implemented more expressively using the temporal operators that are described in the previous section. The code excerpt in listing 7.16 is a rule which implements the desired behaviour for the application. The rule is triggered whenever a person enters the living room (line 2). At this point in time, the system recalls the last time when the person left the living room (line 3). This timestamp is used as the end of a `since` interval (line 4), of which the starting point is the previous time the user was spotted by the system (line 5). The fact being sought for in this interval are the songs that are accumulated by a `findall` statement (lines 6-8). These songs are then deleted from the current playlist (using the `DeleteFromPlaylist` context event handler) as they should not be repeated (line 1).

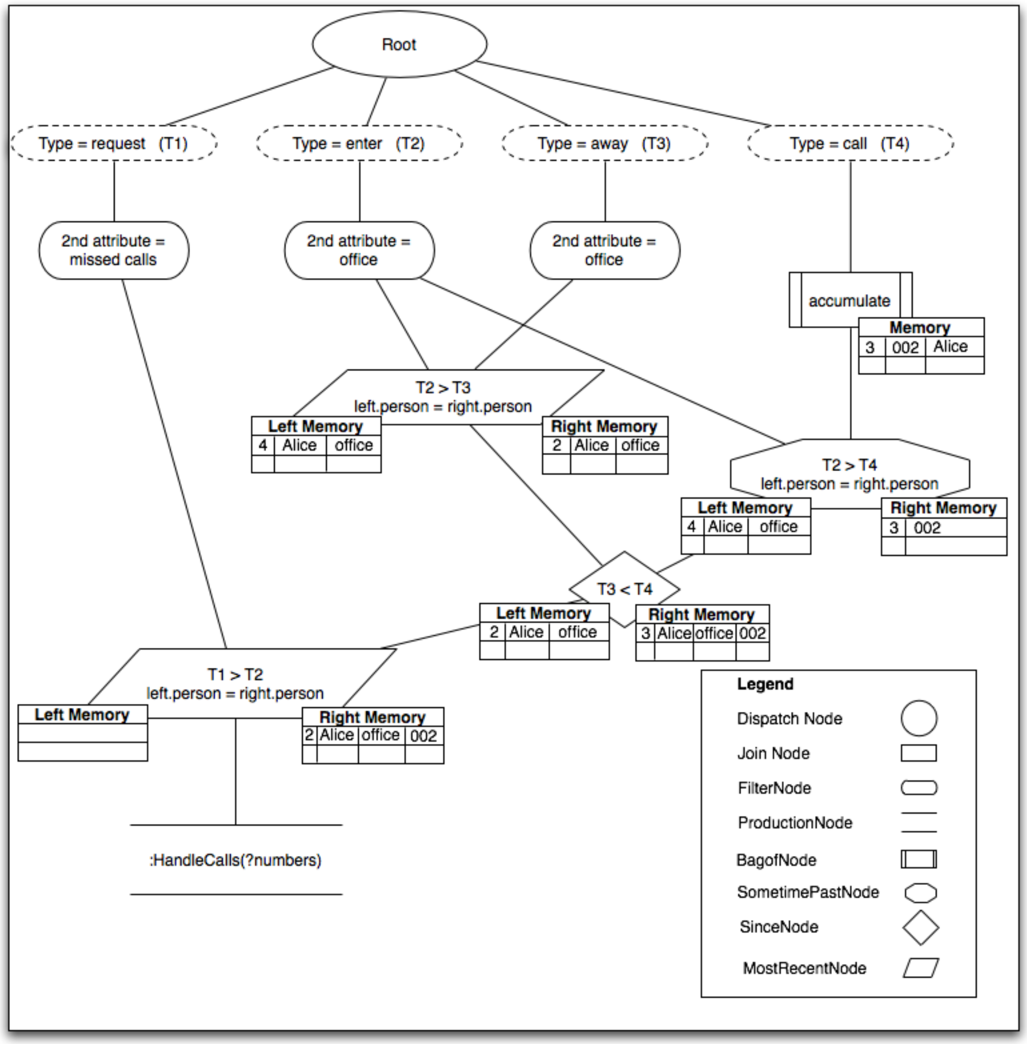


Figure 7.6: Network for *missed calls* example at timestamp 4

```
:DeleteFromPlaylist(?person, ?songs) :-
   location(?person, ''Living Room''),
   mostRecent( not location(?person, ''Living Room''),
               since(
                      mostRecent( location(?person, ''Living Room'') ),
                      findall( ?song,
                               ( sometimePast( played(?song). ). ),
                                ?songs). ). ).
```

Listing 7.16: Jukebox: implementation using temporal operators

The network for this *jukebox* example is depicted in figure 7.7.

7.4 Conclusion

Crime Time is an extension of the basic Crime implementation that is discussed in chapter 6. By extending this basic implementation with temporal operators, time-related declarative rules can be written as all facts are extended with a specific timestamp. Crime Time introduces temporal operators for reasoning about the past. This extension of Crime is inspired by the aspect-oriented programming language HALO where reasoning about the past is realised by implicitly keeping a context history. HALO introduces temporal logic extending the Rete algorithm and hence enabling reasoning about the past in a declarative way. The temporal operators of HALO are re-implemented in Crime Time.

The current implementation of Crime Time only introduces operators for reasoning about the past. A first extension relies in the introduction of temporal operators for reasoning about the future. The next chapter discusses Optimal Crime, another extension of the basic Crime implementation. Chapter 8 presents a language where the basic Rete algorithm is optimised by a technique, called scaffolding.

