FAKULTÄT FÜR !NFORMATIK

Integration of Heterogeneous Software-Intensive Systems with Rule-Engine-Based Event Correlation, Analysis and Generation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Magister der Sozial- und Wirtschaftswissenschaften

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Paul Alexandrow
Matrikelnummer 0025410

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer/Betreuerin: Ao. Univ.-Prof. Mag. Dipl.-Ing. Dr. Stefan Biffl
Mitwirkung: Dipl.-Ing. Dr. Alexander Schatten

Wien, 17.11.2008

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

November 17, 2008

Kurzfassung

Die Integration heterogener Systeme stellt eine große Herausforderung dar. Eventbasierte Architekturen, die die Events zwischen solchen Systemen sammeln und umwandeln können, sind ein vielversprechender Ansatz dazu.

Analysten müssen große Mengen von Events aus solchen Systemen verarbeiten und relevante Muster darin erkennen können. Es gibt bereits Tools dafür, wie das Open Source Projekt Naiad, die Events aufgrund ihrer Attribute korrelieren und aggregieren können. Allerdings beschränkt sich die Korrelation in den meisten Fällen auf das Finden von identen Attributen, eine semantische Verarbeitung steht noch aus. Letztere wäre aber ein Schlüsselfaktor um Geschäftsprozesse mit der Fülle an Events aus heterogenen und dynamischen IT Systemen sinnvoll zu verbinden.

Letztlich muss jeder praktikable Lösungsansatz auch entsprechend skalierbar sein, um eine gleichbleibend hohe Performance gewährleisten zu können.

Der Kern dieser Arbeit befasst sich mit der Frage, ob und wie eine konventionelle Rule Engine gemeinsam mit einem Tool zur Korrelation von Events genutzt werden kann, um die skalierbare und semantische Verarbeitung von Events zu ermöglichen.

Zuerst soll Naiad Rules, ein allgemeines Konzept für die regelbasierte, semantische Korrelation von Events vorgestellt werden. Ziele dieses Ansatzes sind die Austauschbarkeit der verwendeten Rule Engine, die Möglichkeit einer GUI gestützten Bedienung, maximale Skalierbarkeit und die Möglichkeit der Verteilung auf mehrere physikalische Systeme. Dahingehende Probleme und deren Lösungen sollen ebenfalls präsentiert werden.

Danach soll gezeigt werden, wie gängige Patterns aus dem Bereich des Complex Event Processing mit diesem Konzept umgesetzt werden können, bzw. wo die Grenzen davon liegen.

Naiad, ein Tool zur Korrelation von Events, wird mithilfe einer Rule Engine erweitert, um das Konzept von Naiad Rules exemplarisch zu realisieren. Dabei soll das in Naiad implementierte SEDA (staged event-driven architecture) Design weitergeführt werden.

Zum Einsatz wird dabei die Open Source Rule Engine Drools kommen. Deren Integration mit den existierenden Komponenten von Naiad sowie dabei aufgetretene Probleme werden detailliert erläutert. Danach werden Engstellen und Lösungsansätze zu diesen diskutiert.

Als begleitendes Beispiel dient das SAW (Simulation of an Assembly Workshop) Projekt an der TU Wien.

Schließlich sollen Resultate hinsichtlich der Performance und der Handhabung des Konzepts vorgestellt werden, bevor ein kurzer Ausblick mögliche weitere Forschungs- und Entwicklungsziele auf diesem Sektor darstellt.

Abstract

The integration of heterogeneous systems, in particular legacy systems, is a major IT-challenge. Event-based architectures provide an approach to integrate systems by collecting and transforming the events exchanged between systems.

System integrators want to analyze the large number of events available from these systems to find meaningful patterns. There are tools for event analysis, like the open source tool Naiad, which help to correlate and aggregate event according to common event types and matching data fields. However, the correlation is typically limited to simple similarity matching, lacking support for semantic matching. The latter is an important feature to allow linking typical business processes to the flow of events coming from heterogeneous and evolving systems. Further, correlation rules can provide better usability to a more extensive range of clients.

Finally, the correlation approach needs to be scalable as the performance of event correlation can be a critical success factor of the business application.

In this work a key research issue is to investigate how a conventional rule engine can be used together with a tool for correlating events to provide scalable semantic event correlation for business process applications.

First, Naiad Rules, a general concept for rule based, semantic event correlation will be developed. The main goals of this concept are interchangeability of the underlying rule engine, the support of a GUI based configuration of rules, a maximum of scalability and the possibility to distribute rule processing to multiple systems at any time. Occurred problems and their solutions to achieve these goals will be discussed in detail.

It will then be shown how common patterns of complex event processing can be implemented with that concept. Limitations of it will be discussed and useful future enhancements to get rid of those limitations will be contemplated.

The design of the open source tool Naiad, a tool for correlating events, will be extended with standardized programming interfaces and configuration files to accommodate a rule engine that is compatible with the staged event-driven architecture (SEDA) paradigm of Naiad.

The design will be prototypically implemented as part of the open source project Naiad using the freely available rule engine Drools. The integration with the already existing parts of Naiad and some major problems with it will be discussed in detail. Then bottlenecks and possibilities to their avoidance will be presented.

As an accompanying example the SAW (Simulation of an Assembly Workshop) project with a simulator and hardware test bed at the Vienna UT will be used. Finally findings about performance and the ease of configurability will be presented, before a short outlook will give an overview about a possible roadmap for future research and development on rule based event processing with Naiad.

Contents

List of Figures

Listings

1 Introduction

1.1 Events and their Relevancy

1.1.1 Definition of Events

The notion of an event is common in many fields of computer science and even in other sciences as well. For instance in software development, events as means to invoke certain portions of code (mostly used in programming of GUIs) are just one very well known example. Therefore the meaning of this term can differ as much as the occasions where it is used at.

In his book *The Power of Events* [12] David Luckham provided the fundamentals for what today is commonly known as Complex Event Processing (CEP) and for this work in particular. His 2001 definition of an event still describes the understanding of events in this work pretty well:

> An event is an object that is a record of an activity in a system. The event signifies the activity. An event may be related to other events [12].

