A Real Time Expert System

Adapting Match Algorithms and Implementing a Tailored Rule Language

10th Semester Software Engineering Project, 2011

Yanik K. Challand

Department of Computer Science
Aalborg University

Selma Lagerlöfs Vej 300
DK-9220 Aalborg Øst
http://www.cs.aau.dk

Title

A Real Time Expert System — Adapting Match Algorithms and Implementing a Tailored Rule Language

Semester theme

Programming Technologies and Embedded Systems

Project term

SW10, 2011

Project group

d610a

Supervisor

Lone Leth Thomsen, Arne Skou

Abstract

This report describes the design and the implementation of a hard real time expert system. It studies the possibility to use expert system in safety critical environments.
The real time expert system is comprised of three elements. A time predictable match algorithm designed for the purpose of this project. A rule language that aims at making the expert system's limitations clear to the experts coding the rules. A system for firing the actions of the rules that are matched by the algorithm.
The system is developed for the SCJ2, a safety critical profile for Java. The tool SARTS is intended to provide analysis to the developers using the expert system. This project shows that hard real time expert system are possible, but not usable in a real development environment due to the scalability problems of the analysis tool chosen.

Participants

Yanik Kim Challand

Preface

The following master thesis is written by Yanik Kim Challand, a Software Engineer student at Aalborg University.

When the words “we” or “I” are used, it refers to the author of this document.

The code presented may differ from the actual code, it can be by mistake but usually is to focus on the interesting part of the implementation.

Knowledge of Java and object oriented language in general is assumed. General knowledge in the field of computer science of the level of a computer science bachelor is also assumed.

All the code referred to in this document and this document itself, can be found at `http://yanik.challand.ch/d610a/master-thesis.zip`.

Yanik Kim Challand

Contents

Chapter 1

Introduction

The nineteenth century saw the rise of artificial intelligences (AI) in literacy, with books like Hoffman's *The Sandman* or Mary Shelley's famous *Frankenstein*. Today, reality is still catching up with fiction with the help of modern computers. As soon as 1938, while developing his Z1 computer, Konrad Zuse recognized that the technology will eventually become an artificial brain. [McC04]

After world war 2, Mark 1, the first stored-program computer comes online at Manchester University. Turing and his colleagues attempt to program it to play chess. While the AI field has yet to adopt its current name, Turing investigate it further and in 1950, publishes "Computing Machinery and Intelligence" proposing the Turing Test. [McC04]

During the summer 1956, the Dartmouth Conference was hosted on the campus of Dartmouth College. It is viewed by many as the birth of the AI field and it saw the demonstration of the first working AI program, the Logic Theorist [McC04]. This is where John McCarthy proposed the term "artificial intelligence" as a name for the field [McC04] that he today defines in his paper [McC07] as:

> It is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable. [McC07]

He then goes on defining *intelligence*, while later acknowledging that there is no definition of intelligence that is not depending on relating it to human intelligence [McC07]. Because human intelligence is only partially understood [McC07], the expectations for the AI field have raised each time achievements were reached [McC04].

Together with the bold claims of AI researchers and government budget cuts, this resulted in two periods where funding for AI research was very low and as a consequence, where AI research output was dramatically reduced. These two periods were called the *AI Winters*. The first one lasted a few years after 1974, while the second AI winter began with the collapse of the Lisp Machine market in 1987 and lasted to the beginning of the 1990's [RN03]. [McC04]

Our interest lies between these two periods, with the rise of *expert systems*. The precursor of these systems was developed in the early 70's and was called Mycin. Expert systems were the first commercial success for the AI field, encouraging the

start of AI companies and knowledge engineering. The following sections introduces expert systems, why we are interested in them and what is the purpose of this project. [RN03, McC04]

1.1 Expert Systems

The commercial success of expert systems lies in the fact that they can solve specific known problems providing expert quality performance in that specific area. These systems are knowledge based: They use compiled knowledge from human domain experts in order to yield conclusions on their domain area. [Lug05]

The structure of an expert system is made of three main components: a knowledge base, a fact base and an inference engine. The relation between these three main components is shown in Figure 1.1 as well as their interaction with the different interfaces. [Lug05]

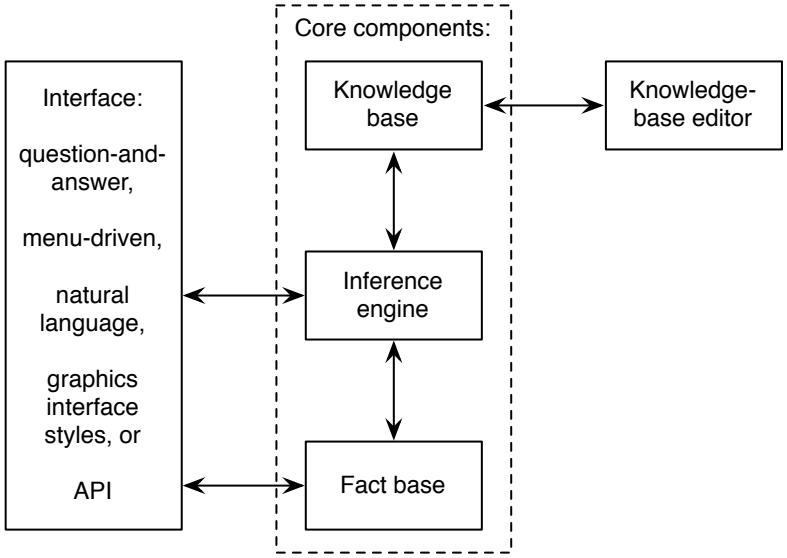

Figure 1.1: The structure of a typical expert system, showing its main components and their interaction between themselves and with the interfaces.

The knowledge base is the heart of an expert system. It contains the knowledge compiled in a set of rules. The inference engine then matches the facts contained in the fact base against the rules, yielding the actions of the rules matched. The actions change the facts and the process is repeated. In some expert systems, the inference engine asks users questions to improve or augment its fact base. [Lug05]

The rules are composed of two parts: the left hand side (LHS) and the right hand side (RHS). The LHS contains the conditions against which the fact are matched, and the RHS contains the actions that are taken when the conditions are matched [Lug05]. The following rule is an example from an expert system that would be used in an insurance company. This is a simple business rule:

> Car insurance subscribers that are older than 25 get a 10% discount.

A quick analyse of this rule show that the left side if the rule would require that a car insurance subscriber is older than 25 years old. If that was the case, the right hand side would be activated and here will allow for a 10% discount. One can also say that the rules have an *if-then* structure [Lug05].

Business rules are likely to change at a rapid pace, and expert systems allow for non-programmers to create rules via a knowledge-base editor. The knowledge base being separated from the code base enables rapid change of the rules by experts in specific business problems and allow for easier maintaining of the code base as well as saving expensive developer-time.

Another benefit of expert systems is speed. Checking every fact against every rule's conditions is time consuming and not the most efficient strategy. Inference engines use *match algorithms* to check the rule base against the fact base in an efficient way. The Rete algorithm has become the dominant match algorithm and today, evolution of it powers most of the current expert systems [CM10].

1.2 Real-Time Expert Systems

Expert systems can be used in many situations, one of which is in an insurance company where it helps identify relevant business rules, like discounts as we showed above. Some situations require real-time as well as intelligent decision-making. This is the case for a car or airplane autopilot, a network router detecting attacks and acting on them before it is too late, or a system that trades on stock markets and needs to buy and sell fast enough to gain money. These last examples are systems that need to make decisions in real-time.

Since the term "real-time" has been abused many time and does not have a clear definition, it is defined for this report in Section 2.1. For the purpose of this introduciton, know that there exists soft and hard real-time systems. We concentrate on hard real-time which defines systems that absolutely must respond within a specified deadline [BW09]. For the rest of this report, when "real-time" is used, hard real-time is meant.

The example systems presented in this section are all hard real-time, since a failure to react within a deadline can at least render them useless, and at most result in the loss of life. While autopilots are obvisously safety-critical systems, the others can be under certain circumstance also considered safety-critical.

According to [Kni02], a system qualifies as safety-critical when its failure could lead to consequences that are determined to be unacceptable. Clearly, the failure of an autopilot can lead to loss of life which is unacceptable. In the network router case, it depends on what network it protects. If it is the network of a car factory which when breached allows for malware to be loaded on new cars, the consequences can as well be unacceptable.

Big loss of money can as well be unacceptable for a company or an individual and this could both happen with a failure of the network router security system or if a stock market trade system fails. There are many systems that can be considered safety-critical and they require to be verified in order to ensure that they will not fail. In the case of hard real-time systems, this includes verifying that the system will meet the specified deadlines every time.

Safety-critical systems like the ones presented in this section could benefit from a hard real-time expert system to take decisions. However such an expert system must be verifiable to guarantee it will not fail. Gensym commercializes *G2*, a real-time expert system that is used among others by NASA for the monitoring of their spacecraft operations and Ericsson for detecting and solving problems in their networks [Gen09]. While Gensyms claim that *G2* is real-time, there is no mention of verifiability.

The inference engine of an expert system is composed of a match algorithm, as described further in Chapter 3. The match algorithm is responsible to identify the rules that are satisfied by the current facts and thus can be considered as the central part of an expert system. Unfortunately, the today widely used match algorithm Rete was shown in [Hal87] to not be time predictable and thus it is not suitable for hard real-time applications. The lack of documented match algorithm that are time predictable and thus verifiable, prevents the use of expert systems in safety-critical systems.

1.3 Problem Statement

Real-time expert systems are dependent on a match algorithm that is verifiable in order to guarantee the reliability of the system. However, the currently de facto match algorithm Rete is not time predictable and thus can not be used in safety-critical applications. Two newer match algorithms have been presented since Rete: *Treat* in [Mir87] and *Leaps* in [Bat94]. These algorithms have not been analysed for time predictability, supposedly because those two articles were released during the second AI Winter.