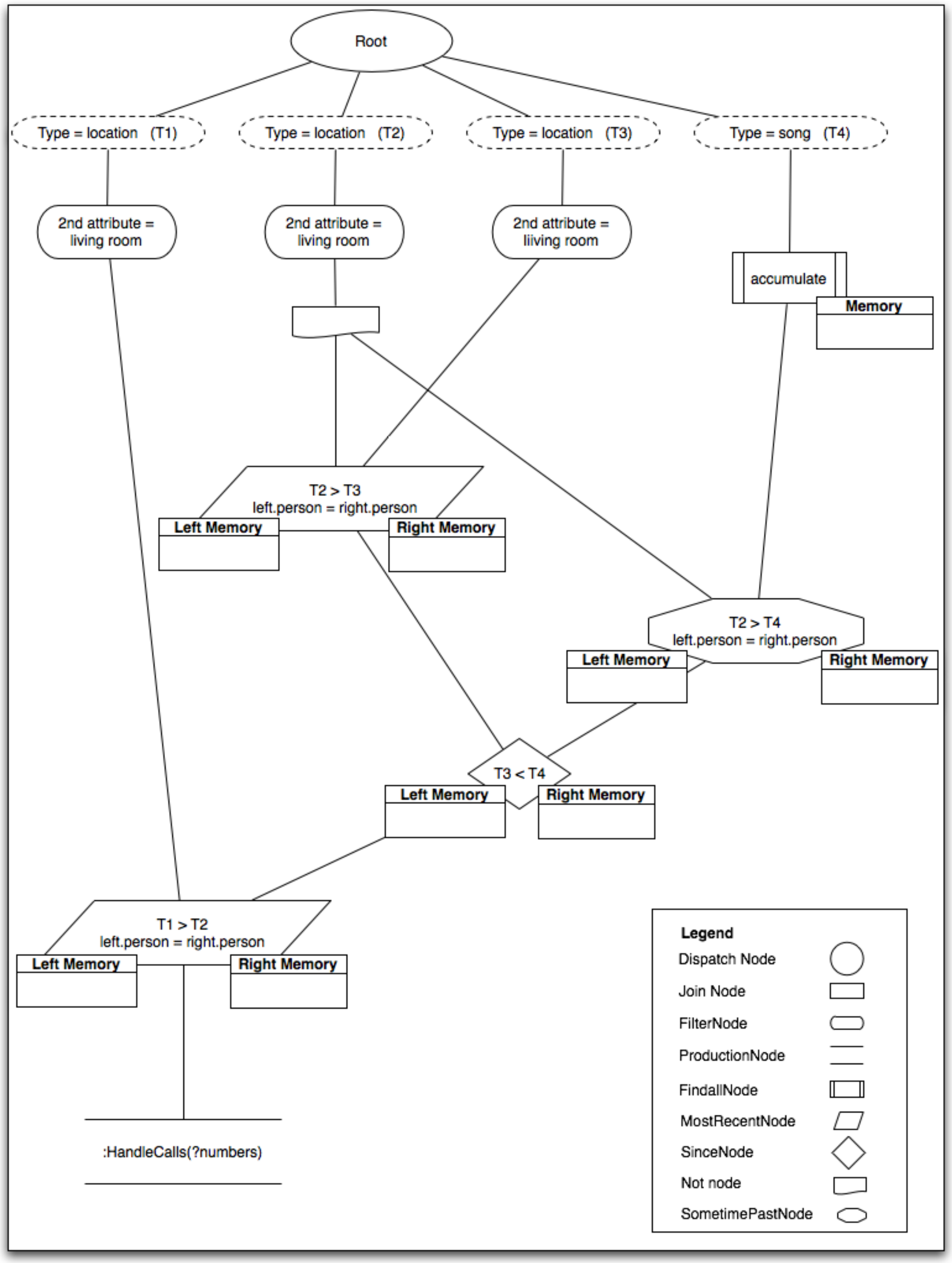


Figure 7.7: Network *jukebox* example using temporal operators

Chapter 8

Optimal Crime

This dissertation advocates the use of a distributed reasoning engine in combination with a truth maintenance system in order to ease the development of context-aware applications. In the previous chapters we have introduced the novel coordination language Crime which is sculpted to deal with the characteristics of a mobile environment. One of these characteristics is that connections are volatile, therefore intermittent disconnections happen frequently. Recall that facts from a co-located device are retracted when the device is disconnected and reinserted when it reconnects at a later time. Although care has been taken to optimise the matching phase of the inference engine by making use of the Rete algorithm, such removals and reinsertions of facts remain relatively costly.

This chapter presents an optimisation which is specifically geared towards the way Crime deals with volatile connections. First we present a technique known as "scaffolding the Rete network", and show how scaffolding can be incorporated into Crime. Subsequently we show the results of applying scaffolding to the Crime language.

Another source for optimisation are the temporal operators discussed in the previous chapter. Currently, Crime automatically assigns timestamps to the facts inserted into the working memory. This automatically timestamping of the various events in the distributed systems results in a monotone increasing time model as seen in chapter 7. When operating in this time model, optimisations can be applied by extending the scaffolding algorithm with these operators.

Although the optimisations presented here greatly improve the computational aspect of the programming language, the network load can also be optimised. In the future work section we present a possible optimisation to decrease the information exchanged over the network. We end this chapter with the conclusions distilled from the conducted experiments.

8.1 Scaffolding the Rete Network

The basic RETE algorithm introduces a first optimisation for the pattern matching phase of CRIME's inference engine. However, the join nodes of the RETE network perform a costly operation during this match phase. Each time an activation takes place – that is when a token is inserted in one of the node's memories – , the join nodes perform a variable consistency check of the inserted token with the tokens residing in the opposite memory as the token is inserted. As demonstrated in chapter 3, the insertion of a token into the RETE network does not only serve the assertion of new facts but also the retraction of facts.

In this section we improve the classic deletion algorithm described earlier by presenting a technique to perform the retraction of a token from the RETE network in *constant time*. This optimisation omits the match operations normally needed to perform the retract operation by making use of causal links. Subsequently we discuss an extension of this optimisation which constructs a parallel datastructure of deleted tokens in order to decrease their reinsertion time as well.

8.1.1 Constant Time Retractions

The RETE algorithm uses a clever cashing mechanism to decrease the number of matching operations required whenever a token is propagated through the RETE network. The join nodes in this network perform variable consistency checks when tokens are inserted. As retractions are modelled by the propagation of a negated token, these variable consistency checks are performed twice; One time when the fact is inserted and once when it is removed. The observation of these redundant checks is what forms the base of the optimisation presented in this section. Similar to the addition of a fact, the cost of a retract operation depends on the number of partially instantiated tokens residing in the network. For every join node, n matching operations must be performed, one for every token residing in the opposite memory. The cost of a match operation itself depends on the number of variable checks that need to be performed by the join node.

Recall the various steps needed to remove a certain fact and the corresponding tokens in the RETE network: First the fact is transformed to a token and tagged with a minus sign so that the nodes know a retract operation is in progress. When this token is inserted in the RETE network, it is propagated through the filter nodes until it is passed to a join node. This join node removes the corresponding token – with a plus sign as it was inserted – from the parent's memory that passed it on. Subsequently, the join node performs a variable consistency check by matching all entries from the other parent's memory against that token. For every found match, it propagates a negative token to its children. All these steps are depicted in figure 8.1.

The essence of the optimisation relies in the introduction of causal links

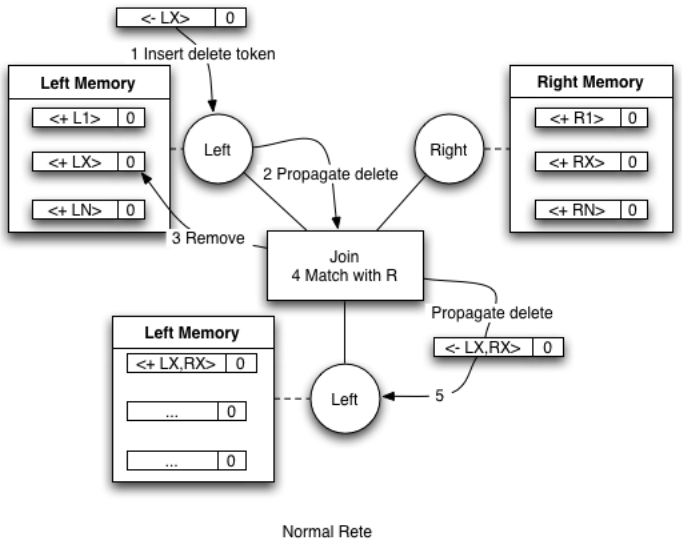


Figure 8.1: RETE normal retract

between tokens. Every token keeps a set of links to all the tokens that were derived from it. These links are used in order to omit the variable consistency checks performed by the join nodes when a negative token is propagated. Instead of performing a costly match operation, the join node can simply follow the causal links of the negated token in order to determine which tokens derived from it. This optimisation decreases the computations needed for the retract operation in the join node to $O(1)$. Note that the memory layout of the memories kept in the network should also support constant time deletions. In order to allow these constant time deletions, OPTIMAL CRIME uses a doubly linked list where tuples can delete and insert themselves in time $O(1)$.