Though there is nothing wrong with this definition the term *system* might misleadingly suggest that events can only be found (and processed) in *electronic* systems. And indeed it is true, that most research into CEP has been done investigating purely electronic systems.

However, more recently Mühl, Fiege and Pietzuch used a slightly different terminology on that topic, and thus accentuate the otherwise easily overlooked fact that events do not equal the data-container that represents them, when they are processed. To them the term *event* stands for the *happening of interest* that is then *observed from within a computer*. In contrast to that they define a *notification* to be *a datum that reifies an event* [16], for example the very instance of a Java [31] class containing data about a real-life event. In Luckham's work, the latter is called the *significance* of an event.

For the rest of this work the term textitevent will be used ambiguously. The context should always make it clear, whether the event itself or the notification representing it is meant.

Regardless of the terminology used, understanding the difference between those two ideas (event vs. notification or significance vs. event) is the key to understanding the full potential of CEP.

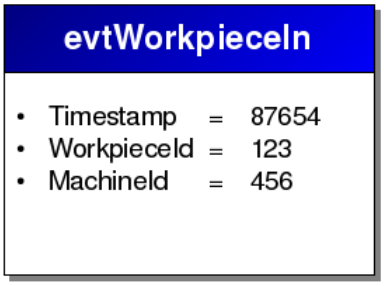

Figure 1.1: An event of type evtWorkpieceIn

1.1.2 Importance of Events

Whenever a computer or similar device generates a notification about any internal happening, it is obvious that this information can then be used for further processing. CEP techniques can be applied to the data and the beneficial effects of those patterns can be leveraged.

Now what if this device would generate a notification not about an internal but an *external* happening? The device would then become an *observer*. This could be any simple sensor like a light barrier or something much more complex - just as long as an electronic notification containing data about the happening can be generated. *Data* in this context can be as little as just the information about the plain occurrence of the event [16].

Hence all the known CEP techniques can be applied to any events as long as they are (electronically) observable. With today's possibilities of measuring technology this includes pretty much all imaginable domains and systems in the world around us, on any desired level of detail.

In purely electronic systems, which still are the main focus of ongoing research into CEP, it is even easier than that: most such systems already include elaborate notification and reporting features that *can be utilized for monitoring and analysis purposes*. Unfortunately *today's platforms suffer from inappropriate management* [26] of this data.

1.1.3 Layers of Interest

An event for instace could be the fact that a workpiece is entering a machine (see chapter 1.5 for the accompanying case-study). In a minimal version it could look like figure 1.1.

Obviously for every evtWorkpieceIn there should be an event of type evtWorkpiece-Out, like the one in figure 1.2.

As one can easily see, the bigger our assembly line gets, the more events we would end up with. This can very quickly exceed any manageable dimension. Besides, we

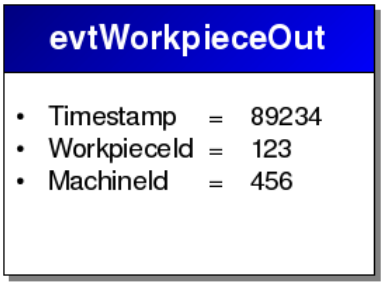

Figure 1.2: An event of type evtWorkpieceOut

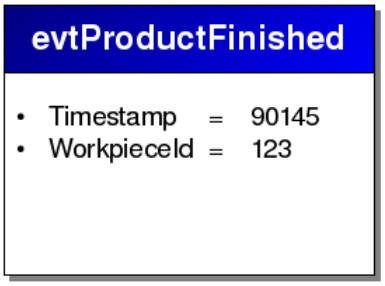

Figure 1.3: An event of type evtProductFinished

might not be interested in workpieces entering and leaving machines at all. Probably we only want to know about finished products leaving our factory. Like the event shown in figure 1.3.

Suppose that unfortunately in our assembly line there is no sensor which could generate such an event. Luckily, knowing the sequence of machines a workpiece has to move through to become the final product, we can derive an evtProductFinished event from all evtWorkpieceIn and evtWorkpieceOut events for a certain workpiece. And thus, we have just created a complex event.

1.2 Complex Event Processing

1.2.1 From Complex to Correlated and Composite Events

To understand what the processing of complex events is all about, one naturally has to have an idea what a complex event is in the first place.

> What is a complex event? It is an event that could only happen if lots of other events happened [12].

This very simple but comprehensible definition describes exactly what the evtProductFinished event from the previous chapter represents. It can only happen, if a workpiece has passed all required machines, that is to say all required evtWorkpieceIn and evtWorkpieceOut events have happened.

Now, knowing the assembly line and the product it is relatively simple to see which evtWorkpieceIn and evtWorkpieceOut events must have happened once a evtProductFinished occurs. But as we said before: evtProductFinished events don't occur on their own. So it is one of the key challenges posed on complex event processing to find out things the other way round. *How* do we find out, that a product was finished?

An additional level of complexity is added to that problem by the fact, that there might be not just one type of product that our assembly line is able to produce. So we might face a huge amount of events of which there is only a fraction we are interested in. Facing a bigger scale, say the events send back and forth between the electronic systems of two or even more globally operating enterprises this is what Luckham describes as the *Global Event Cloud*.

> We talk of a cloud rather than a stream because the event traffic is not ... nicely organized. ... They (events) do not necessarily arrive ... in the order they were created or in their causal order [12].

So the first step in CEP is to collect all the events from the event cloud that are of interest to the question we're currently facing. This is called *correlating* the events. Based on information specific to the current problem we have to find groups of events that represent "regions if interest" within the event cloud. In our example we might want to correlate (= put into separate groups) "all events containing the same workpieceId in their attributes". Occasionally the acronym CEP stands for Correlated Event Processing. However, as we will see, this is only one part of the big picture.

To add to confusion, Mühl, Fiege and Pietzuch introduce the notion of a composite event:

> A composite event ins published whenever a certain pattern of events occurs in the ... system [16].

In our example the composite event might be the evtProductFinished event whereas the pattern that caused the composite event to be published is the set of all evtWorkpieceIn and evtWorkpieceOut events required to finish the product. Depending on the particular definition of composite events and/or the system used, a composite event might contain all its causing events, references to its causing events, all or parts of the data contained in its causing events or none of that at all.