Treat and Leaps have some similarities with Rete and can be viewed as evolutions or optimisations. Those similarities could yield that they are as well not time predictable, but even in this case understanding them can inspire in the creation of a new algorithm. Together with Rete, Treat and Leaps are further presented in Chapter 3.

In addition to show that Rete has unpredictable response time, [Hal87] presents some restrictions that if applied to Rete would allow it to be time predictable. However there is concern that these restrictions compromise the power and flexibility of expert systems [LCS+88]. In the case that restrictions have to be applied for a real-time expert system to be viable for safety-critical applications, making them apparent to the expert writing the rules through the rule language is a strategy that could reduce their impact.

[LCS+88] states that research to make a hard real-time expert system should be a priority, and in addition of the commercial success of G2 this motivate the main goal of this project: To provide a prototype of a hard real-time expert system that can be verified with an existing verification tool. This prototype should have a rule language that facilitates the verification.

In order to build such a prototype, some partial goals need to be reached. They are listed below together with their restrictions and main requirements.

Some of these goals a purely learning goals, this being a university project. Here is a list of them:

- Design and build a time predictable match algorithm.

  - Using either Rete, Treat, Leaps or an new match algorithm inspired by them.
- Design and implement a rule language that suits the match algorithm and facilitates verification.
  - It should include support for stating rule's priority.
  - Support for RHS should not be included.
- Design and implement the hard real-time expert system itself, incorporating the match algorithm and the rule language.
  - It should provide a fact base updating function.
  - It should provide a function that check the rules' LHS and return which rules were matched.
  - It should support firing RHS according to matches.
- Show that verification with an existing tool is possible using an example case.

As can be seen in the list, the prototype does not support the execution of the rules RHS. This restriction allows to keep the project's size reasonable and focused on the problem of interest, which is the matching of rules against facts. It should be noted that the match algorithm is developed in priority with the possibility for the rules LHS to check equality. Depending on the time left, it will be followed by other prototypes that allow inequality check and then negativity in that order.

Chapter 2

Real-Time Systems

In Section 1.2 real-time systems were briefly introduced with some examples and definitions. This chapter goes more into the details and further defines what are real-time systems. Since the focus is on understanding real-time systems and their specific issues, we will not for this chapter talk about expert systems.

In Section 2.1 we define some important terms used in this report and present the basics of what a typical real-time system is. This is followed by a section about the specific issues encountered with real-time, such as time limits and making sure the program executes within those limits.

Section 2.3 presents different ways Java has been adapted to be used in real-time systems, both with implementations of a real-time specification and with dedicated processors. Finally the tools used to tackle the issues discussed in Section 2.2 are presented with Java in mind.

2.1 Definitions and Basics

There are many interpretation of the term "real-time system" and it often only refers to high performance systems. The literature helps gives different definitions. These are taken from [BW09]:

> A real-time system is any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified period.
>
> A real-time system is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment.

The definitions above both mention bounded time and external stimuli. [BW09] goes further and adds that *the correctness of a real-time system depends not only on the logical result of the computation, but also on the time at which the results are produced.*

Process control system like the in Figure 2.1 is an example of a typical real-time system. This is a flow control system that take input from the flow meter to then regulate the flow with the valve (output). It has to change the valve angle within a

finite period in order to avoid overloading the receiving equipment. Here we can see that if the valve angle is changed too late, it will result in a failure even if the angle was right.

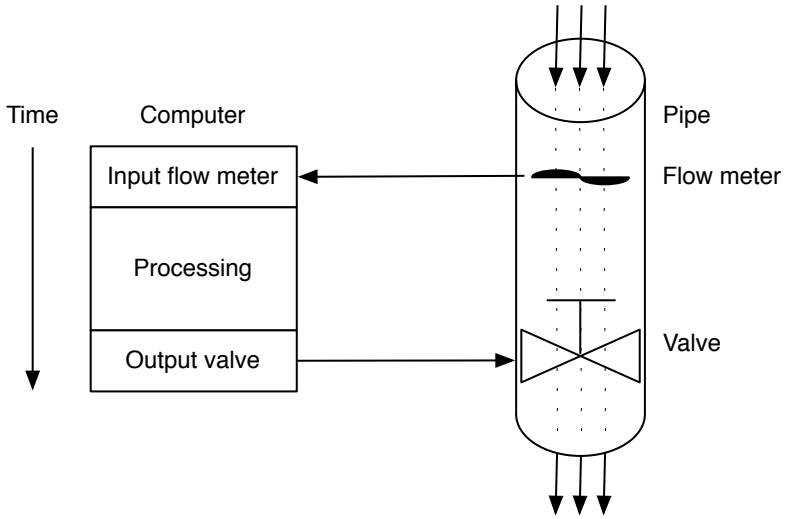

Figure 2.1: Example of real-time system. This is a flow control system (taken from [BW09]).

Now that we know more about real-time system, lets look at what soft and hard real-time means. Hard real-time is already introduced in Section 1.2 but it is repeated here with an example for clarification. As stated before, soft real-time system are not in the scope of this project, nevertheless defining them helps to grasp the nature of hard real-time systems and they are therefore both defined here.

In soft real-time systems missing the deadline that has been set does not result in a critical failure, instead the system degrade the quality of the service [BB09]. This is typically seen in a video player where each frame has to be shown within a deadline that could be 1 millisecond. Past that deadline the next frame should be shown and this one no longer does makes sense. It is just dropped thus reducing the quality of service.

Hard real-time systems fail when a deadline is missed. Usually they loose their purpose if they don't respond within the deadlines. The example above is typically hard real-time since failure to respond within the deadline could cause damages to the equipment.

A subset of hard real-time system is called safety-critical. As stated in Section 1.2 systems qualify as safety-critical when its failure could lead to unacceptable consequences. What is unacceptable can be variable, but anything that would yield loss of life or in our society big losses of money are unacceptable. Example range from a car ABS, a plane autopilot to the systems that power Wall Street's market server.

Another characteristic of real-time systems is that they are inherently concurrent. They have more than one task (usually threads) that need to be done within a deadline [BB09]. In the example above, the system needs to periodically read the flow meter and then adjust the angle of the valve if this is necessary. We can see here that there are two kinds of tasks, periodic and aperiodic.

Periodic tasks are released with a fix interval and like all real-time task it has a deadline. The deadline must be shorter or equal than the interval otherwise it would not have enough time to finish. It might also have an offset time which control when it is released after the initialisation of the program.

Aperiodic tasks can be released at any time, which is not suitable for hard real-time systems since they could be released another time before the first released has had enough time to finish. Hard real-time systems use a specialisation called sporadic tasks which have a minimum inter-arrival time, i.e. it can not be released more often than its minimum inter-arrival time. Aperiodic and sporadic tasks are released by events and have a deadline as well.

2.2 Specific Issues

Hard real-time systems introduce some issues that need to be controlled in order to ensure the system will meet its deadlines. These issues cannot be controlled if the system is not predictable, that is it does not behave in way that can be predicted mathematically [BB09].

Since every task has a deadline, there is a need to prove that every task can be complete within this deadline. In order to do so, the *worst case execution time (WCET)* of each tasks must be equal or lower than their respective deadlines.

But even if all the tasks are proven to take less time than their deadlines, there can still be an issue since they are threads that have to share resources. If deadlocks occur, one or more tasks cannot be completed and thus misses their deadlines. So the tasks must also be proven to be schedulable.

Memory is an issue as well. A system that keep on allocating memory will eventually use all the memory available to it and fail. Safety-critical system are forbidden to fail, therefore memory allocation should be controlled as well. In fact, some real-time programming languages forbid to allocate memory once the system has finished its initialisation phase. The usage of garbage collector is not possible for the moment since they are unpredicatble, but this is being the subject of much research at the moment [FPJ07, SV07] .

2.3 Real-Time Java

Java is limited in the context of real-time applications and the requirements that are necessary for real-time java were examined by the National Institute of Technology. The Real-Time Specification for Java (RTSJ) mostly succeed in fulfilling these requirements. Since RTSJ's target all real-time systems, it provides functionalities that are not suited for hard real-time systems. [GB]

Some Java profiles were implemented especially for hard-real time systems. This is the case for Ravenscar Java described in [JKK02] and Safety Critical Java (SCJ) described in [MSR07]. [TBO08a] present a further development of SCJ called SCJ2 which introduce some features for time calculation.

Ravenscar Java is a subset of RTSJ that improves in several key area related to hard real-time such as predictability of timing [TBO08b]. Even if it is an improvement

over RTSJ, it has problems of its own [MSR07]. One of these is that its usage of class inheritance makes many interfaces accessible through its API even though they are not usable. Another undesirable aspect of Ravenscar Java is that it includes scheduling parameters as part of the program logic, even though it can be analysed from the program code by a tool. Ravenscar is further described in [JKK02, TBO08b].

SCJ's overall idea is to keep things simple, e.g. by leaving scheduling parameters to analysis tools [MSR07]. It also uses object composition in order to only expose usable interfaces and the classes are kept to a minimum. This lower the access to hard real-time programming for Java programmers. In addition a further implementation of SCJ is described in [TBO08a] where it is called SCJ2. This introduce a few features that further lower the barrier to hard-real time programming.

For all those reasons SCJ2 is the chosen profile for this project and it is described in the remaining of this section. This description is based on [TBO08a].

Safety Critical Java 2

An application running on SCJ2 can only be in one of the three states as showed in Figure 2.2 during its lifetime. The changes of state occur when the singleton class `RealtimeSystem` `start` and `stop` method are called. The three states are described below followed in Listing 2.1 by a depiction of the class `RealtimeSystem` which represents the runtime system.