An example of this optimisation is shown in figure 8.2. In this example two causal links are drawn, one from the token `<+ LX>` to the token `<+ LX, RX>` and one from `<+ RX>` also to `<+ LX, RX>`. In the first step of this example a token `<- LX>` is propagated to the left node. This node simply passes this token to the join node in step two. The join node instructs the left memory to remove the token at hand, because this memory is organised as a doubly linked list this deletion is performed in time $O(1)$. In step four, the join node follows the causal links of the removed token in order to construct the token that must be propagated to its children. This is where the optimisation takes place, in the normal RETE algorithm the join node performs a consistency check of the token `<- LX>` with the entire right memory.

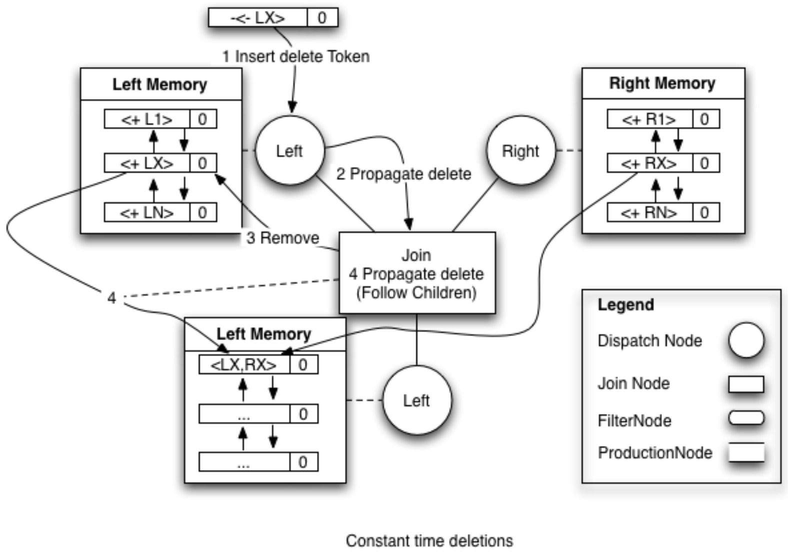


Figure 8.2: Constant time retractions

8.1.2 Reinserts

Volatile connections are inherent to an ambient environment, as Crime is designed for environments where connections can not be assumed stable, retractions and reinserts of facts are a recurring phenomena. While the previous section extended the basic Rete algorithm to handle retractions efficiently, here we discuss how the algorithm from the previous section can be improved to also perform reinserts of facts efficiently. This optimisation is known as *scaffolding* [47], which optimises both the retractions and deletions by incorporating a justification-based truth maintenance system – as described in chapter 3 – into Rete.

The aim of the optimisation is to decrease the cost of the reinsert operations by circumventing costly match operations in the join nodes of the Rete network. This is achieved by moving tokens to a special memory, dubbed *deactivated memory*, when they are retracted instead of deleting them. Whenever a retracted token is reinserted, the token can be retrieved from the deactivated memory and all the affected tokens can be recursively *reactivated*. Care must be taken to ensure that when a token is reactivated, it is still matched with all tokens inserted in the opposite memory after the token was deactivated. To determine which tokens meet that requirement, a unique *timestamp* is associated with every token. These timestamps should not be confused with those introduced for the temporal operators.

Listing 8.1 shows pseudo code for the retract operation performed by a token. Every token has a counter named *active* which equals zero when the token is active and negative whenever it is inactive. When a token is retracted, its active counter is decreased. Whenever the token was active – counter equal to zero – before the retraction, the token removes itself from the active memory. Subsequently the token updates its timestamp and the node in which the token resides places it in the deactivated memory (line 5). Thereafter, the derived tokens are informed of the retraction of their parent token by invoking the `retract` method (line 6-7). When the tuple was already deactivated before its retraction, no further operations must be performed.

```
 1  public void retract() {
 2          if(this.active-- == 0) {
 3                  this.removeFromMemory();
 4                  this.setTimestamp();
 5                  node.setToInactive(this);
 6                  for(child in this.children) {
 7                          child.retract();
 8                  }
 9          }
10  }
```

Listing 8.1: Inactive method

The propagation of the token $< -LX >$ is shown in figure 8.3. The numbers indicated in the second field of the token are the active counters. The tokens with a dashed line are affected by the retraction, one is in the left memory above the join node. This token $< +LX > |0$ is moved to the deactivate memory and its counter is decreased. It subsequently invokes the retract method of its child $< +LX, RX > |0$, resulting in the movement of its child to its deactivated memory. Because the combining the removed token $< +LX >$ with $< +RX >$ has already been performed previously and has been CACHED by making use of causal links, the join node can be completely bypassed, as the functionality of join node consists of the combining its left and right memory.

Re-insert operations can be optimised as tokens which were retracted previously are not thrown away and causal links are installed. Pseudo code for the `activate` method that implements the reinsert operation is shown in listing 8.2. Instead of inserting the token again in the RETE network, it reactivates itself by increasing its active counter. First of all, the reinserted token verifies whether all the supported tokens – tokens residing in one of the join node's memories, so a subpart of the reinserted token – are active. This can be realised by checking if its active counter equals zero. When this test succeeds, the token removes itself from the deactivated memory (line 3). Subsequently, the join node is informed of this reactivation and performs match operations with the newly added tokens from the opposite memory (line 4). These match operations are still necessary in this optimisation, because tokens can be added in the other join node's memory while the reinserted token was inactive. One of these newly inserted tokens can contribute to a token that must be propagated to the join node's children. After performing the variable consistency checks with those newly inserted tokens, the timestamp of the reinserted token is updated (line 5). Subsequently, the children of this token are informed of this reactivation (line 6-7).

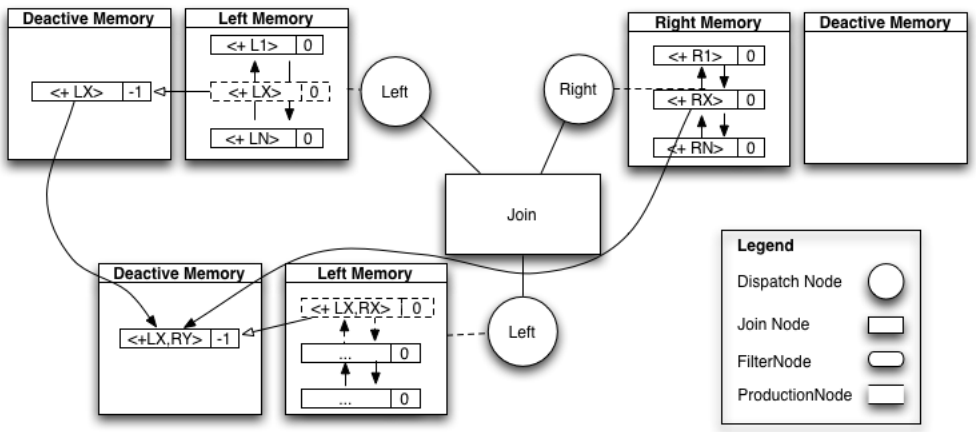


Figure 8.3: Constant time retractions

```
public void activate() {
        if(++this.active == 0) {
                this.removeFromMemory();
                node.reactivate(this);
                this.setTimeStamp();
                for(child in this.children) {
                        child.activate();
                }
        }
}
```

Listing 8.2: Activate method

8.1.3 Negation

Negations can be scaffolded in much the same way as the normal conjunctions performed by join nodes, however the semantics is different. Left memory activations are propagated the same way as with normal scaffolding, as is explained in section 8.1.2. However, right activations are reversed: When a token is inserted or reinserted in a negated join node, it propagates a retract token – a token with a minus sign – to its children. When right activations are retracted, a positive token is propagated to the node's children causing activations.