To sum things up we can say that the correlation and the composition of events are parts but not all of the techniques that are commonly referred to as complex event processing. One part does not necessarily require the other, but the benefits of applying both are obvious.

David Luckham comes to a similar conclusion:

> CEP consists of very simple techniques, a mix of old an new. ... In CEP, new techniques are combined with well-known techniques in a unified framework [12].

Additionally it is worth mentioning that the most important part of CEP, besides the correlation and composition of events, is the *presentation* of the data that was

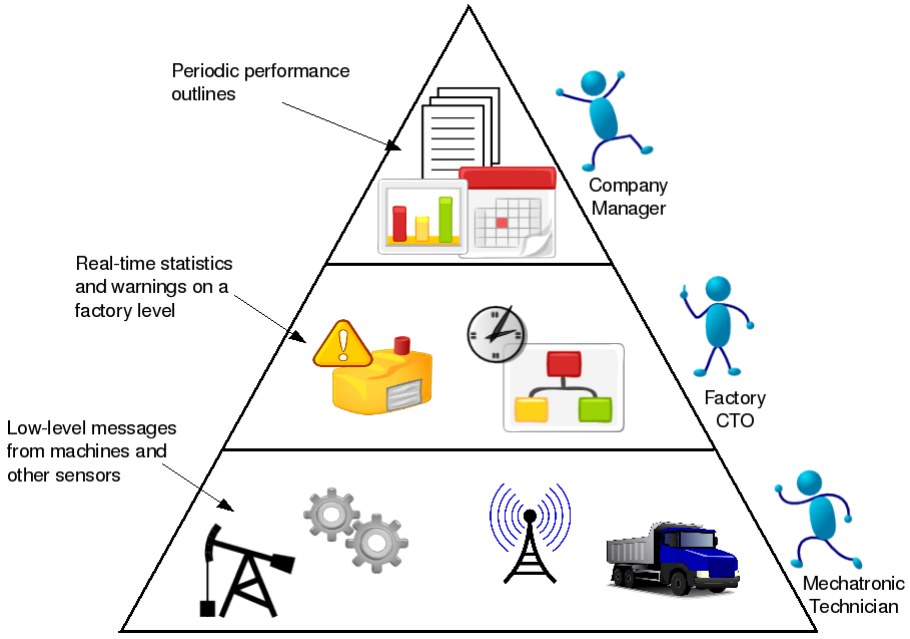

Figure 1.4: Different roles in a company require different views on its events.

processed. Very high level composite events are of no use, if they don't aid any decision-maker in managing or troubleshooting the system at hand. This however is out of focus of this work.

1.2.2 Advantages of Complex Event Processing

As already mentioned, the biggest advantage of complex event processing is that the very same principles and techniques can be applied to any system that can be recognized as a crowd of generators of events.

Second, and we will see this later in this work, a CEP system is not one big chunk of code, but consists of many relatively simple pieces that can be plugged into each other as needed. Therefore, for simple tasks it is not necessary to implement and configure a complicated system. A CEP system can easily adapt its own size and complexity to the tasks assigned to it.

The concept of composite events enables the abstraction of the underlying system on any desired level of operation. Further, for different roles in an enterprise or other system the generation of different views on the system is possbile, combining multiple correlations and compositions of events (figure 1.4). This helps to separate concerns between agents with minimal extra cost, because all the processing is starting from the same basic set of events, and thus from the same set of - possibly expensive - observers.

Moreover the loosely coupled nature of a CEP system makes it extremely flexible when it comes to adapting to changes in the underlying system or creating new views onto it. In theory all of this even is possible on the fly, while the observed system *and* the CEP system are running.

1.2.3 What is Complex Event Processing not?

Though apparently sharing the same intentions, CEP systems differ strongly from any systems implementing an ETL (Extract, Transform, Load) [10] approach like traditional datawarehousing software. Both try to understand and manage a given system by analyzing relatively small pieces of information and processing them into a bigger picture. The important difference is, that ETL systems usually operate in a batch-mode during a given time-window while CEP systems process events from a continuous stream of data in or near real-time. This favors CEP systems over the ETL approach whereever latency is critical [26].

Another approach to low-latency event processing is event stream processing (ESP). ESP systems work on a stream of events, that is a sequence of events ordered by time. Opposed to that is an event cloud, a set of events without any explicit ordering in time [13]. When looking at an event cloud, more focus has to be given on the relations between those events. Time is just one if them. Luckham identifies *the three most common and important relationsships between events* as the following [12]:

Time states that one event happened before another one.

Cause states that one event had to happen in order for the another one to happen.

Aggregation one higher level event that consists of a number of lower level events. Luckham calls this a *complex event* [12], Mühl, Fiege and Pietzuch a *composite event* [16].

Typically ESP systems cannot handle too elaborate relationships between events, as they cannot recognize cause and aggregation, and time only to a limited degree. Of course this is not a disadvantage per se, but just another focus. On the other hand, manufacturers of ESP applications have recently started to include more and more features of CEP into their products, which is gradually closing the gap between ESP and CEP applications [13].

1.3 BRM Systems and Expectations for Complex Event Processing

Business Rule Management Systems (BRMS) have been around for quite some time now. Their main purpose and conventional scope is to separate business logic, that is policies that have to be enforced in certain business situations, from processes and the software systems that manage them. They further try to support automatic decision

making and thus aim to increase a business' agility when reacting to internal and external chances, opportunities and threats.

BRM Systems express business policies through (mostly large) sets of relatively simple rules in an *if/then* format. Most such systems come with their own language and syntax to define Business Rules, and many of them even allow the definition and usage of Domain Specific Languages (DSL). This significantly lowers the cost and shortens the delay of handling Business Rules, as non-technical but managing personnel can create and modify them.

The core part of any BRM System is a *Business Rule Engine*. A specialized piece of software, developed to survey **huge amounts of information**, transforming forementioned **large sets of rules** to machine-oriented instructions and applying them to the information they are fed with. Therefore **Pattern Matching**, deciding whether the *if*-part of a rule is satisfied, is obviously one very important task of a Business Rule Engine.