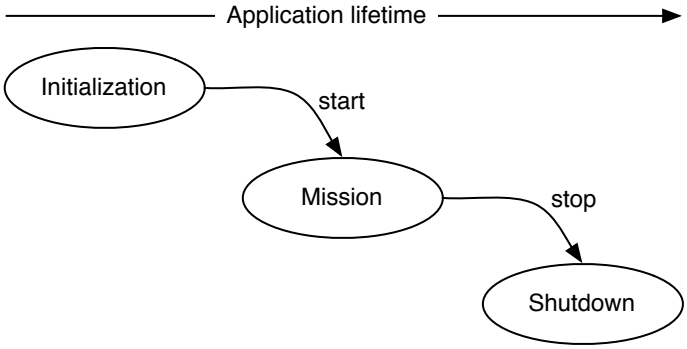

Figure 2.2: The three states an SCJ2 application can be in during its lifetime.

Initialisation During this phase, the memory used during the mission phased and the threads are instantiated. This is done in the main method and has no timing constraint. When finished `RealtimeSystem.start()` is invoked to start the mission phase.

Mission This phase is when the application runs in real-time and thus deadlines applies. It runs until `RealtimeSystem.stop()` is called.

Shutdown Before entering this phase, the application ensure that no threads are in a critical state. Then during the phase, the application cleans up and shutdowns.

```
public class RealtimeSystem {
  private RealtimeSystem ()

  public static void start()
  public static void stop()
  public static void fire(int event)
  public static RelativeTime currentTime()
  public static RelativeTime currentTime(
                              RelativeTime destination)
}
```

Listing 2.1: The singleton class `RealtimeSystem` which represent the runtime system

Real-time Threads and Paramters

As discussed before, real-time applications have different task to accomplish either periodically or aperiodically. As SCJ2 is designed for hard real-time systems, aperiodic task are actually sporadic. These tasks are executed by different threads and SCJ2 has two types of them: periodic threads and sporadic threads. These threads have two methods in common as showed below in Figure 2.3 followed by their description.

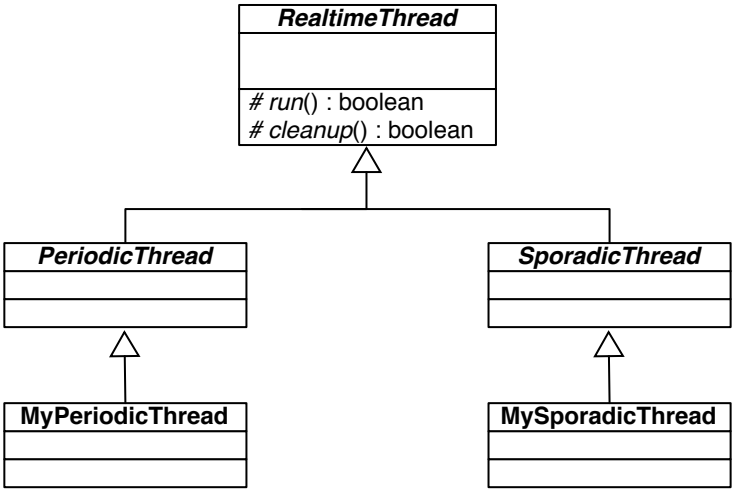

Figure 2.3: Structure of the thread classes

`run()` This is the method called during the mission phase to execute the task that the thread represents. The boolean that it returns indicate whether the thread if the thread can be shut down which is when it is not in a critical state.

`cleanup()` This method is called during the shutdown phase, when all threads have returned true from their `run()` method.

The constructors of `PeriodicThread` and `SporadicThread` must be supplied with `PeriodicParameters` and `SporadicParameters` respectively. These classes encapsu-

late the required arguments for the sake of readability. This is one point where SCJ2 differs from SCJ where parameters are specified directly into the constructor.

Listing 2.2 shows the definition of `PeriodicParameters` where `period` represents the release period, `deadline` represents the deadline which must be lower than the period in order for the thread to be schedulable, and `offset` represents the offset before the thread is triggered for the first time after the initialisation phase. It should be noted that the implementation of SCJ2 on the JOP processor does not support periods longer than 35 minutes (represented in microseconds), because JOP uses 32 bit integers and Java only supports signed integer.

```
public class SporadicParameters {
  public PeriodicParameters(RelativeTime period,
                            RelativeTime deadline,
                            RelativeTime offset)

  public PeriodicParameters(RelativeTime period,
                            RelativeTime deadline)

  public PeriodicParameters(RelativeTime period)

  public final RelativeTime getPeriod()
  public final RelativeTime getOffset()
  public final RelativeTime getDeadline()
}
```

Listing 2.2: The definition of `PeriodicParameters`

Listing 2.3 shows the definition of `SporadicParameters` where `event` is an unique identifier of a specific sporadic thread. This is used when an event occurs and the corresponding task must be fired, calling `RealtimeSystem.fire(int event)` showed in Listing 2.1. The `minInterarrival` represents the minimum interarrival time and the `deadline` obviously represents the deadline which has a default value equal to the `minInterarrival`. The original SCJ uses strings to identify events, but the authors of [TBO08a] argues that a similar level of explicitness can be achieved by defining constants.

```
public class PeriodicParameters {
  public PeriodicParameters(int event,
                            RelativeTime minInterarrival,
                            RelativeTime deadline)

  public PeriodicParameters(int event,
                            RelativeTime minInterarrival)

  public final int getEvent()
  public final RelativeTime getminInterarrival()
  public final RelativeTime getDeadline()
}
```

Listing 2.3: The definition of `SporadicParameters`

SCJ2 provides several class that inherit from `RelativeTime` to represent the time with granularity. The goal is to release the developer from the burden of making its own conversion or handling the wrap around. It provides class for measuring time in nanoseconds, microseconds, milliseconds and seconds. This is one addition of the features added on top of SCJ which only gives time in microseconds (from the `RealtimeSystem.currentTime()` method).

This feature unfortunately introduce a cost in execution time and memory consumption, which is why they also provide an optimized alternative where `RelativeTime` is omitted and replaced by an integer representing microseconds. `RealtimeSystem.currentTime()` is then replaced by `RealtimeSystem.currentTimeMicros()`.

Restrictions

Some restrictions apply to SCJ2 and they are presented here. SCJ2 lacks a memory model, which has as consequence that no object should be created during the mission phase. This actually is a disadvantage compared to SCJ which implement a memory model similar to Ravenscar Java composed of immortal and scoped memory.

Recursion is not allowed since it can have unpredictable behaviour and adds complexity to the analysis. Furthermore, the standard Java library are prohibited since the are not optimized for hard real-time and can have an unpredictable WCET. Dynamic class loading is disabled to allow for static analysis. The profile does not actually enforce these restrictions.

2.4 Tools

This project uses exclusively the tool names SARTS which was specifically developed with SCJ2 [TBO08a]. SARTS is a model-based schedulability analysis tool for hard real-time systems, which translates hard real time java system into UPPAAL model. It is developed with JOP as hardware platform target. [TBO08a]

It is used in this project to make sure the prototypes are schedulable. It provides a tool to measure the WCET of a specific method called `MethodWCET`. This tool is used to measure the WCET in the different test of this project.

SARTS was chosen due to the availability of its source code ([Boe08]), the fact that it is specifically build for SCJ2. In addition its schedulability analyse is less pessimistic than other tools available [TBO08a].

Chapter 3

Match Algorithms Presentation

In Section 1.1 we discovered the structure of expert systems. The inference engine is where an expert system selects the rules that match the fact base's current state. The inference engine then choose in which order the selected rules have to be fired. This is usually called conflict resolution and is for example based on priorities attributed to each rule. Finally, the rule's RHS are runned in the choosen order. This process is illutrated in Figure 3.1. [Doo95, Lug05]

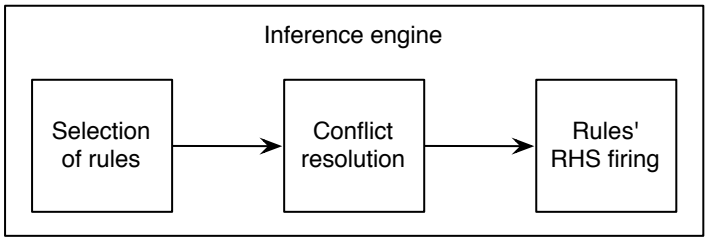

Figure 3.1: An illustration of the components comprised in an inference engine and how they interact with each other.

The set of rules selected are usually called the conflict set and producing it, requires to match each rules' conditions against each facts contained in the fact base [Doo95]. This is the role of the multiple match algorithms we are looking at in this project. Since we do not further investigate conflict resolution nor rule firing, the RHS of the rules will not be described.

In this chapter, I will describe Treat as well as Leaps using a set of example rules and facts. These are presented in Section 3.1 together with a description of the composition of a rule and a fact. Since both Treat and Leaps reuse concepts introduced by Rete, it is descibed after the example presentation.

3.1 Examples Presentation

The example introduced here is to be used to present the algorithm in this chapter. It is purposely simple in order to keep the explanation of the algorithms clear. The

chosen example is inspired from a simple child play called "Block" and consisting of a set of blocks painted in various colours. These blocks are then stacked on a table.

Figure 3.2 shows the particular set of blocks we are using for this example, as well as their disposition. Each bock has been given an identifier in the form of `B#` where `#` is a number (e.g. `B2`). Our fact base will be constituted by the information contained in Figure 3.2 and it will be called *working memory* (WM), which is the term used in the literature [Doo95, Mir87, Bat94].

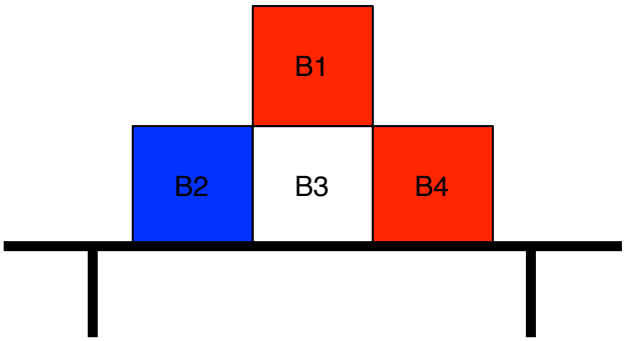

Figure 3.2: The blocks and their position on the table. Each block has an identifier starting with `B` followed by a number.