8.2 Scaffolding Crime

The extra constructs *Bagof* and *Findall* can also be optimised by using a technique similar to the one used for scaffolding join nodes of the Rete network. Those extra statements that are provided by the Crime language, typically accumulate certain variables – which are part of the tokens passed by their parent

– in one single list. Although these nodes only receive input from one parent, a match operation is still needed to group those variables together. For the bagof statement this accumulation is performed according to the values of unbound variables, whereas for the findall statement this grouping depends solely on the bound variables.

8.2.1 Bagof and Findall

The accumulation of facts depending on their unbound variables is just the same as performing a match operation, as is done by the join node. In the implementation of bagof, we use the same methods for determining whether a fact belongs to the same group as for verifying the variable consistency checks that are performed in the join node of the RETE network. Hence, scaffolding a bagof statement results in optimising this match operation, as was also the case for the scaffolding process conducted on join nodes. Instead of performing a match operation each time a token is inserted or retracted, we can keep a forward reference from that token to the datastructure keeping the accumulated list to which the token belongs.

We explain this optimisation by making use of the rule shown in listing 8.3. The rule collects all persons in a room by making use of the bagof operation. In this example there is only one unbound variable, namely `?room`. Recall that the bagof statement accumulates according to the unbound variables of the statement, hence the `bagof` on line 2 in the code excerpt accumulates all persons residing in the same room.

```
1  GroupLocations(?persons,?room) :-
2          bagof(?person,
3                  ( personInfo(?person,?id),
4            location(?id,?room). )
5                  ?persons).
```

Listing 8.3: Activate method

Consider the case where there are four facts inserted which indicate that Alice and Bob are in the kitchen. When Alice leaves the kitchen this results in the retraction of the fact `location(75773, Kitchen)`. Normally, this would cause the bagof node to match the variable binding `Kitchen` with all of the groups that have been made previously, in this example also `LivingRoom` as can be seen in the `Bagof Memory` shown in figure 8.4. Because the extended scaffolding algorithm keeps forward references from the token `< - location(75773, Kitchen) >` to the group `Kitchen`, this accumulated list is informed of the retraction. It subsequently sends out a remove of the previous values, and deletes Alice from the accumulated list followed by an insertion of a positive token where Alice is left out of the accumulated group. The various steps are shown in figure 8.4.

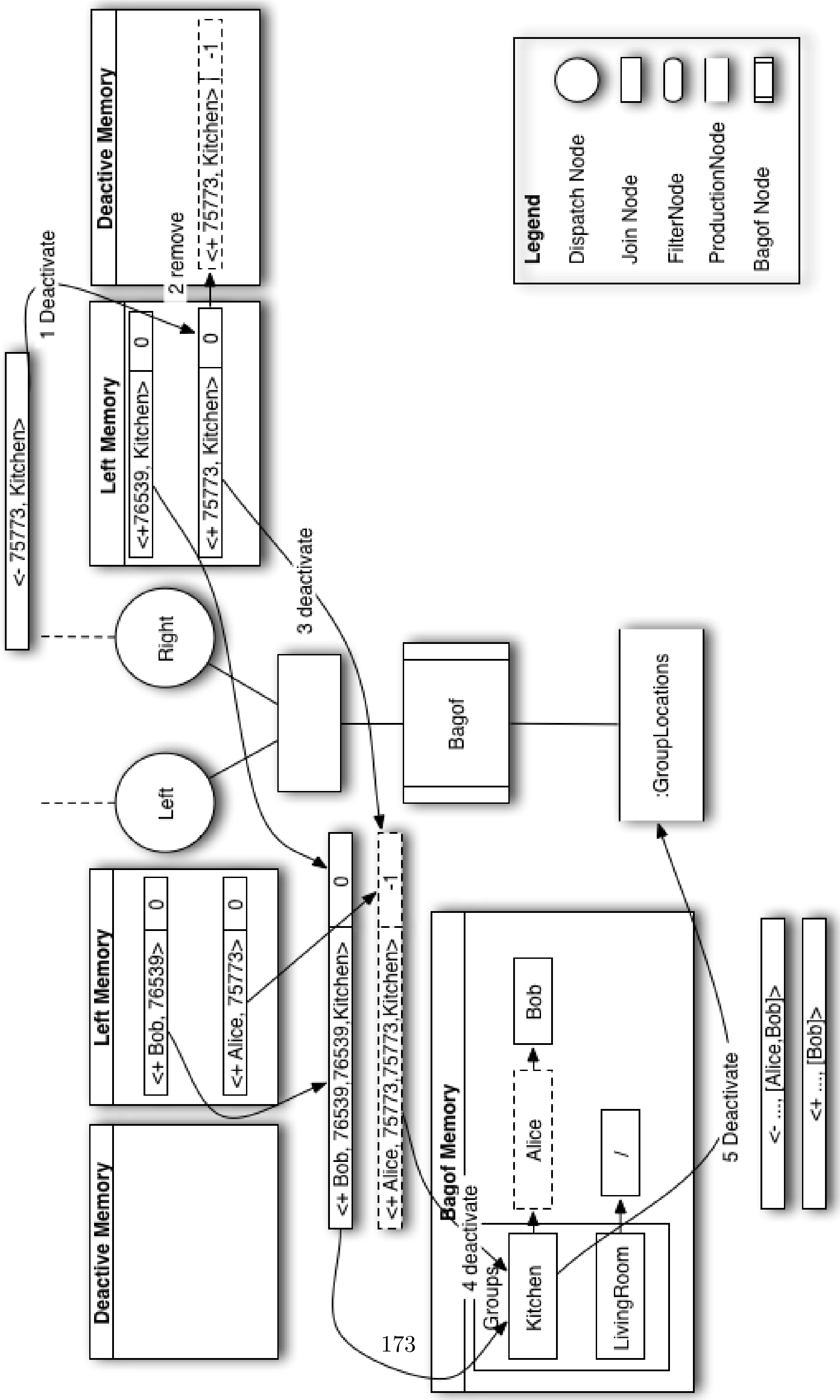


Figure 8.4: Optimisation of delete operation in bagof node

8.3 Optimise Time

Chapter 7 introduced Crime Time as an extension of Crime enabling reasoning about the past by introducing temporal operators. These operators that were presented in more detail in the previous chapter, each have a strong resemblance to the join node, as they can be considered as join nodes that perform extra timestamp-related tests. Hence, beside the match operations the corresponding nodes of the network also check time constraints.

8.3.1 Sometime-past

Recall that the *sometime-past* node, explained in chapter 7, is implemented as a join node performing an extra time-related test. Whenever a token is inserted in the sometime-past node, the node first performs a match operation as is also performed by the join nodes of the Rete network. Subsequently, the node verifies whether the left matched tokens have a timestamp strictly greater than the timestamp of the corresponding right token.

Optimising a sometime-past node can be realised as these timestamp-related tests can omit unnecessary match operations. Consider both memories of a sometime-past node that are shown in listing 8.1. The timestamps of the tokens are displayed at their last argument.

Whenever a new token is added, its timestamp is at least equal to the greatest timestamp so far, which is three in this example. Recall that timestamps of tokens are initialised and increased whenever a new activation is executed by the Crime engine. So, it could be possible that the insertion of `location(76539, LivingRoom)` and the addition of the fact `personInfo(Bob, 76539)` are part of the same activation, and hence the timestamps of their corresponding tokens are equal. Otherwise, the timestamp of the inserted token is strict greater, as in this example it equals four.