These requirements and aims of Business Rule Engines on the other hand are quite similar, if not identical for the most part, to those of a CEP system. In the latter we need to process huge amounts of information as well, we need to apply rules to that information[1], and we need this done as fast as possible.

Hence the basic idea of this work, to utilize an already existing, conventional Business Rule Engine to implement those features that are needed in CEP systems. Expectations are, that the following benefits of existing rule engines can be leveraged for complex event processing as well:

- Using an existing engine and its libraries significantly reduces **development time and cost**.
- Rules are **relatively simple** and thus can be understood and managed by less trained personnel than complex relations of events. The use of a Domain Specific Language might be able to further utilize that effect.
- A rule engine's **high-performance algorithm for pattern matching** is already optimized for the task of finding certain constellations in huge clouds of events.

1.4 Subject and Motivation of this Work

This work investigates if and how conventional rule engines can be used to support the implementation of a CEP application. The suggested patterns will then be tested on an existing CEP application (Naiad, see chapter 3) applied to a real-life environment (SAW, see section 1.5).

[1] How to filter events, how to aggregate them, what to look for in the first place.

Naiad is an open source, highly flexible and easily extensible event processing server that provides automatic correlation of events. As we found out in the previous chapter, event correlation is just one part of complex event processing. To be a full featured CEP application, Naiad still lacks the composition of events, the automatic generation of alerts and metrics, the graphical presentation of the processed data and other general principles of CEP.

First and foremost this work is going to show how a rule engine can be used to implement the composition (mapping) of events. That is how to aggregate multiple low-level events into less high-level events by detecting patterns in pre-correlated clouds of low-level events. In this document this extension will be referred to as *Naiad Rules*.

Furthermore we will see, how the relatively simple features of pattern detection and event aggregation can be adopted to implement a couple of more sophisticated concepts of CEP (see chapter 5).

The utilization of rule engines in the field of complex event processing is not a totally new idea. For instance in [19], quite some time before the notion of CEP gained popularity though Luckham's book [12], the Alarm Correlation Engine's (ACE) goal was to improve (automatic) network management. It was using a specific high-level language that supported a condition/action like process, which had great similarity to conventional rules. Without being denominated like this, the data containers describing those actions could already be seen as complex events.

Other works where conventional rule engines are mentioned to be used for CEP include papers of Josef Schiefer, especially [28].

1.4.1 The Sense and Respond Paradigm

In their work Schiefer and Seufert described SARI (Sense and Respond Infrastructure) [28]:

> ... SARI ... manages the processing of past-oriented, present-oriented and future-oriented data in order to support business processes with Business Intelligence in near real-time.

SARI works in continuous loops, sensing business information, interpreting and analyzing that information, making decisions upon that information and sending those decisions back into the system (responding), as shown in figure 1.5 Sensing in this respect means gathering information in a non-intrusive way, that is with the monitored system not even knowing.

Besides providing low-latency results, SARI aims to be a *distributed, scalable platform* that follows the paradigm of Service Oriented Architectures (SOA) [18]. SARI's architecture is shown in figure 1.6.

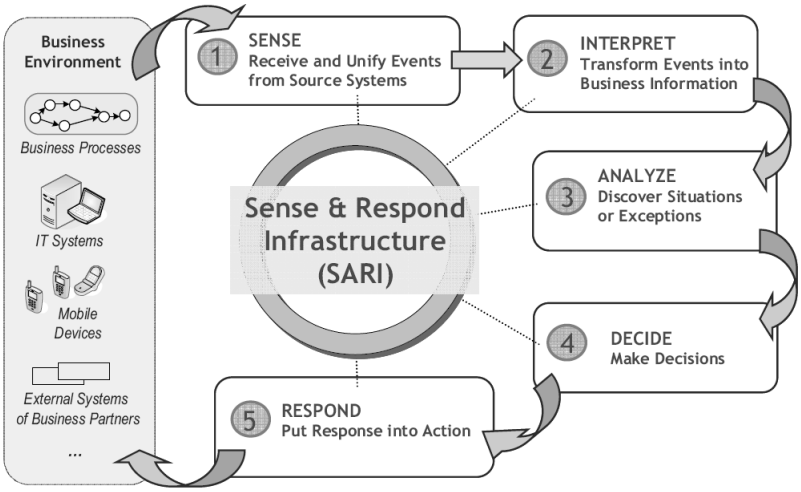

Figure 1.5: SARI Sense & Respond Loops [28]

Naiad so far has followed the SARI principles in many aspects, which is why any rule engine based extension should stick to them as well. Naiad Rules will follow the paradigm of SOA, enabling it to be hooked into Naiad's event bus without affecting any other existing or yet to-be components (unless this is wanted). The theoretical foundation for this has been presented in [27] and carried on by Rozsnyai in [23], a detailed presentation of which can be found in section 2.4.

1.4.2 Requirements to a Rule Engine based CEP Application

Numerous requirements to a CEP application have already identified in works before this one. The most fundamental of which naturally deal with the features and techniques of complex event processing itself. Among others, a CEP application must

- ... be able to **subscribe to a system's events** (or get hold of them another way).
- ... foremost *provide techniques for defining and utilizing relationships between events* [12], in other words provide means to **implement common patterns of CEP**, such as composite events, filters, constraints, alarms, etc.
- ... be **understandable and manageable** by experts on the domain at hand, not software engineers. Customization should only consist of configuration, not the development of completely new software components.
- ... support the **graphical presentation** of the processed data, enabling quick and unerring judgments by a business' decision makers.
- ... support *real-time, drill-down, event-base* ***diagnostics*** [12].