The goal is to find the colour pattern of some simple country flags such as France or Indonesia. In Figure 3.2 we can see that `B2, B3, B4` form a French flag or that `B1, B3` form the Indonesian flag. In order to find these flags, we need to define them in a formal way so they can be used in a rule.

The formal language we are going to use is *OPS5*[1] as it is very simple and used throughout the literature. It is also very similar to *Soar* which is used by [Doo95] to describe Rete. OPS5 represent a fact or a *working memory element* (WME) in the following syntax: `(identifier ^attribute value)` where it is possible to have multiple pair of attribute and value [Bro86]. Fact 3.1 formally describes the blocks from Figure 3.2 and their positions in the block world.

```
(B1 ^colour red ^on B3)
(B2 ^colour blue ^on table ^left-of B3)
(B3 ^colour white ^on table ^left-of B4)
(B4 ^colour red ^on table)
```

Fact 3.1: The blocks `B1`, `B2`, `B3` and `B4` formally described in OPS5

The rule base, also called *production memory* in the literature, contains the rules or productions. The LHS of these rules in composed of one or more conditions that needs to be satisfied in order for the rule's RHS to be fired. In OPS5 a rule has an identifier a well as a name that serves as a description. Variables can be used in the conditions and are denoted like this <x> [Bro86]. Below is an example of a rule in OPS5 showing its structure. The sign `-->` show the beginning of the RHS [Bro86]. It will not be further used.

[1] More on OPS5 in [Bro86].

```
(identifier name
  (<x> ^colour red)    ;This is a comment.
  (<x> ^left-of B3)
-->
  ;RHS
)
```

Rule 3.2 finds the pattern of a French flag in the current set of blocks. It uses three variables (`x`, `y` and `z`) to represent the blocks and states that the colour of these blocks should be from left to right: blue, white and red. In the block world showed in Figure 3.2, it should find a French flag combining the blocks B2, B3 and B4 in that order.

```
(r1 French-flag
  (<x> ^colour red)     ; C1
  (<y> ^colour white)   ; C2
  (<z> ^colour blue)    ; C3
  (<z> ^left-of <y>)    ; C4
  (<y> ^left-of <x>)    ; C5
)
```

Rule 3.2: A rule finding a French flag pattern. After each condition, a comments gives it a name for later references.

Rule 3.3 finds the pattern of an Indonesian flag in the current set of blocks. As it has only two colours (white and red) on top of each other, only two variables are necessary. This rule should find the blocks B1 and B3 in the block world pictured in Figure 3.2.

```
(r2 Indonesian-flag
  (<x> ^colour red)     ; C6
  (<y> ^colour white)   ; C7
  (<x> ^on <y>)         ; C8
)
```

Rule 3.3: A rule finding an Indonesian flag pattern. After each condition, a comments gives it a name for later references.

Having defined the fact base (Fact 3.1) as well as the rule base (Rule 3.2 and 3.3) of our example, we can go on to explain how the different algorithms work to find the French and Indonesian flags in the block world presented in Figure 3.2.

3.2 Rete

The first description of Rete is Forgy's thesis [For79], which is unfortunately not available in digital form making it more difficult to find than [For82]. The later describes an implementation of Rete for a particular low-level machine which tends to confuse the reader. [Doo95] provides us with a readily available description that is clear and well constructed. As a result the following is based on [Doo95].

In order to keep the complexity of the description low, a change of structure for our example WMEs described in Fact 3.1 is needed. Basically, each WME will be considered as a triple using the form: `(identifier ^attribute value)`. This results in more WMEs, but does not render the model less powerful [Doo95]. The resulting WMEs are presented in Fact 3.4 and are given names.

```
w1:(B1 ^colour red)   w2: (B1 ^on B3)
w3:(B2 ^colour blue)  w4: (B2 ^on table)  w5:(B2 ^left-of B3)
w6:(B3 ^colour white) w7: (B3 ^on table)  w8:(B3 ^left-of B4)
w9:(B4 ^colour red)   w10:(B4 ^on table)
```

Fact 3.4: The WMEs presented in Section 3.1 in the form of triples.

Overview

Rete is based upon a dataflow network having two main features: *state saving* and *node sharing* between productions. State saving allows to only compute the new or updated WMEs, which saves computational time, while sharing nodes between productions saves space and reduce the complexity of the network.

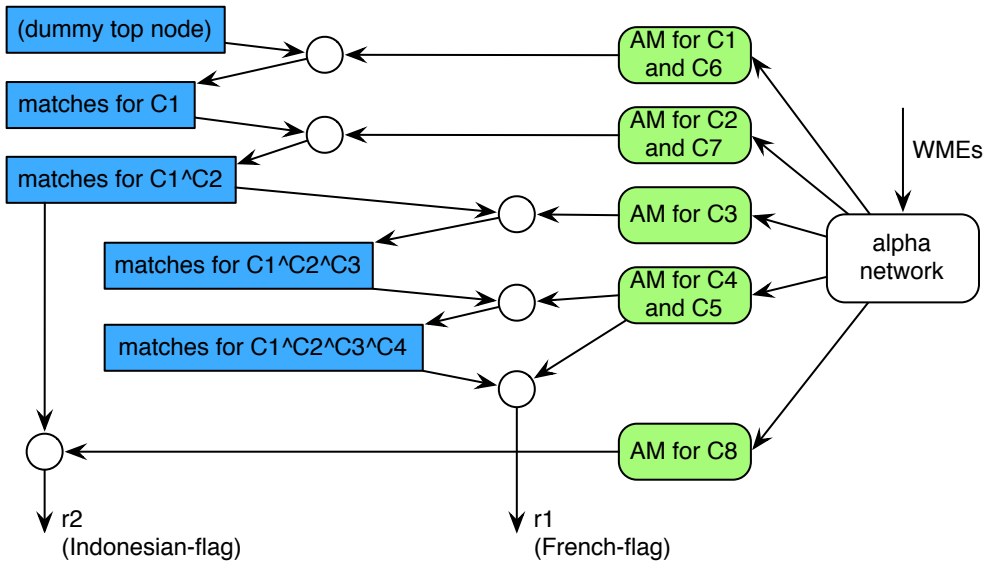

Figure 3.3: Overview over a Rete network, where the green nodes are alpha memories (AM), the blue nodes are beta memories and the round node are join nodes. The alpha network reuses the alpha memories that matches C1 for C6 and C2 for C7 since they are the same.

The network is composed of two distinct sub-networks: the alpha and beta networks. Each network has its own task and provides state saving facilities in the form of *alpha memory* (AM) nodes the alpha network, and *beta memory* nodes for the beta network. The alpha network purpose is to test the WMEs for single conditions and store them in an alpha memory.

The beta network joins the alpha memories with its beta memories in order to form partial matches of a production and eventually, match a production. A production

that is matched is said to be instentiated while partial matches are called *tokens*. Figure 3.3 illustrates a Rete network based on the example presented in Section 3.1. The alpha memories are showed in green and the beta memories in blue. Note that some nodes are shared between conditions or partial matches.

The Rete algorithm can be interacted with, in four ways. Adding and removing WMEs to react to a change in working memory. Note that an update is handled by removing and then adding again the concerned WME. The two last are adding and removing a production. Below, we first describe the alpha network and the beta network for addition of WMEs. The addition of productions and removing of both production and WMEs are treated following that.

Alpha Network

When a WME is added to the working memory, it first goes through the alpha network where the necessary tests are performed. Figure 3.4 shows an alpha network for the example presented in Section 3.1. As you can see the network first tests which attribute the WME has, and then if necessary goes on to test the value of this attribute. More complex WMEs, e.g. typed WMEs, can add levels to this network, but the principle stays the same.

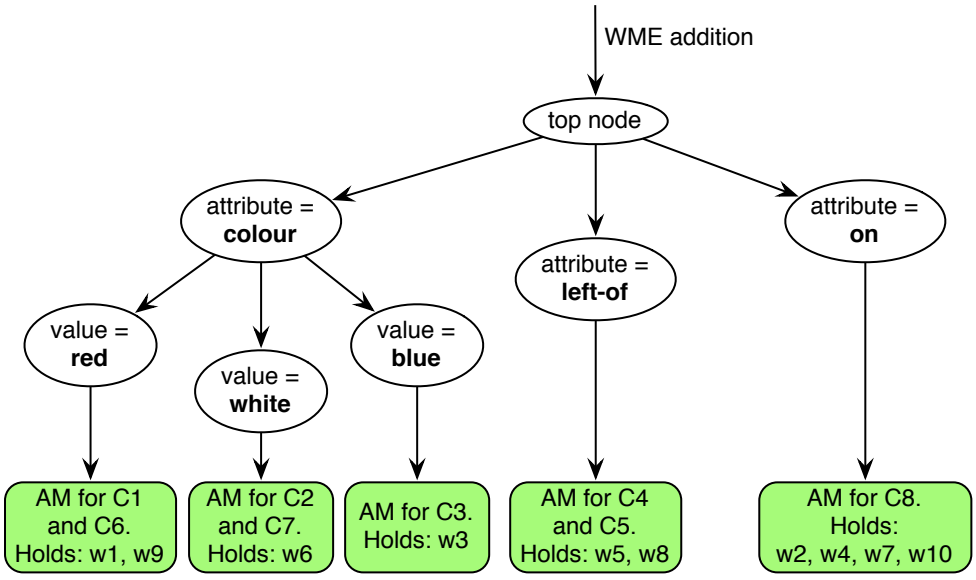

Figure 3.4: An alpha network build with the example showed in Section 3.1. Notice the alpha memories holding the WMEs that fulfil their respective conditions.

The nodes performing the test are called one-input nodes or constant-test nodes. As their name indicates, they perform the constant tests and do not test for variable binding consistency apart from one exception. When a variable appears more than once in a condition, e.g. `(<x> ^self <x>)`, the alpha network performs an intra-condition test.