Left memory	**Right memory**
$< +personInfo(Alice, 75773); 2 >$	$< +location(75773, Kitchen); 1 >$
$< +personInfo(Bob, 76539); 3 >$	

Table 8.1: Left and right memory of a sometime-past node

The insertion of the token with timestamp four causes a right-activation, but no matching operations should be performed by the sometime-past node. As the time-constraint specifies that the time of the left fact should be strictly greater than the timestamp of the inserted token, this constraint can never be fulfilled. So, whenever a right-activation is executed, a sometime-past node can omit any operation, because the time-related constraint is always violated.

However, in case of a left-activation the sometime-past node must perform match operations as the right-memory possibly contains tokens with a timestamp strict smaller and whose variables may be consistent. These left-activations can again be scaffolded as seen in the previous sections.

8.3.2 Most-recent

This operator has similar semantics as sometime-past with the explicit restriction that only one matching fact can be returned. In other words, a rule body of the form `f1(), most-recent f2()` only matches a single fact `f2` which occurred before a matching fact `f1`. As the *most-recent* node, which is discussed in the previous chapter, resembles the sometime-past node, the same optimisations can be performed. The matching operations performed by the node can be omitted in case of right-activations, as is explained in section 8.3.1.

8.3.3 Since

The *since* operator takes two explicit arguments (respectively timestamped with `t2` and `t3`) and matches facts such that `t1` > `t3` > `t2`. In other words, a rule body of the form `f1(), since (f2(), f3())`, matches events `f3` which occurred between `f2` and `f1`. Here again the insertions of a fact `f3` can never result in a match because of the use of a monotonous time model.

8.4 Results

Here we present the result of the optimisation of the scaffolding technique presented in the previous sections. The stress test we performed consists of the rule shown in listing 8.4. The Rete network of this rule consists of eight join nodes and the match operation in every join node must check three variables in the worst case. The test begins by inserting the facts `a(1,2,3)` up to `i(1,2,3)` resulting in the triggering of the rule. The fact `a(1,2,3)` is removed an reinserted every iteration. In every iteration a linear increasing amount of garbage facts are inserted, these garbage facts are of the form `b(x,y,z)` up until `i(x,y,z)`. Because these garbage facts lack a proper fact `a(x,y,z)` no further matches are generated. However the use of these garbage facts simulates that the user is walking around and picks up information form various sources, however these sources do not contribute to an interesting event.

```
output(?x,?y,?z) :- a(?x,?y,?z),
                     b(?x,?y,?z),
                        ...,
                        i(?x,?y,?z).
```

Listing 8.4: Optimal Crime test rule

As can be seen in the figure 8.5 when increasing the "garbage" facts, the time needed to check all these facts by the standard Rete algorithm is increased quadratically. The `x` axis in the figure indicates the number of iterations while the `y` axis indicates the time in ms. The scaffolding algorithm on the other hand does not suffer from these "garbage" facts as it only needs to perform these variable consistency checks only once. The optimisation behaves linear to the amount of garbage that is inserted into the Rete network. Taking into account that the test is performed on a workstation and not on a mobile phone the difference in performance is significant. In order to get a realistic outcome of the stress test, the test has been performed 1000 times. The results are shown in figure 8.5.

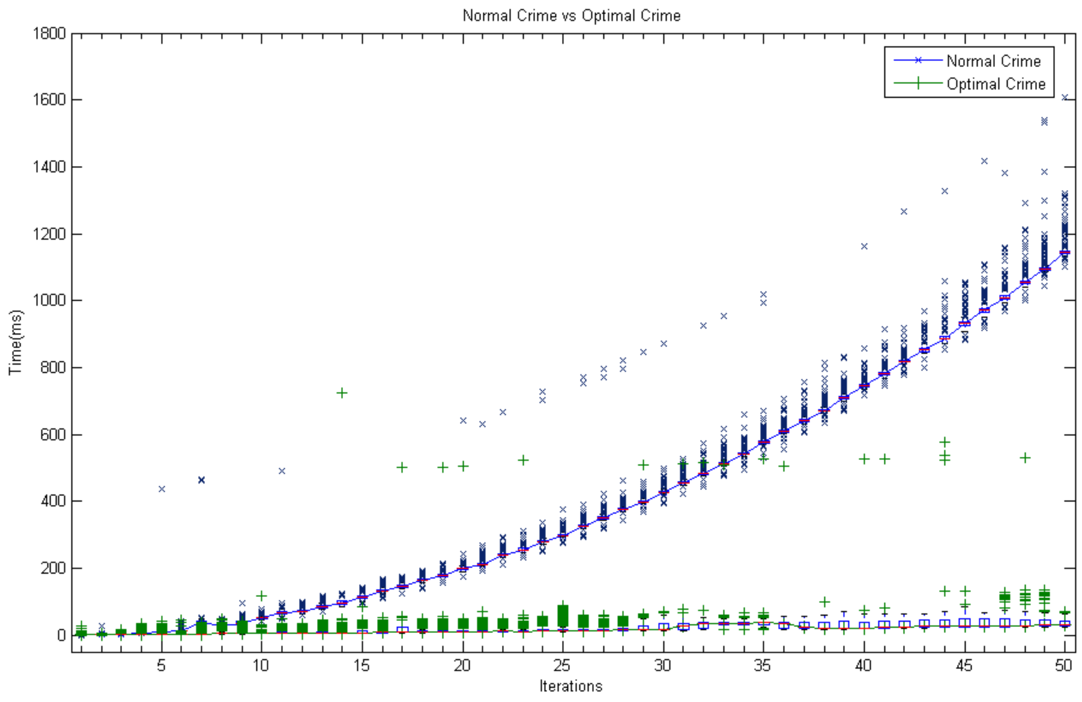


Figure 8.5: Stress test executed by optimised Crime

8.5 Future work

8.5.1 Network Optimisation

Volatile connections are inherent to an ambient environment, therefore in Crime retractions and reinserts happen frequently. In this chapter we presented an optimisation which decreases the computational processor power needed for the derivation engine of the Crime language. Here we discuss an optimisation technique that uses the cashing in deactivated memories to minimise the exchange of facts between mobile devices.

The idea of this optimisation is to tag the tokens with a monotone increasing identifier. These ids can then be used in order to reactivate facts from the working memory. Because the size of the ids is smaller than the actual facts, the total amount of data needed to be sent over the network is decreased. The implementation of this optimisation is mainly located in the exchanges of facts on a reconnect. We explain one possible protocol to exchange facts on a connection by means of figure 8.6.

When two client connect with each other, they first exchange the highest fact id received from the connecting client (step one and two on the figure). After receiving this id, the clients react by sending the ids of their active facts up until the id received in the previous step. The clients reactivate the received ids, which is step 5 in the figure. To finalise, the clients send their active facts higher than the id received in step one and two.