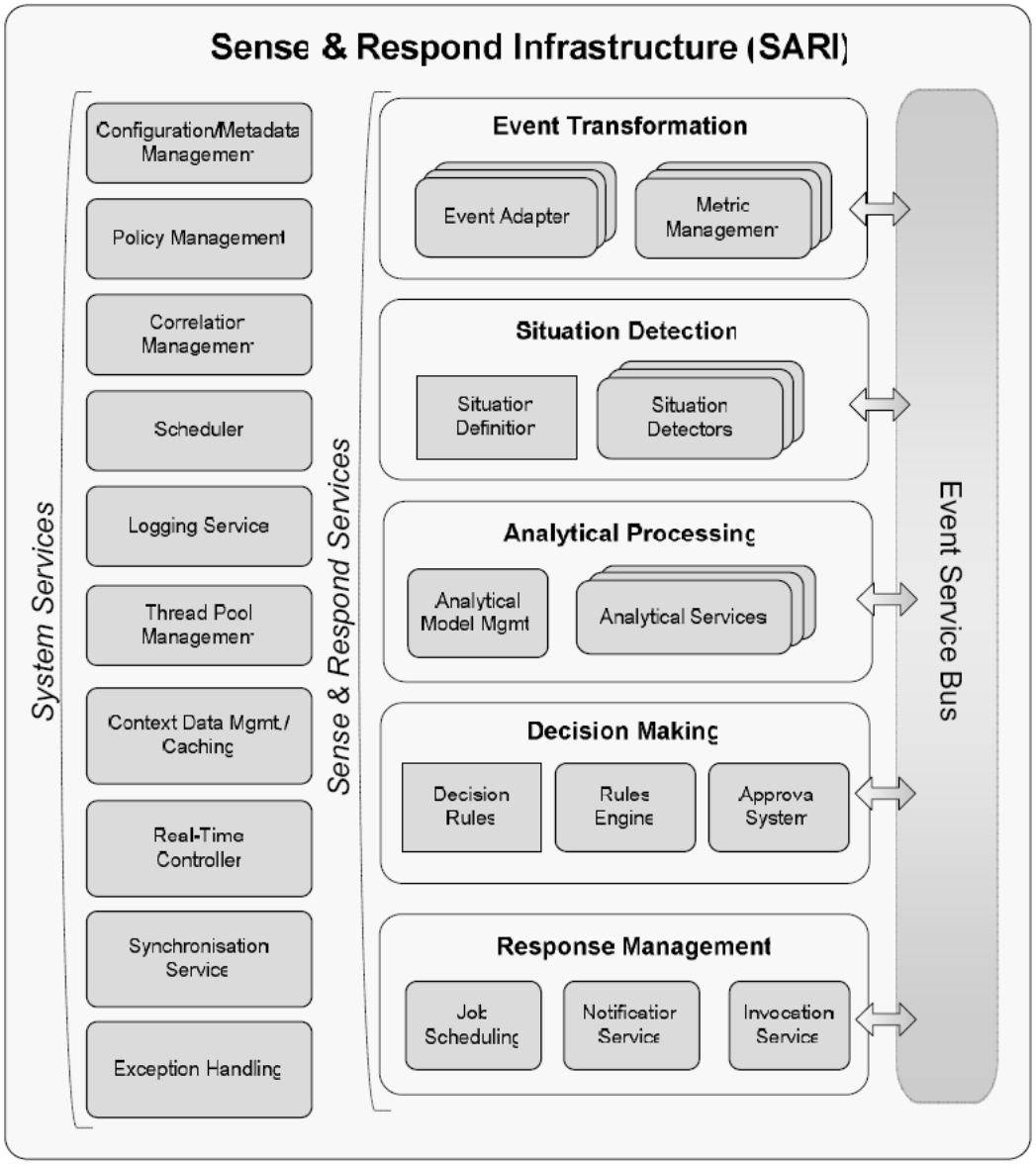

Figure 1.6: SARI Architecture [28]

However there are a number of "meta-requirements" as well, that do not directly contribute to the set of features of a CEP system, but rather make its application in today's business environments feasible and useful.

- Should enable **easy integration** into existing systems [28]. In a best case scenario the complex event processing is just hooked into the monitored system, without it being affected in any way. This, of course, only applies to the Sense part of SARI. Today's enterprise service bus (ESB) systems make this relatively easy.

- Under the assumption that many small interconnected computers are generally cheaper than a few powerful systems, a CEP application should be **distributable** down to almost atomic EPAs (event processing agents, see [12]).

- Not only because of the previous item, a CEP solution should be **scaleable** while continuously integration data with **minimal latency** [26].

- Following the SARI paradigm, it should be **extensible** without affecting already existing parts of the system.

- **On the fly configuration** should be possible, as *the specification of the events of interest, how they should be viewed and acted upon, can be changed while the system is running* [12].

For the part of rule processing in a SARI environment (called Sense and Respond Rules) additional requirements have been identified and theoretically also met in [27]:

Event-triggered Rule Evaluation To support near real-time decisions, rules shall be evaluated (and triggered) as soon as a new event relevant to them comes in.

User-friendly Rule Modeling and Adaptability Same as for the general requirements, even technically less experienced personnel should be able to create and modify rules. Sense and Respond Rules suggest a tree like structure (see section 2.4) that very much promotes the use of graphical editors.

Building Complex Rules with Divide and Conquer The aforementioned tree like structure supports the breakdown of rules to individually quite simple rules that can then be combined to reflect more complex business situations.

Event Pattern Recognition The core part of a rules left-hand-side in most cases is comprised of the recognition of certain patterns within a cloud of events.

Service-oriented Rule Processing has already been discussed in this section.

1.4.3 Anticipated Benefits

Using a conventional, out of the box, third-party rule engine obviously comes at the cost of decreased overall performance and less freedom when designing a CEP system. The big advantage, which this work tries to depict, is, that this approach implies a much shorter development time than any solution that is build from the scratch. This still is easily comprehensible, as all the pattern matching does not need to be programmed and only interfaces for event input and output have to be developed and "plugged into" the rule engine. It is not quite as easy as that (as we will see in section 4.2), but the general idea holds true.

Now what is harder to believe is, that using an out of the box product still leaves the freedom to implement at least the most common patterns of complex event processing. These are filters, constraints and maps [12]. In chapter 5 it will be shown that this is possible in most if not any desired situations, and in addition present approaches for more advanced patterns like EPNs [12], Cut and Join or Distributed Event Detection [16].

Though most papers that provided the background for this work only handled one single or very few domains (Business Processes [26] [12], computer networks [19], ...), the presented approach will be flexible enough to be applicable to any system that satisfies the requirements defined in section 1.1.2.