The tests performed in the constant-test nodes can be any boolean function and Rete's implementations usually support at least simple relational test, such as greater-than,

lesser-than, etc. Some implementations even allows user defined tests [All83, For84].

Looking at our example and Figure 3.4, let's see what happens when adding the WME `w9`. First it goes through the top node which does not perform any test and just sends the WMEs through to all its children. At the attribute level, `w9` is tested in all three nodes and only passes through the one corresponding to it, `colour`. At the value level, `w9` only passes through the node testing for `red` and is then saved in the alpha memory that matches `C1` and `C6`.

Beta Network

In a beta network there is mainly two types of nodes: beta memories and join nodes. As we have seen, each beta memory holds a set of tokens that are partial matches of a production, and have one parent and one or more children. The join nodes bind the variables between the tokens and WMEs of two parents, a beta memory on the left and an alpha memory on the right. The join results in new tokens that are added to its child, which is a beta memory as well. Other nodes like production nodes or negation nodes exits, but to keep this presentation simple I will not talk further about them.

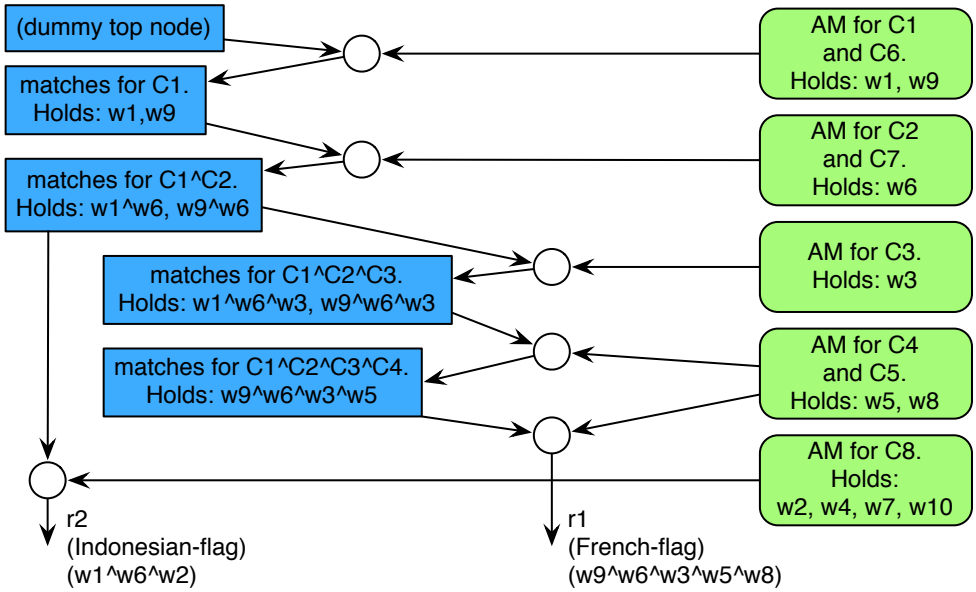

Figure 3.5: A beta network build with the example presented in Section 3.1. Notice the beta memories holding sets of tokens.

A join node can either be activated by its parent alpha memory or its parent beta memory. These activations are respectively called right and left activation and happen when a one of the nodes parent adds a token or a WME to its memory. In the case of a dummy beta memory node, the join is always successful which is necessary to have at least on beta memory holding a token so the next join node actually has something to join. When activated from one side, a join node searches its other side's parent for items having variables binding consistent with the new item and joins them if any are found.

An illustration of the beta network build from the example showed in Section 3.1 is showed by Figure 3.5. In each beta memory node, there is a set of tokens that were joined by the node parents. Notice that `r1` and `r2` share the same path in the beginning as their two first conditions are identical.

Looking again at our example, let's assume that all the WMEs but `w5` and `w8` are already added. Adding `w5` will result in the right activation of both join nodes attached to the AM for `C4` and `C5`. The lower join node would be activated first and since its parent beta memory is empty at that moment, it cannot join anything. Then the higher join node would be activated and would bind <z> to `B2` and <y> to `B3`, passing them to its child beta memory. The beta memory that just added `w5`, would activate its child join node, which would unsuccessfully try to bind `w5` with <x>. Finally `w8` is added to the alpha memory which activates the lower join node first, and binds <x> to `B4` completing the match for `r1`.

3.3 Treat

Treat is inspired from Rete and shares the same general idea as it is also a dataflow network divided in two parts. It is designed by Daniel P. Miranker and was first described in his Ph.D. Thesis later published in [Mir90]. Miranker states inside his thesis and the article [Mir87], that Treat outperformed Rete in 5 cases [Mir87, Mir90]. This is disputed by [NGR88], arguing about the measurement methods. However, as I am interested in knowing if Treat can be used for safety critical applications, speed is not critical.

The rest of this section is based on both [Mir87] and [Mir90] unless stated otherwise. Since Treat is a dataflow network divided in two parts as Rete and to avoid confusion, I use the terms *alpha* and *beta* as for Rete to describe the parts Miranker call *select* and *join*. I first describe the alpha network, followed by the beta network and WMEs removal.

Alpha Network

The alpha network of Treat is almost identical to the one of Rete, with the exception of the alpha memories. The part filtering the WMEs before they arrive to the alpha memories is exactly the same as for Rete. Miranker implement it with hash tables in order to optimize it, but this is also applicable to Rete as shown in [Doo95]. As in Rete the goal of Treat's alpha network is to performs constant tests as well as intra-condition tests. Take a look at Section 3.2 and especially Figure 3.4 if you need to refresh your understanding of alpha network.

Figure 3.6(a) shows Treat's alpha memories in green. As you can see, each memory has three partitions named: Old, New and Del. The Old partition holds WMEs that have already been added to the network. The New partition holds a new WME that is currently in the process of being added. It will only stay in the New partition until the computations provoked by the addition are finished, and then it is moved to the Old partition. As for the Del partition which stands for "delete", it holds on a WME currently in the process of being deleted. When the computations needed to delete the WME is finished, it is then removed from both the old and the delete partition.

Looking at the example from Section 3.1 and at the Treat alpha memories in Figure 3.6(a), let's see what happens when the WME `c1` is added. Remember that `w1` passes through the alpha network before arriving to the alpha memory for the conditions `C1` and `C6`. So upon addition of `w1`, it is added to the New partition of the alpha memory, triggering the beta network. When the computations are done, it is moved from the New partition to the Old partition.

As for deletion, it is illustrated be Figure 3.6(b). First `w1` goes through the alpha network and then is added to the Del partition of the alpha memory. This trigger the deletion in the beta network and delete `w1` from the Old and Del partitions when the beta network computations are finished.

Beta Network

The beta network of Treat differs from the one of Rete as it does not save intermediates state in beta memories, but do keep track of which WMEs are used in which instantiated productions. Treat then needs to recompute the joins on every WME addition which could seem costly. However on WME deletion, it only needs to delete instantiated productions containing the deleted WME, thus avoiding the cost of keeping up to date any beta memories.

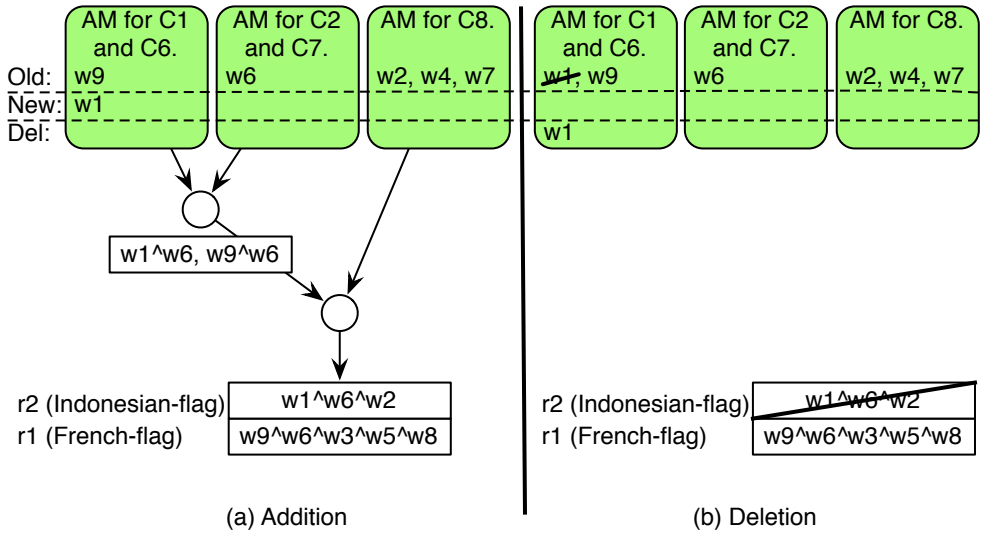

Figure 3.6: (a) shows the addition of WME `w1` to an alpha memory and the subsequent activation of the beta network. (b) shows the deletion of WME `w1` and the subsequent deletion of the instantiated production `r2` that depended on it.

A Treat beta network is activated by the alpha memory having a non empty New partition. The WMEs contained in the alpha memory are then joined with WMEs from another alpha memory. This is done with join nodes similar to the one in Rete resulting in new tokens that are then joined with another alpha memory which in turns creates new tokens. This process is followed until no more joins are possible and results in the instantiation of productions rules if any of them were matched. An example of a Treat beta network is shown by Figure 3.6 where (a) shows an addition and (b) a deletion.

Continuing the example started in the alpha network description of Treat, let us see what happens during the addition of WME `w1` in Figure 3.6(a). The presence of `w1` activates the join node below it, where `w1` and `w6` are joined to form a token. `w9` and `w6` are also joined to form a token as Treat uses both the New and Old partition for the joins. The second join node joins the token `w1^w6` to the WME `w2` resulting in the instantiation of production `r2`, which detects an Indonesian flag. The production `r1` was already instantiated and is left untouched. Note that not all alpha memories are depicted in the figure.