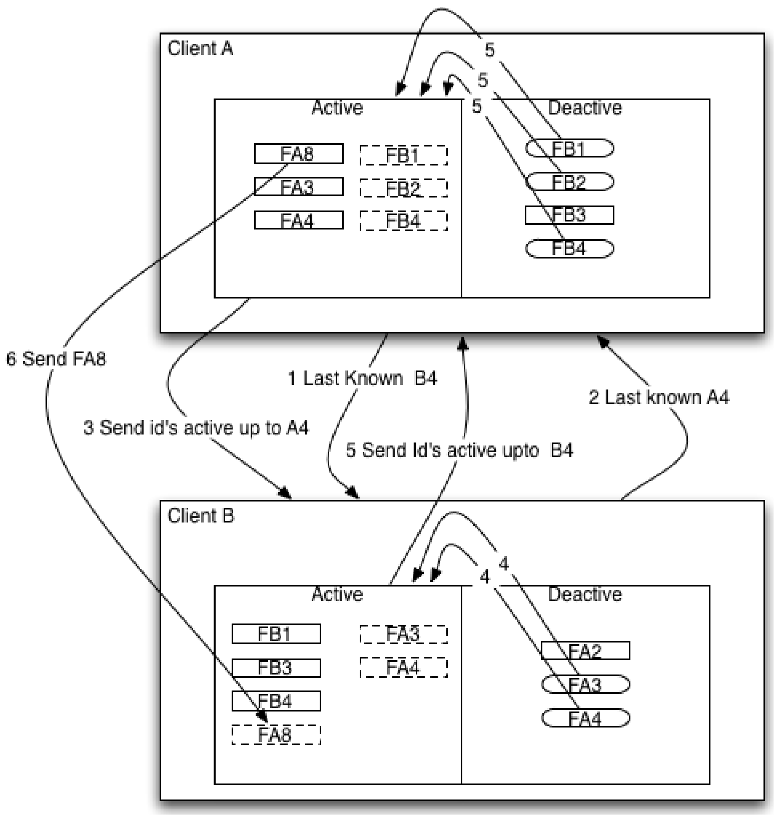


Figure 8.6: Network optimisation

8.5.2 Garbage Collection

As most mobile devices do not have much memory to spare, it's important to use the available memory wisely. One aspect to keep in mind is to free memory when it's not needed anymore, this can be implemented manually by the programmer or automated by the use of a garbage collector. Garbage collecting is the process to identify and reclaim memory allocated by an application which does not influence the output of the application by its removal. Tracing garbage collectors operate by first identifying all objects accessible by the application and then reclaim the memory allocated by all other objects. As these object are not accessible for the application, their removal does not influence the application at all. JAVA incorporates a sophisticated kind of tracing garbage collector, and therefore in standard CRIME tokens removed from the RETE network are reclaimed by the garbage collector as they are no longer accessible by the application itself. However, by the introduction of a deactivated memory for the optimisations shown in the previous sections, retracted tokens can not be reclaimed as they are still accessible by the application (via the deactivated memory). This is acceptable as most of these tokens are used to boost the performance of the application. However in certain cases, tokens residing in the deactivated memory do not have the potential to be reactivated. As they have the inability to contribute to the triggering of a rule in the system, their removal does not influence the reasoning process of the CRIME application and thus these tokens are qualified as garbage. The same holds for certain tokens kept into the nodes implementing the temporal operators.

HALO is a temporal logic pointcut language which like CRIME uses the RETE network for its derivation engine. HALO provides built-in garbage collection for its temporal operators and hence this mechanism could be adapted in order to identify the garbage produced by CRIME. However as HALO does not incorporate a deactivated memory, much garbage produced by these temporal operators in CRIME would not be recognised.

In the rest of this section we present the relevant parts of this garbage collection mechanism of HALO applied to the rules of CRIME and show how this algorithm should be adjusted to take into account the deactivated memories. For a full dissertation of the garbage collector of HALO we refer the interested reader to [30].

One difference with the HALO garbage collector is that in HALO facts themselves can be garbage collected, whereas in CRIME only tokens residing in the RETE network can be garbage collected. The reason for this is that facts in CRIME are exchanged with other entities in the network and thus can be relevant for others. This is in contrast with HALO where facts expressing join points are only relevant for the application at hand. Consider the rule shown in listing 8.5: this rule produces `lastPlayed` facts representing the *most recent* song played just before Alice walks into the room.

```
lastPlayed(Alice,?song) :-
        login(Alice),
        mostRecent(played(?song,Alice).).
```

Listing 8.5: Garbage collection rule

Assuming that the facts shown in listing 8.6 are subsequently inserted into the knowledge base of CRIME, we can then conclude that both `tokens` inserted at time one and two may be garbage collected. Token two is trivial as the `lastPlayed` rule does not have a prerequisite matching this fact and thus can never be triggered by such a token. Note that the fact can not be removed as it may trigger rules residing on other entities in the approximation. Token one is also qualified as garbage as the existence of a *more recent* fact `played(''songB'',Alice)` prevents it to contribute to the activation of the `lastPlayed` rule.

In HALO the garbage collection procedure stops with the removal of this fact. However, in CRIME all tokens derived from fact one should also be deleted as their parent can never be activated again. This could be incorporated into CRIME by a special *destroy* operation in the tokens which acts similar to the activate method shown in listing 8.2. However, instead of reinserting the token, it is just dismissed. By removing all references to these tokens, they are eventually garbage collected by the underlying garbage collector of JAVA.

```
played(''songA'',Alice).
played(''song1'',Bob).
login(Alice).
played(''songB'',Alice).
login(Alice).
```

Listing 8.6: Garbage collection rule

8.6 Conclusion

Volatile connections are inherent to an ambient environment, therefore in CRIME retractions and reinserts happen frequently. In this chapter we presented an optimisation inspired by this phenomena which tackles the most costly operation of the derivation engine of CRIME. This optimisation, known as scaffolding, is extended to be used with the specific operators of CRIME and its extension CRIME TIME. Small scaled test have shown that this optimisation performs significantly better than the normal RETE algorithm. Where the normal RETE algorithm slows down when used extensively, the optimised version does not. Future work includes the incorporation of a garbage collector for the temporal operators and minimising the network load.

Chapter 9

Conclusion

The focus of this dissertation was to provide mobile applications with a consistent view on their environment by giving them fine-grained control over the effects of disconnections. The Fact Space Model we have developed, is a model finding its roots in the Tuple Space Model but differs from it in two ways: First of all, the Fact Space Model is conceived as a distributed fact space were both the assertion of facts and their retraction are considered as relevant events. Secondly, our proposed model provides the programmer with a logic language to describe the mapping of context information to the actions to be performed.

To provide a model which is able to cope with the characteristics of a mobile environment, the Fact Space Model combines a set of features distinguishing itself from existing work on coordination languages, distributed reasoning engines and context aware middleware. First of all, context information is shared by making use of a federated fact space rather than by a publish-subscribe mechanism. This enables the immediate detection of information loss and subsequently the ability to reaction upon this disappearance. As information deletion is a direct translation of a disconnection between mobile devices, applications using the Fact Space Model for its coordination benefit from the ability to compensate the effects of these disconnections. Furthermore, the Fact Space Model offers a fully reactive logic language based on forward chaining. This language offers users with the benefits of reactive tuple spaces but extends this existing model by allowing the reaction upon a multitude of events and the ability to exploit the provided reversing mechanism. A feature which is, to the best of our knowledge, currently not provided by any other coordination language for mobile ad hoc networks. Finally the Fact Space Model differentiates itself from existing distributed reasoning engines in that it does not rely on a centralised architecture for its communication. We not only presented this model, but we also provided a proof-of-concept implementation, dubbed Crime.

9.1 Contributions

This dissertation advocates the use of a distributed reasoning engine in combination with a truth maintenance system in order to ease the development of context-aware applications.

We first give an overview of the important contributions:

- the Fact Space Model, a novel architecture for developing ambient and context-aware applications, specifically tailored for dealing with frequent disconnections
- proof-of-concept implementation of the fact space model, dubbed CRIME, which is a logical coordination language
- CRIME TIME, an extension of CRIME, incorporating language support under the form of temporal operators for reasoning about past context
- optimisation of the Crime query engine, with regards to its deployment in pervasive environments

We next discuss these contributions in more detail with regards to CRIME's deployment in pervasive environments.