1.4.4 Delimitations

The primary goal is not to show how Business Rules can be implemented and carried out in a CEP environment. Though this would be totally possible with the presented approach, it is merely a convenient side product of the work and hand. The way rules are used here goes beyond plain decision making: they are used to implement the very patterns of CEP themselves.

Again it should be mentioned, that the rule engine will not be used to find correlations between incoming events, though this of course would be possible. Following the separation of correlation and further processing and thus promoting a separation of concerns, as it was suggested by Schiefer and McGregor in [26], our rule engine will only operate on pre-correlated sessions of events. Benefits of this approach are discussed in section 2.3.

1.5 SAW: Simulation of Assembly Workshops

The SAW project of the Vienna TU's IFS institute studies agent-based solutions in an assembly workshop's environment. "Agent-based" means that there is no central control element, but all parts of the system, like working cells (machines), conveyor belts and others, communicate with each other and try to implement an optimal strategy in terms of total efficiency. This, of course, is a quite interpretable notion, hence the notion of an *optimal strategy* is very wide. Evaluating different strategies is just one of many subtopics of the SAW project. Failure detection and handling

is another one. An advantage of agent-based systems over centralized approaches is *the ability to react dynamically and appropriately to any structural/functional changes within the agent-based system* [37].

Heart of the project is a Java based simulator of such assembly workshops. It originated from the MAST (Manufacturing Agent Simulation Tool) simulator, which was presented in [36]. Further improvements have been made to the Tool, resulting in a system of four parts as described in [37]:

Agent Control Part This part represents the agents which make up the whole system. The JADE Agent Platform[2] is used for that.

Emulation Part In the end, the software agents are supposed to control real-world machinery. For development and testing purposes, sensor-signals from those machines have to be emulated, which is this part's task.

Runtime Interface Ideally no programming should be necessary when switching from an emulated to a real assembly workshop. This unified interface provides the basis for that. It even facilitates a stepwise "going-live", with parts of the machines emulated, parts being the real deal.

Graphical User Interface A GUI where layouts of workshops can be created and modified per drag-and-drop interaction. While a simulation is running, its progress can be observed on those layouts.

The SAW team at the IFS contributed some more improvements. Figure 1.7 shows the current state of the simulator's GUI with a sample workshop loaded.

The relevance of the SAW project for CEP and its adequacy as a showcase domain for Naiad is obvious: any piece of communication between any agents or machines can be seen as an event. Some examples of such events have already been given in the introduction, many more will follow.

[2] `http://jade.tilab.com`

Figure 1.7: Screenshot of the SAW Simulator

2 Related Work

2.1 A short History of Naiad

The application now known as Naiad is the newest product in a line of event handling systems developed at the Vienna UT's IFS [35].

The Project started out by the name of EventCloud and was introduced by Szabolcs Rozsnyai in [22]. It fetched events from Senactive's InTime [29] for further processing. Its core feature was not a real-time analysis of events, but a web based search interface, where events that have already been correlated and stored in a database could be queried with a *"Google-like" search experience*. The three main functions of EventCloud were [22]:

- Extracting and transforming event data from the source system and integrate them into EventCloud's own data structure.
- Full text index over simple and correlated events.
- Search functionality including a sophisticated query syntax and various filter functions.

Technologies that were utilized included Java [31], Lucene [2], PostgreSQL [20] and Spring [30].

EventCloud supported two kinds of searching. "Rank 1" searches was just a simple full text search over all indexed events (and their attributes). "Rank 2", the core feature of EventCloud, was a search over correlations of events. So, if any event within a correlation matched the query, the correlation was added to the list of results, like shown in figure 2.1.

Rozsnyai discussed an unimplemented "Rank 3" as well. This would search not only between events in the same correlation, but even take relationships between correlations into account, and thus linking and finding indirectly related events.

The ETL (Extract Transform Load) approach of EventCloud unfortunately was exceedingly static, which made it a good proof-of-concept starting point and lesson for further research, but a dead end for developing a real-time CEP application.

EventCloud then got renamed to EventServer when Roland Vecera presented a couple of important changes in [34]:

Figure 2.1: “Rank 2” search results in EventCloud [22]

- Correlation, was discovered as a core feature, and was implemented as a reusable service.
- Rank 3 indexing, as proposed in [22], was implemented.
- Understanding that plain searching for events did not satisfy given needs, a pattern for the calculation of metrics was devised.
- The whole architecture of the application was changed into a very flexible service based one.

The new service based architecture was still built upon the Spring framework. Services that have been implemented at that time were a correlation service (as the core feature), a service for persisting events to a database, services for creating rank 1 to 3 indexes, a service providing simple timing metrics and a service for performance logging. Besides, Event Server was prepared to fetch events from other sources than Senactive’s InTime by implementing an EventAdapter Interface (see figure 2.2).

Due to its service oriented infrastructure, the EventServer was much more reusable and applicable in a wider variety of situations than EventCloud. However, the services it included still promoted a quite static use of the whole application (indexing and searching at a later point in time).

Event Core later got the Name IFS:CEP, under which, still focusing on the use of open source products, it was ported from Spring to Mule [17] as the underlying framework. Mule provides a huge amount of connectors to existing ESB (Enterprise Service Bus) solutions and other common protocols out of the box. Further does it support the easy (re)distribution of service agents (Mule UMOs), that can form queue- and tree-like routes for incoming events. Thus Mule enables scalable solutions in heterogeneous software environments.