In Figure 3.6(b) a deletion is presented. The process is straight forward as the presence of `w1` triggers the deletion of the instantiated production `r2` followed by the deletion of `w1` from the alpha memory's Old partition. Remember that Treat keeps track of which WMEs are used by the instantiated productions.

3.4 Leaps

Leaps was presented for the first time in [MBLG90] by Miranker et al. and stands for Lazy Evaluation Algorithm for Production Systems. Both Rete and Treat maintain a conflict set where all the productions that are instantiated can be fired. [MBLG90] show that when a production is fired, many productions that are waiting to be fired in the conflict set becomes obsolete since the fired production often have a RHS that changes the working memory.

Leaps central concept is to compute only one production and immediately fire it. In order to do that, Leaps needs to resolve the conflicts while it is trying to find a matching production. If you look back at Figure 3.1, you can see that Leaps includes both the productions (rule) selection and the conflict resolution while Rete and Treat only includes the productions selection.

Since Leaps uses an *alpha* network as well, it will not be described again. The only difference with Rete's alpha network lies in the alpha memories. Leaps add a timestamp to the WME each time it is saved in an alpha memory. Looking at the example used throughout this chapter and assuming that the block were added from left to right and bottom to top (`B2`, `B3`, `B4`, `B1`), Figure 3.7 shows the WMEs and their timestamp inside the alpha memories. In order to remind you what WMEs belong to which block, their definition is repeated in Fact 3.5.

```
w1:(B1 ^colour red)   w2: (B1 ^on B3)
w3:(B2 ^colour blue)  w4: (B2 ^on table)  w5:(B2 ^left-of B3)
w6:(B3 ^colour white) w7: (B3 ^on table)  w8:(B3 ^left-of B4)
w9:(B4 ^colour red)   w10:(B4 ^on table)
```

Fact 3.5: The WMEs presented in Section 3.1 in the form of triples.

Notice that memories order their WMEs using the timestamp. This property is used later when joining the WMEs in the part of the algorithm that would be called beta network in Treat and Rete. This is explained in the following section.

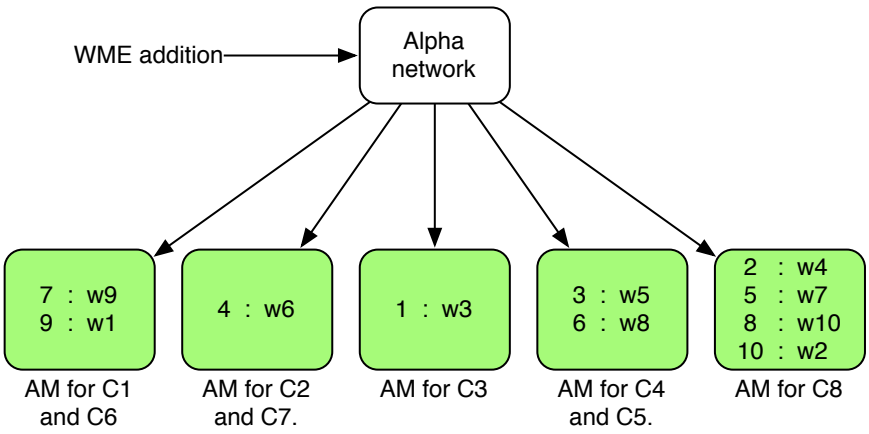

Figure 3.7: The alpha memories in Leaps where each WME has a timestamp that was added to it when saved.

Computing Production Instantiations

As explained above, Leaps fires the first production that it finds. In order not to fire the same instantiation twice, it must save the search state. This is done with a stack as showed in Figure 3.8. Elements of the stack consist of a set of pointers representing the state of a best first search for instantiations. In Figure 3.8 the elements of the stack consist of the timestamp of the WMEs the pointers are pointing to.

For simplicity, the example presented here only uses Rule 3.3, which searches for Indonesian flag patterns. Figure 3.8 (initial state) represents the initial state before the search begins. For each rule that has $n-1$ condition elements, the set of pointers is "$ts_0, \ldots, ts_{n-1}$", where ts_i is timestamp. Each time a WME is added, a corresponding initial search is pushed to the stack in the form of "$ts_0 - 1, \ldots, ts_i, \ldots, ts_{n-1}$", where ts_i is the timestamp of the newly added WME.

For each entry in the stack, there is dominant timestamp (DT) that is used to seed the best-first search. The DT is set to the most recent timestamp in a stack entry. In Figure 3.8 (initial state), the entry at the top of the stack has DT equal to 10. Seeding search with the most recent timestamp results in an algorithm that fires the most recent instantiation.

When the search begins, the top element of the stack is popped and the search seeded with the DS of this element. The other pointers are pointing on the highest timestamp the corresponding timestamps in the element allows. In the example, Figure 3.8 (a) shows where the pointers point when the first element of the stack is popped. If the WMEs pointed at match the rule, the search is stopped and the rule fired. This is the case in the example. Assuming that the working memory is not altered by the rule fired, the search continues.

The search then tries to match the rule with WMEs that have lower timestamps than the one tried before. Here it is arbitrary decided that it tries in a left to right manner. The seed does not change and the search backtracks if necessary. Figure 3.8 (b) shows that the search continues with trying to match WME `w9`.

When the search with a specific seed is exausted, the algorithm pops the next element of the stack and seed the search with its DT. In the example showed in Figure 3.8 (c),

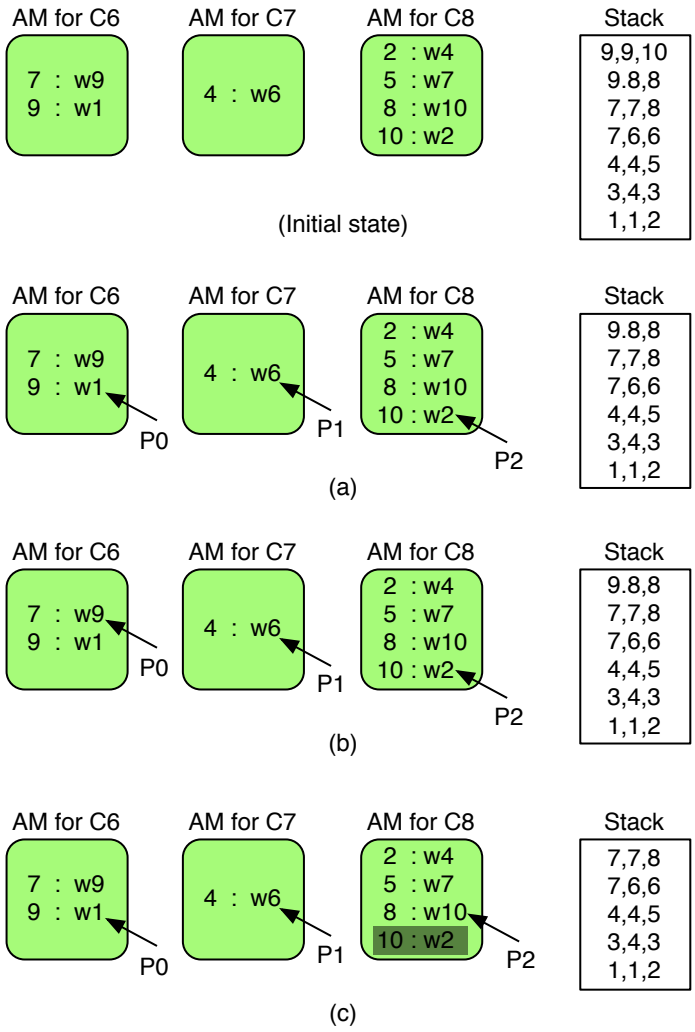

Figure 3.8: This illustrate the best-first search, the stack and the pointers used in Leaps to match rules.

the new seed is 9 with WME `w1`. Notice that the WME with timestamp 10 is shaded as it will not be included in the search again. From here the search continues until it finds a combination that matches the rule. When the stack is empty, it stops.

If the working memory is altered by a fired rule, the current search state is pushed on the stack again eventually followed by a new element pushed because a WME was added.

Chapter 4

Real Time Expert System

This chapter presents a real time expert system developed to work with SCJ2 and use SARTS as an analysing tool. Both the design and the implementation are presented in a top down approach. First the overall design of the system is explained together with alternative designs that are evaluated. Then follows the main components that are the match algorithm and the rule language and compiler.

The match algorithms presented in Chapter 3 are discussed and the design of the one used for this expert system is described. Focus is on the WCET throughout the chapter, especially for the match algorithms since it is the main area of interest of this project.

In order to calculate a WCET for the match algorithm, certain limitation need to be in place and they are presented in this chapter. The rule language aims to make these limitations explicit in addition to providing the tools required for defining rules. It is presented in Section 4.3.

4.1 Overall Design

As Section 2.3 presented, SCJ2 has two type of threads: sporadic and periodic. Usually a thread execute a specific task, such as reading data from a sensor periodically and then fire a sporadic thread according to the data it just read. In the case of an expert system, the same periodic thread that reads a sensor gives the data to the expert system so it can make a decision on what action to fire. There are different ways to achieve this, they are listed below and then explained.

- The threads that update the data, run the expert system when data is updated, *including* resulting action.
- The threads that update the data, run the expert system when data is updated, *excluding* resulting action.
- The expert system runs in its own sporadic thread and is fired by threads that have updated data. Actions are *included* in the expert system's thread.
- The expert system runs in its own sporadic thread and is fired by threads that have updated data. Actions are *not* run in the expert system's thread.

- The expert system runs in its own periodic thread. Actions are *included* in the expert system's thread.
- The expert system runs in its own periodic thread. Actions are *not* run in the expert system's thread.

The list above shows that there are really two main questions:

1. Does the expert system run in the different threads of the system that uses it, or does it run in its own thread (sporadic or periodic)?
2. Do the resulting actions run in the same thread as the expert system that triggered them, or do they run in their own thread?