CRIME is based upon the federated tuple spaces provided by LIME. These tuple spaces provide an imperative mechanism to share information in a decentralised environment. Moreover as federated tuple spaces are based upon the LINDA tuple spaces, they also support both space and time uncoupling, as we discussed in chapter 2. In order to react upon a certain tuple entering the tuple space, LIME provides the notion of events, however these events are unable to react on a combination of tuples entering the tuple space. Therefore, the tuple space model provided by LIME suffers from the inability to react on a combination of contextual changes. Furthermore, the language doesn't support a reversing mechanism to undo the reactions on tuples which are no longer present in the tuple space.

CRIME is an immediate instantiation of the Fact Space Model, which overcomes these shortcomings as it uses a logical inference engine for reasoning about a combinations of facts residing in the fact space. The use of a logical inference engine is also supported by other languages like for example GAIA which is described in chapter 4. As for GAIA, the rules of this logic language allow to keep track of causal links between the current context and possible reactions on that context information. However, there's no support for redoing the actions whenever the context is changed. Reacting and reversing the actions based on the sensed context is possible in CRIME as the retraction of a fact in the fact space can cause the prerequisites of a rule to fail and the consequences of that rule are reversed, as specified.

The extension of CRIME with temporal operators adopted from HALO, is a track that has not been followed by other context-aware frameworks. Most of these frameworks provide a context history which can be queried to extract past information, although this results in similar semantics, the use of dedicated operators significantly improves the expressiveness of the language. Moreover the integration of these declarative temporal operators provide additional benefits for mobile devices as the used past information that has to be kept is minimised to that information used by the temporal operators.

The RETE algorithm on which the CRIME query engine is built, has been optimised for the use in a pervasive environment. In Crime, context sharing is based upon connectivity: in a mobile environment, where intermittent disconnections occur frequently, inserting and removing context should be done *fast*. The most costly operation of the RETE algorithm resides in the matching phase performed by the join nodes. By incorporating of a justification-based truth maintenance system into the RETE network of CRIME, a large part of this matching phase is omitted. Obviously, the specific optimisation of the RETE network to deal with the characteristics of a mobile environment, has not been done for other context-aware frameworks.

9.2 Future Work

This section gives an overview of possible extensions for the Fact Space Model and CRIME. These extensions stem from various research area which were not investigated into depth but are worth exploring to produce a coordination language usable for the production of industrial applications.

Visual Programming

The work conducted in this dissertation could be used build a visual programming language, where the rules in the system are drawn. A large component library should be available consisting of actions and widgets. This component library should provide the most commonly needed actions used for mobile devices. In this visual programming language, the user should only be concerned about the adaptation of the application at hand. A good starting point to investigate this, could be the adaptations of Mac applications as these provide a rich set of hooks through their applescript support.

Optimisation

One path of future research is to focus on the usability of CRIME for small devices by further optimisation of the language. On one hand, we should aim to decrease the network load and on the other hand decrease the memory consumption and processing power needed. As the processing cycles of mobile devices are always reflected in the power consumption of these devices, it is worth the effort to optimise as much as possible. For minimising the network load we have already proposed the use of a more intelligent sharing mechanism inspired by the optimisations shown in chapter 8. Decreasing the memory consumption could be realised by the incorporation of a garbage collector similar to the one used in HALO [30].

Smarter Behaviour

Another path to follow is the extension of the existing model with machine learning techniques in order to produce applications which learn their wanted behaviour themselves. Not only could this open the doors for more intelligent systems but also more adaptable systems which in time could take into account new information which was unavailable at their production time. A technique worth investigating in order to derive new rules from a set of examples is known is called *inductive logic programming* and is described in the book Simply Logical [23]. The catch when opting for this technique is to find a suitable way to collect the examples.

Implementation

By gaining experience in the development of context-aware application, we have already detected the need to support temporal operators. As extensions to the language itself are endless, we limit ourselves to only the most relevant and appealing features in mind.

The current implementation of CRIME does not support the retraction of rules, also when rules are added locally they only take into account the facts perceived after their addition. The aim of the dynamic addition and removal of rules is to provide more flexible and extensible applications. Extending the inference engine and network layer of CRIME in order to allow the addition and removal of rules both locally and remotely is thus worth investigating

Exception handling is an important research area of distributed programming languages, therefore future research should concentrate on the incorporation of an exception handling system into CRIME in order to deal with runtime errors. A premature idea is the use of specialised error facts which in their turn could trigger specialised error handlers to deal with the error at hand.

The current implementation of CRIME provides the user with multiple tuple spaces as a basic abstraction in order to group related facts. As this method allows CRIME application to shield certain facts from eachother, there is no mechanism that prevents a malicious user to read the tuples residing in the tuple space. Therefore, applications written in the current implementation of CRIME are not suited for security critical applications. Concretely, this future work would concentrate on providing an abstraction layer to secure the use of the fact space by making use of encryption.

Bibliography

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs - 2nd Edition (MIT Electrical Engineering and Computer Science)*. The MIT Press, September 1996.

[2] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, pages 304–307, London, UK, September 1999. Springer-Verlag.

[3] Owen Holland And. Stigmergy, self-organisation, and sorting in collective robotics. *Artificial Life*, pages 173–202, 1999.

[4] Jakob E. Bardram. *The Java Context Awareness Framework (JCAF) A Service Infrastructure and Programming Framework for Context-Aware Applications*. Springer, 2005.

[5] Peter Barron and Vinny Cahill. Using stigmergy to co-ordinate pervasive computing environments. *wmcsa*, 00:62–71, 2004.

[6] Peter Barron and Vinny Cahill. Using stigmergy to co-ordinate pervasive computing environments. In *WMCSA '04: Proceedings of the Sixth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'04)*, pages 62–71, Washington, DC, USA, 2004. IEEE Computer Society.

[7] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (uri): Generic syntax. Rfc 3986, W3C, January 2005. http://www.gbiv.com/protocols/uri/rfc/rfc3986.html.

[8] Tom Slrensen Brian Nielsen. *Distributed Programming with Multiple Tuple Space Linda*. PhD thesis, Aalborg, Denmark, 1994.

[9] P. J. Brown, J. D. Bovey, and Xian Chen. Context-aware applications: from the laboratory to the marketplace. *Personal Communications, IEEE [see also IEEE Wireless Communications]*, 4(5):58–64, 1997.

[10] S. Brueckner and H. Parunak. Swarming agents for distributed pattern detection and classification. AAMAS 2002, 2002.

[11] Christoph Brzoska. Temporal logic programming with metric and past operators. In *IJCAI '93: Proceedings of the Workshop on Executable Modal and Temporal Logics*, pages 21–39, London, UK, 1995. Springer-Verlag.

[12] Federico Cabitza and Bernardo Dal Seno. Djess - a knowledge-sharing middleware to deploy distributed inference systems. In *WEC (2)*, pages 66–69, 2005.

[13] Guanling Chen and David Kotz. A survey of context-aware mobile computing research. Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, November 2000.

[14] S. Corson and J. Macker. Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations. RFC 2501 (Informational), January 1999.

[15] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. *Lecture Notes in Computer Science*, 1995:18–??, 2001.

[16] Philippe David and Thomas Ledoux. Wildcat: a generic framework for context-aware applications. In *Proceeding of MPAC'05, the 3rd International Workshop on Middleware for Pervasive and Ad-Hoc Computing*, Grenoble, France, 2005. ACM Press.