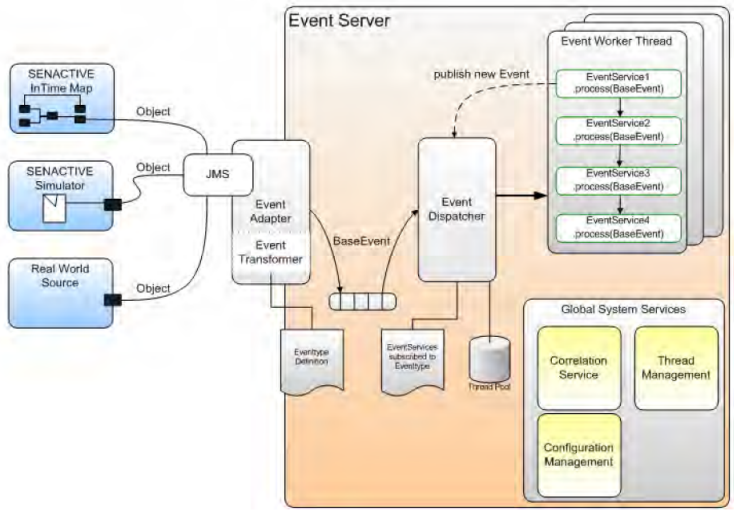

Figure 2.2: EventServer architecture overview [34]

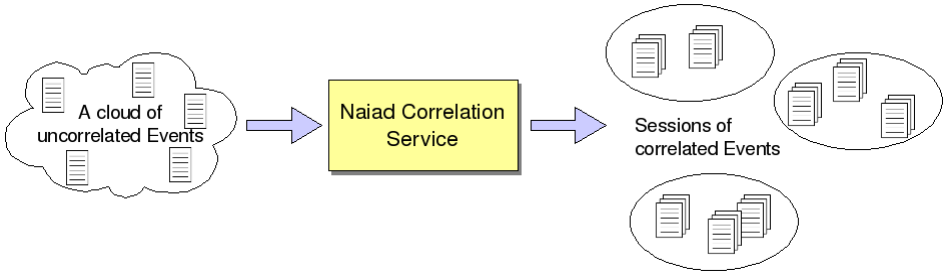

Figure 2.3: Naiad's function seen as a blackbox

The main idea in IFS:CEP remained the same as presented in [26] and followed by [34]: incoming events are first correlated into *correlation sessions* which then can be accessed by other downstream services. This provides a maximum amount of modularity and extensibility.

> A correlation session is a container with a set of data items that exist for each relationship between events [26].

While [26] was mainly thinking of metric calculating services, this of course can be any components that benefit from correlation-enriched events in sessions. This feature, consisting of a CorrelationManager and a SessionManager working closely with each other, as the basic enabler for any other features became part of IFS:CEP's core.

Marian Schedenig [25] created appropriate interfaces for these core functions and implemented a CorrelationManager supporting Rank 2 and 3 correlations as well as a first memory based SessionManager. With his work completed IFS:CEP was renamed

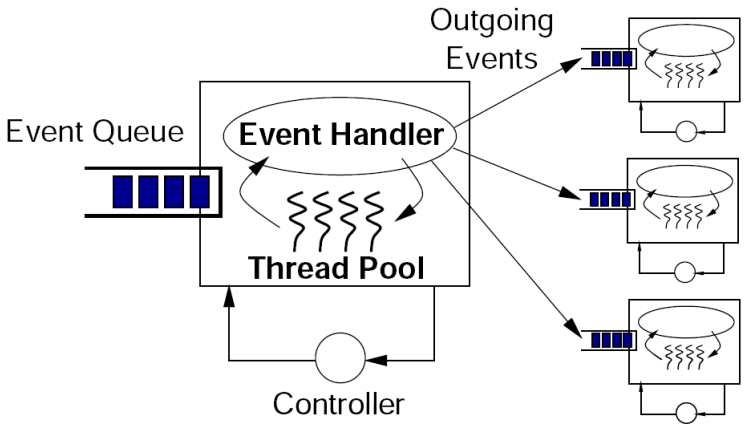

Figure 2.4: A SEDA stage [38]

to Naiad[1].

2.2 The SEDA Model

As an application to be deployed within the Mule framework, Naiad follows the principles of a Staged Event-Driven Architecture (SEDA).

> SEDA decomposes an application into a network of *stages* separated by *event queues* and introduces the notion of *dynamic resource controllers* to allow applications to adjust dynamically to changing load [38].

This approach satisfied the four primary goals for SEDA identified by Welsh, Culler and Brewer:

- Support massive concurrency
- Simplify the construction of well-conditioned services
- Enable introspection
- Support self-tuning resource management

In detail, each stage consists not only of its *event handler*, that is the code doing all the work, but also of a *queue* for incoming events, a *thread pool* and a *controller* as shown in figure 2.4.

The last thee components are superimposed onto the event handler, without it even knowing it. This helps to keep the actual working code simple. The controller is in charge of monitoring the size of the queue and the mean time events take to be processed. Based on those values i can then in- or decrease the number of threads its

[1] In Greek mythology Naiads were nymphs living near sources of fresh water.

event handler is replicated in. These numbers can also be logged and used for debugging and performance analysis of services, which has *traditionally been a challenge for complex multithreaded servers* [38].

Originally SEDA was designed to be used in internet applications, where huge amounts of requests have to be processed in relatively short time windows and with a minimum latency. Therefore, the first presented application was Haboob, a high-performance HTTP Server. However, seen from a software developers point of view, huge amounts of incoming events are not too different from HTTP requests. This made SEDA a very good architecture to choose for Naiad as well.

Mule [17] supports SEDA like, self-tuning resource management out of the box. The event handlers of each stage are called Universal Message Objects (UMO), are almost nothing more than plain old java objects (POJO), and therefore very easy to create, understand and maintain. Just like suggested in SEDA, Mule takes care of all queue handling and thread replication. In Naiad, each service is implemented with its own UMO.

There is of course a downside to automatic resource management, that should be mentioned here as well: Because UMOs don't know anything about the count or state of their twins and processing time of one event per UMO is virtually random, events running through the network of stages might possibly overtake each other. Therefore extra care must be taken, to avoid race-conditions and related situations. More about that subject is discussed by Schedenig [25].

2.3 Separation of Correlation and Rule Processing

The basic idea how the correlation and rule service work together in Naiad was described in [26] and later [34]. The core concept there is, that the correlation of related events takes place in a central correlation service which assigns incoming events to sessions prior to any further processing (like rule based processing). Figure 3.1 illustrates that concept.

This approach of separation of concerns [21] leads to a much clearer arrangement of services and thus to shorter and less error-prone development cycles. Another benefit is the higher level of flexibility to distribute an application or even to redistribute it on the fly. Distributability is closely linked to scalability and thus overall speed of an application.