Starting with the first question, the consequences of the two solution are here explained. Running the expert system in the same thread as a thread that updates its data is closer to the systems that exist today as the decision of what to do after an update of the data would still happen inside the same thread. In that case the expert system would be able to run immediately after the data is updated, whereas if the expert system is to run in its own thread it would have to wait to be scheduled after being fired. One major disadvantage of running the expert system in the same thread is that the WCET of every thread that include the expert system will have to include the expert system WCET, which changes each time the rules changes or when a new WME is added to the system.

If a system is running the expert system in a dedicated thread, the decision process would not start instantly as explained above. However updating the rules or adding new WMEs would only influence the WCET of the expert system thread itself, making it easier to maintain the program using the expert system. Keep in mind that one goal of an expert system is to ease the maintenance of the code, and especially permit to change the rules frequently. Running in its own thread, the expert system can be both triggered periodically by using a periodic thread, or on demand by using a sporadic thread. Because of the ease of maintenance coming to the small cost of a small delay when firing, this expert system is running in its own thread.

As for the second question, running an action in its own thread has the advantage to allow not implementing actions in the rule language. The actions can then be implemented by a specialist of real time systems using a sporadic thread, a solution than blends nicely with SCJ2's design. So running an action in its own thread, allow for more flexibility than running in the same thread as the expert system, however a small delay is introduced again as it has to be scheduled. In this case the more flexible solution is chosen as it is elegant as well.

Implementation

As we saw above, the expert system implemented, run in its own thread either sporadic or periodic, and the actions are implemented as sporadic thread. In addition to the advantages discussed, it should be noted that running the expert system in its own thread eases the measure of the effect of rules updates. The class representing the real time expert system in shown in Figure 4.1 where in this case inherit from `SporadicThread`.

The application using SCJ2 has three states as described in Section 2.3. When being in the *initialization* state, `RTExpertSystem` is being created, which in turns starts

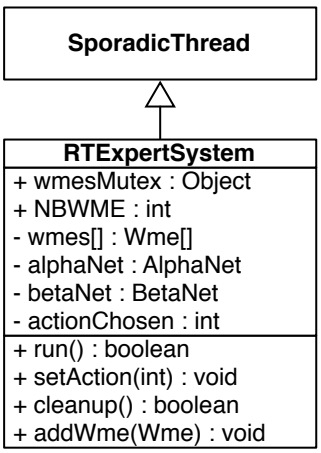

Figure 4.1: The class representing the real time expert system and its API

the creation of the required classes of the match algorithm that are described in Section 4.2. This is during this state that the rest of the application can add the wmes using `addWme(Wme wme)`. The number of WMEs is limited by `NBWME` which is itself defined with the rules as described in Section 4.3. Limiting the number of WMEs is done in order to be able to set a bound on the loop via the `@WCA` comment annotation shown in Listing 4.1. As we'll see in Section 4.2, limiting the number of WMEs is necessary to set loop bounds et limit array size in the match algorithm as well.

```
public class RTExpertSystem extends SporadicThread {

  // int for iterating in loops
  private int i = 0;

  private int actionChosen;
  public static final int NBWME = 10;
  private Wme[] wmes;
  public Object wmesMutex = new Object();
  private AlphaNet alphaNet;
  private BetaNet betaNet;

  public void run() {
    actionChosen = -1;
    synchronized(wmesMutex) {
      matchAlgo.clearMemories();
      for (i=0; i < NBWME; i++) { // @WCA loop=10
        alphaNet.passThrough(wmes[i]);
      }
      actionChosen = betaNet.passThrough();
    }
    if (actionChosen > -1) {
      RealtimeSystem.fire(actionChosen);
    }
  }
```

```
}
```

Listing 4.1: The class `RTExpertSystem` and its `run()` method.

The second application state being *mission*, this were the expert system does its work. When the thread is being fired, SCJ2 calls the `run()` method which in turns uses the match algorithm to find a rule to fire. The match algorithm sets the action (a sporadic thread) to be fired via the `actionChosen` field. As you can see there is only one rule fired each time. This is inspired from the Leaps algorithm where they show that in a the majority of times, firing one rule renders the other rules to be fired obsolete since the rule fired changes the working memory.

Looking at Listing 4.1 and the `run()` method in particular, you can notice that a mutex is used to ensure that the other threads don't modify the WMEs while the match algorithm finds instantiations. You can also see that the memories are cleared and that every WME is passed through the alpha network. This is done because the worst case scenario is that all WMEs have been updated. Not looking for updated WMEs and simply passing them all through the algorithm every time simplifies the match algorithm without changing the WCET. Finally it triggers the betaNetwork to find the best instantation based on their priority.

4.2 Match Algorithm

The match algorithm is specially designed with WCET in focus and inspired by all three algorithms presented in Chapter 3. Both Leaps and Treat are created to be faster than Rete in average time, but since in safety critical applications only WCET matters, their optimizations are not helping.

But as we saw earlier, this algorithm only yield one rule to be executed when it find one. This is inspired by Leaps since it observes that actions change the working memory and in a majority of times, renders the other instantiations obsolete. In a real time application an action would probably trigger a reaction in the real world which would then be detected by sensors and thus change the working memory.

All three algorithms use an alpha network with different memories. This algorithm that I call Owcma (Optimized for Worst Case Match Algorithm), uses an alpha network as well together with the simple memories that Rete uses. This is be discussed in Section 4.2.

Owcma, like Treat and Rete uses a beta network to join the condition elements (CDE) represented by the alpha memories. The beta network is discussed in Section 4.2.

The match algorithm first passes all the WMEs through the alpha network as seen before and then triggers the beta network. Each time an incantation in found, it is set as the chosen instantiation provided that its priority is higher than the one already chosen. When the beta network can't find more instantiations, it returns the value of the chosen instantiation.

Implementation of the Alpha Network

Three different ways of implementing the alpha network were tried. The first one was having different nodes checking for the different unary conditions, that would

implement a node interface. This is thought as to be the most elegant and object oriented method. The first tests revealed that the chosen analysing tool SARTS does not support Java interfaces.

The second approach replaced the interface with an abstract class that the nodes inherited and implemented the abstract methods. This is equivalent to using an interface, but less elegant. The WMEs were sent through tha alpha network using the method `passThrough(Wme wme)` presented in Listing 4.2.

```
public void passThrough(Wme wme) {
  curNode = rootNode.getFirstChild();
  // Traverse the tree
  // The bound is the sum max nb child per height
  // (root node not included in bound since it's bypassed
      before entering the loop)

  while(curNode != null) { // @WCA loop=4
    curNode.setPassingWme(wme);
    if (curNode.hasFirstChild()) {
      curNode = curNode.getFirstChild();
    } else { // arrived to a leaf node
      curNode = curNode.getNextSibling();
    }
  }
}
```

Listing 4.2: The method used to pass WMEs through the alpha network using nodes inheriting from an abstract node class.

The methods used on the nodes in Listing 4.2 were all defined in the abstract class `Node`. `node.hasFirstChild()` would only return true if the current WME passed the node tests. When it did not pass the test, the algorithm tries the next sibling until it reaches a memory node or that it did not pass any test on a level (i.e. the attribute level.). Unfortunately this design was abandoned since inheritance creates an explosion of states to check when using SARTS and making the analysis of WCET too slow to be practical (more than 24 hours).

The final design uses the fact that the network is generated by a compiler from rules, to hard code the tests and the graph in the nodes. There is one node per similar tests as shown in Figure 4.2. The tests are hard coded in each node using simple `if then else` as shown in Listing 4.3 which is the attribute test for the example rules presented in Chapter 3. `Wmes` is a class holding all the possible value of a WME as constant for convenience.

```
public void passThrough(Wme wme){
  if (wme.attribute == Wmes.ATTRIBUTE_COLOUR) {
    colourTests.passThrough(wme);
  }
  else if (wme.attribute == Wmes.ATTRIBUTE_LEFTOF) {
    leftofMemory.add(wme);
  }
  else if (wme.attribute == Wmes.ATTRIBUTE_ON) {
    onMemory.add(wme);
  }
}
```

Listing 4.3: The attribute tests node with hard coded tests for the rule presented in Chapter 3.

Alpha memories store the WMEs that gets through in an array. The size of the array is of the total number of WMEs since they could all have the same values and thus arrive in the same alpha memory node. The `AlphaNetMemory` is presented in Figure 4.3 where you can also see what API it presents. Note that the `clear()` method just sets `lastAddedIdx` to -1 so the old memories are just written over. `nbWme()` returns the number of WMEs added since the last clear to avoid reading old WMEs.

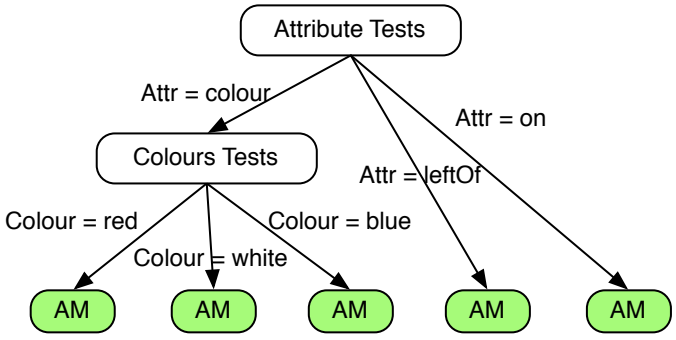

Figure 4.2: The alpha network as it is implemented. It represent the same alpha network as the one showed in Chapter 3.

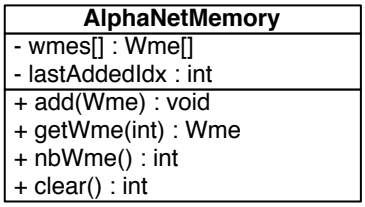

Figure 4.3: The class representing an alpha memory.

Implementation of the Beta Network

The beta network is based on Treat's beta network with dummy nodes like the one in Rete that are used to start the process of joining the nodes. A token with the combination of WMEs is passed between the join nodes until it fails to join or it arrives at an instantiation node. Instantiation nodes try to elect their actions as the chosen action, which gives priority according to the priority number the rule was given. Figure 4.4 illustrates this process.