[17] Anind K. Dey, Gregory D. Abowd, and Andrew Wood. Cyberdesk: a framework for providing self-integrating context-aware services. In *IUI '98: Proceedings of the 3rd international conference on Intelligent user interfaces*, pages 47–54, New York, NY, USA, 1998. ACM Press.

[18] Anind K. Dey, Daniel Salber, and Gregory D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16, 2001.

[19] Anind K. Dey, Daniel Salber, and Gregory D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16, 2001.

[20] E. W. Dijkstra. The structure of the multiprogramming system. *CACM*, 11(5):341–346, May 1968.

[21] Robert B. Doorenbos. Production matching for large learning systems. Technical report, Pittsburgh, PA, USA, 2001.

[22] Jon Doyle. Truth maintenance systems for problem solving. In *IJCAI*, page 247, 1977.

[23] Peter Flach. *Simply logical: intelligent reasoning by example*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[24] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. In J. Mylopoulos and M. L. Brodie, editors, *Artificial Intelligence & Databases*, pages 547–557. Kaufmann Publishers, INC., San Mateo, CA, 1989.

[25] Ernest Friedman-Hill. *Jess in Action : Java Rule-Based Systems (In Action series)*. Manning Publications, December 2002.

[26] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[27] Joseph C. Giarratano and Gary Riley. *Expert Systems: Principles and Programming*. PWS Publishing Co., Boston, MA, USA, 1994.

[28] Brendan Gowing and Vinny Cahill. Meta-object protocols for c++: The iguana approach. Technical report, 1996.

[29] P.-P. Grass. La reconstruction du nid et les coordinations inter-individuelles chez bellicositermes natalensis et cubitermes sp. la thorie de la stigmergie: Essai d'interprtation du comportement des termites constructeurs. *In: Insectes Sociaux 6*, pages 41–83, 1959.

[30] Charlotte Herzeel. A temporal logic language for context-dependence in crosscuts. Master's thesis, Brussel, Belgium, 2006.

[31] Charlotte Herzeel, Kris Gybels, Pascal Costanza, and Theo D'Hondt. Modularizing crosscuts in an e-commerce application in lisp using halo. ILC 2007, 2007.

[32] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.

[33] M. Elizabeth C. Hull. Occam - a programming language for multiprocessor systems. *Comput. Lang.*, 12(1):27–37, 1987.

[34] IST Advisory Group (ISTAG). Ambient intelligence: from vision to reality. IOS Press, 2005.

[35] Dirk Kalp, Milind Tambe, Anoop Gupta, Charles Forgy, Allen Newell, Anurag Acharya, Brian Milnes, and Kathy Swedlow. Parallel ops5 user's manual, 1988.

[36] I. Kassabalidis, M.A. El-Sharkawi, R.J. Marks II, P. Arabshahi, and A.A. Gray. Swarm intelligence for routing in communication networks. *Global Telecommunications Conference, 2001. GLOBECOM '01. IEEE*, November 2001.

[37] John Keeney. Chisel: A policy-driven, context-aware, dynamic adaptation framework. *Proceedings of the Fourth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, pages 3–14, 2003.

[38] Brian W. Kernighan and Dennis M. Ritchie. *C Programming Language*. Prentice Hall, March 1988.

[39] Ohbyung Kwon, Keedong Yoo, and Euiho Suh. ubies: An intelligent expert system for proactive services deploying ubiquitous computing technologies. In *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 3*, page 85.2, Washington, DC, USA, 2005. IEEE Computer Society.

[40] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[41] Marco Mamei and Franco Zambonelli. Self-maintained distributed tuples for field-based coordination in dynamic networks. *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 479–486, 2004.

[42] Ronaldo Menezes and Robert Tolksdorf. Adaptiveness in linda-based coordination models. *Proc. of the 1st InternationalWorkshop on Engineering Self-Organising Applications*, July 2003.

[43] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), April 1965.

[44] A. Murphy, G. Picco, and G.-C. Roman. Lime: A middleware for physical and logical mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, pages 524–536. IEEE Computer Society, 2001.

[45] Lionel M. Ni, Yunhao Liu, Yiu Cho Lau, and Abhishek P. Patil. Landmarc: indoor location sensing using active rfid. *Wirel. Netw.*, 10(6):701–710, 2004.

[46] Scientific Computing Associates Incorporated. Paradise. User's guide and reference manual. Technical report, 1994.

[47] Mark Perlin. Scaffolding the RETE network. In *Proceedings of the International Conference on Tools for Artificial Intelligence*, pages 378–385. IEEE Computer Society, 1990.

[48] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. LIME: Linda meets mobility. In *International Conference on Software Engineering*, pages 368–377, 1999.

[49] Anand Ranganathan and Roy H. Campbell. An infrastructure for context-awareness based on first order logic. *Personal Ubiquitous Comput.*, 7(6):353–364, 2003.

[50] Manuel Romn, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, October 2002.

[51] A. Rowstron and A. Wood. Bonita: a set of tuple space primitives for distributed coordination. In *Proc. HICSS30, Sw Track*, pages 379–388, Hawaii, 1997. IEEE Computer Society Press.

[52] Antony I. T. Rowstron and Alan Wood. Solving the linda multiple rd problem using the copy-collect primitive. *Science of Computer Programming*, 31(2-3):335–358, 1998.

[53] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*, chapter Planning, pages 375–416. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[54] N. S. Ryan, J. Pascoe, and D. R. Morse. Enhanced reality fieldwork: the context-aware archaeological assistant. In V. Gaffney, M. van Leusen, and S. Exxon, editors, *Computer Applications in Archaeology 1997*, British Archaeological Reports, Oxford, October 1998. Tempus Reparatum.

[55] Schaefer. T. supporting meta-types in a compiled, reflective programming language. *PhD Thesis, Department of Computer Science,University of Dublin*, 2001.

[56] B. N. Schilit and M. M. Theimer. Disseminating active map information to mobile hosts. *Network, IEEE*, 8(5):22–32, 1994.

[57] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, 1994.

[58] Ellen H. Siegel and Eric C. Cooper. Implementing distributed Linda in standard ML. Technical Report CMU-CS-91-151, Pittsburgh, PA 15213, October 1991.

[59] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill Higher Education, 2001.

[60] P. ssimo, V. Cahill, A. Casimiro, K. Cheverst, A. Friday, and J. Kaiser. Cortex: Towards supporting autonomous and cooperating sentient entities. *In Proceedings of European Wireless 2002.*, 2002.

[61] Mladen Stanojevic, Sanja Vranes, and Dusan Velasevic. Using truth maintenance systems: A tutorial. *IEEE Expert: Intelligent Systems and Their Applications*, 9(6):46–56, 1994.

[62] Leon Sterling and Ehud Y. Shapiro. *The Art of Prolog - Advanced Programming Techniques, 2nd Ed.* MIT Press, 1994.

[63] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[64] Éric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-aware aspects. *Proceedings of the 5th International Symposium on Software Composition (SC 2006), at ETAPS 2006*, pages 227–242, mar 2006.

[65] R. Want, A. Hopper, V. Falcao, and J. J. Gibbons. The active badge location system. *ACM Transactions on Information Systems*, 10(1):91–102, 1992.

[66] M. Weiser. Ubiquitous computing. *Computer*, 26(10):71–72, 1993.

[67] Terry Winograd. Architectures for context. *Human-Computer Interaction, 16 (2)*, pages 401–419, 2001.

Index