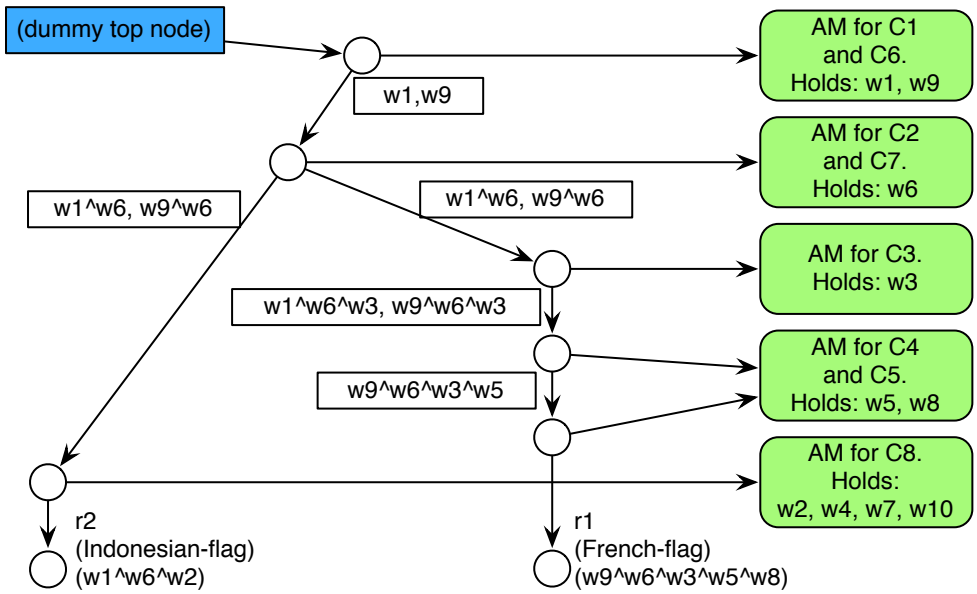

Figure 4.4: Illustration of the implementation of the beta network using the example presented in Chapter 3.

The token is simply a double array of WMEs (`Wme[][]`) that is passed along to join nodes. It holds the different groups of WMEs that have been join together, where a group is an array of WME. The join nodes are implemented in the same way as alpha nodes, which is the test are hard coded. It goes through the groups present in the token and tries to associate them with the WMEs found in the alpha memory the join is associated with. If it succeeds making one association, it sends the token to the next nodes. Listing 4.4 shows how this process is started.

```
public class BetaNet {
  // The number of dummy node is generated from the rules
  private DummyNode[] dummyNodes = new dummyNodes[1];
  private int i = 0;
  private int curAction = -1;

  public int passThrough() {
    for (i = 0; i < dummyNodes.length; i++) { // @WCA loop=1
      dummyNodes[i].startJoin();
    }
    return curAction;
  }
```

```
  public void setAction(int newAction) {
    curAction = newAction;
  }

  public int getAction() {
    return curAction;
  }

}
```

Listing 4.4: The `BetaNet` class

4.3 Rule Language

The language used is relatively simple and is heavily inspired from Drools Rule Language [Tea10]. It allows the definition of variables in order to compare different WMEs. A rule must have a priority set between 0 and 100 and finally it must have a action to fire. Listing 4.5 shows a the rule of the Indonesian flag presented in Chapter 3.

```
define numberWme = 10
define global Wmes = constants.Wmes

rule
  priority 89
  when
    $x : Block( colour == Wmes.RED )
    $y : Block( colour == Wmes.White )
    eval ( $x.on == $y.id )

  then fire 1
end
```

Listing 4.5: An example showing a rule and the definition of the maximum number of WMEs

As you can see the language also provide the possibility to define globals and require that the number of WMEs is defined. Using `eval` allows comparing the variables defined in the rule. The compiler is developed using SableCC [Sab11] and the grammar can be found with the rest of the code on the website.

Chapter 5

Conclusion

This project aimed to produce an expert system suitable for hard real time usage. The focus was set on adapting existing match algorithm in order to create one that could have a measurable worst case execution time. A rule language was to be developed in order to support this expert system and make limitations visible.

An expert system designed at run on the Java SCJ2 hard real time platform was indeed developed and described in this report. It features a rule language adapted to hard real time usage and defining explicitly the limitations. The system developed is capable of finding rules instantiations via its match algorithm. It is also capable of firing the intended actions of the found instantiations.

However none of the planned tests have been documented in this report. Conducting them has turned to be a consuming task which is not finished at the time of printing. The testing of a simple rule set with no more than 10 working elements takes more times than available as it has taken more than seventy-two hours for a single pass. Furthermore, multiple passes are necessary to find the worst case execution time of a method.

One conclusion can be extracted from this: Even if the expert system developed in this project is time predictable, the tool used to calculate its worst case execution time (WCET) is not practical to use in a development environment. Without the WCET analysis, it is not possible to use an application for a safety critical applications.

To conclude this project, I will state expert systems cannot be used in safety critical environment until the scalability of the analysis tools improves dramatically.

Bibliography

[All83] E.M. Allen. YAPS: Yet another production system. *Maryland Artificial Intelligence Group, University of Maryland, Department of Computer Science*, 1983.

[Bat94] D. Batory. The LEAPS algorithms. *Department of Computer Sciences, University of Texas at Austin, Technical Report*, pages 94–28, 1994.

[BB09] E.J. Bruno and G. Bollella. *Real-Time Java Programming with Java RTS*. Prentice Hall, 2009.

[Boe08] T. Boegholm. SARTS. `http://sarts.boegholm.dk/` Online: 28/01-2011, 2008.

[Bro86] Lee Brownston. *Programming Expert Systems in OPS5*. Addison-Wesley, student edition, 1986.

[BW09] A. Burns and A. Wllings. *Real-Time Systems and Programming Languages*. Pearson Education Limited, 2009.

[CM10] Yanik K. Challand and Tim M. Madsen. Using rule engines in home automation: A study in existing systems. 2010.

[Doo95] R.B. Doorenbos. *Production matching for large learning systems*. PhD thesis, Citeseer, 1995.

[For79] C.L. Forgy. On the efficient implementation of production systems[Ph. D. Thesis]. 1979.

[For82] C.L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Name: Artif. Intell*, 1982.

[For84] CL Forgy. The OPS83 Report. Department of Computer Science, 1984.

[FPJ07] A.L. Hosking F. Pizlo and Vitek J. Hierarchical real-time garbage collection. In *In LCTES 07: Proceedings of the 2007 ACM SIG- PLAN/SIGBED conference on Languages, compilers, and tools*, pages 123–133. ACM, 2007.

[GB] et al. G. Bollella. The Real-Time Specification for Java. `https://rtsj.dev.java.net/rtsj-V1.0.pdf` Online: 28/01-2011.

[Gen09] Gensym. G2 rule engine platform. `http://www.gensym.com`, 2009.

[Hal87] P. Haley. Real-Time for RETE. *Proceedings of ROBEXS 87: The Third Annual Workshop on Robotics and Expert Systems*, 1987.

[JKK02] A. Wellings J. Kwon and S. King. Ravenscar-Java: a high integrity profile for real-time Java. `http://www.cs.york.ac.uk/ftpdir/reports/YCS-2002-342.pdf` Online: 28/01-2011, 2002.

[Kni02] John C. Knight. Safety critical systems: challenges and directions. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 547–550. ACM, 2002.

[LCS+88] T.J. Laffey, P.A. Cox, J.L. Schmidt, S.M. Kao, and J.Y. Read. Real-time knowledge-based systems. *AI magazine*, 9(1):27–45, 1988.

[Lug05] George F. Luger. *Artificial intelligence: Structures and Strategies for Complex Problem Solving*. Pearson Education Limited, 5th edition, 2005.

[MBLG90] D.P. Miranker, D. Brant, B. Lofaso, and D. Gadbois. On the performance of lazy matching in production systems. In *Proceedings of the 1990 National Conference on Artificial Intelligence*, pages 685–692, 1990.

[McC04] Pamela McCorduck. *Machines Who Think : A Personal Inquiry into the History and Prospects of Artificial Intelligence*. A K Peters, Limited, 2004.

[McC07] John McCarthy. What is artificial intelligence? `http://www-formal.stanford.edu/jmc/whatisai/whatisai.html`, 2007.

[Mir87] D.P. Miranker. TREAT: A better match algorithm for AI production systems. *Proceedings of AAAI 87*, 1987.

[Mir90] D.P. Miranker. *TREAT: A better match algorithm for AI production systems*. PhD thesis, 1990.

[MSR07] B. Thomsen M. Schoeberl, H. Soendergaard and A.P. Ravn. A profile for safety critical java. In *In ISORC 07: Proceedings of the 10th IEEE International Symposium on Object and Component- Oriented Real-Time Distributed Computing*, pages 94–101. IEEE Computer Society, 2007.

[NGR88] P. Nayak, A. Gupta, and P. Rosenbloom. Comparison of the Rete and Treat production matchers for Soar (A summary). In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 693–698. Citeseer, 1988.

[RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.

[Sab11] SableCC. SableCC. `http://sablecc.org/` Online: 28/01-2011, 2011.

[SV07] M. Schoeberl and J. Vitek. Garbage collection for safety critical java. In *In JTRES 07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 85–93. ACM, 2007.

[TBO08a] H. Kragh-Hansen T. Boegholm and P. Olsen. Model-based schedulability analysis of real-time systems. 2008.

[TBO08b] H. Kragh-Hansen T. Boegholm and P. Olsen. Real-time java. 2008.

[Tea10] JBoss Drools Team. Drools - The Rule Language. `http://downloads.jboss.com/drools/docs/5.1.1.34858.FINAL/drools-expert/html/ch04.html` Online: 28/01-2011, 2010.