Resulting from that concept of separation is, that any set of rules is only operating on the events of one single correlation session - the session becomes the *scope* for a ruleset. A rule can never be evaluated for two or more events that do not share the same correlation session. If a situation arises where this would be a desired behavior, the reason for that lies within the configuration of the correlation service, as it does not reflect the needed views on the domain of events. In that case other criteria for correlating the events have to be found, that make sure that related events really end up in the same sessions.

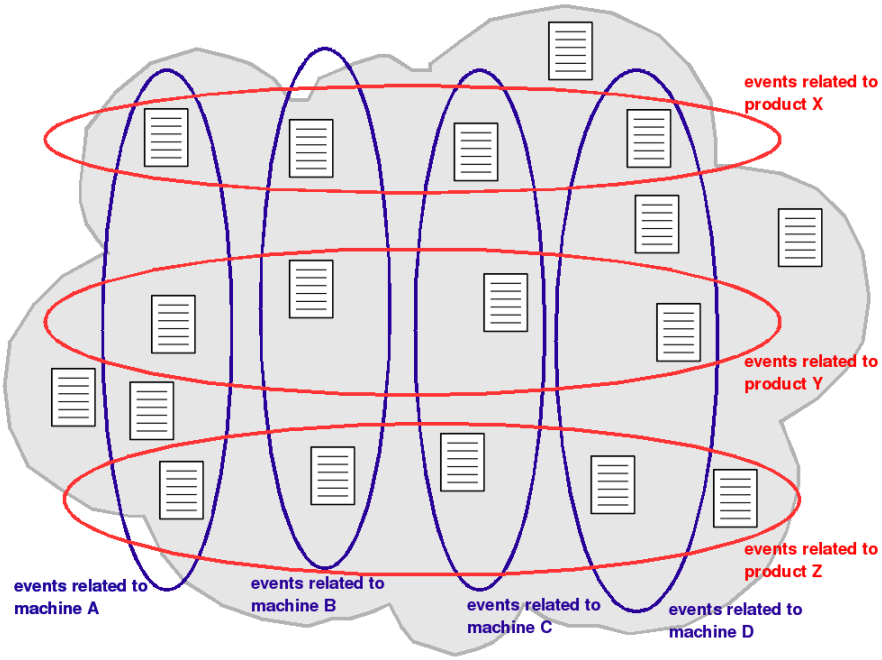

Figure 2.5: Overlapping Sessions in an event cloud

Sessions overlapping for certain events are no problem, because every event can be a member of multiple sessions at the same time. If it makes no sense to extend a session's correlation criteria to include some new aspect of the domain, the best thing to do is to just create a completely new correlation session for that aspect. A correlation session should always reflect one area of interest, or *view*, inside a target domain. This helps to keep rules (or any other extending service) simple and maintains the scalability of the application.

Figure 2.5 shows different views within the domain of the SAW.

2.4 SARI Rules

SARI Rules, as they were suggested in [27] and adapted in [23] form the basis of how the rule service for Naiad should work and how it should be configured by a domain engineer. In the original paper SARI Rules tried to address six key requirements: Event-triggered Rule Evaluation, User-friendly Rule Modeling, Building Complex Rules with Divide and Conquer, Event Pattern Recognition, Adaptability and Service-oriented rule processing. They have already been discussed in section 1.4.2.

> Sense and respond rules are organized in rule sets and allow to construct decision scenarios, which use *event conditions* and *event patterns*

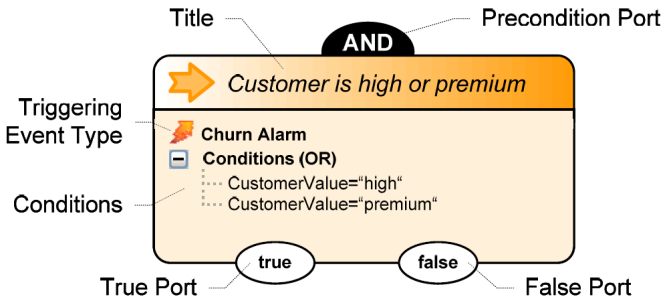

Figure 2.6: Event Condition in SARI Rules [27]

> for triggering *response events*. Event conditions and event patterns can be arbitrarily combined with logical operators [27]...

Event conditions (figure 2.6) are triggered, when an incoming event matches the event type defined for that condition and satisfies a list of boolean expressions contained in it. Every event condition possesses a *true port* and a *false port*, either of which may be connected to another element of the rule set. This might be another event condition, an event pattern or a response event. The true port is activated, when the event condition evaluates to true, the false port in the opposite case. Optionally every event condition may posses a *precondition port*, where conditions or patterns can be connected, which should be used as a precondition for the current event condition. Precondition ports can be of type AND or OR.

Event patterns (figure 2.7) can detect series of events. Opposed to event conditions they require some kind of state information maintained within a session. Besides a definition of the event types that can trigger the pattern it contains a *pattern definition* that defines some boolean matching criteria for those events and how they should be causally related. Same as event conditions, event patterns may posses a precondition port. However, instead of a true and a false port they only feature a matched port, which is activated when incoming events have satisfied the pattern definition.

Response Events (figure 2.8) respresent events that should be created once their precondition port is activated. The attribute of such events can either be constants, bindings to attributes of triggering events or calculation expressions containing any of the previous.

An example of a couple of these elements being connected together to form a rule set can be found in figure 2.9.

In [23] Rozsnyai identifies the key benefits of SARI Rules:

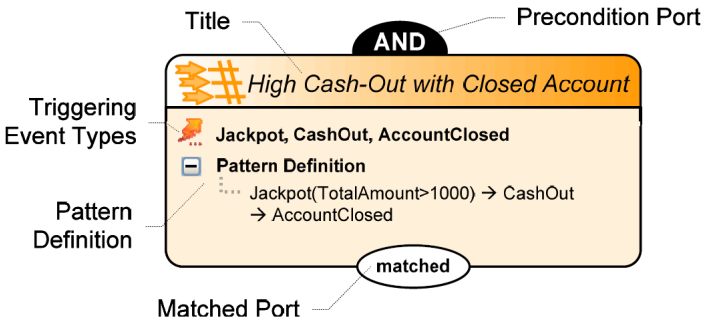

Figure 2.7: Event Pattern in SARI Rules [27]

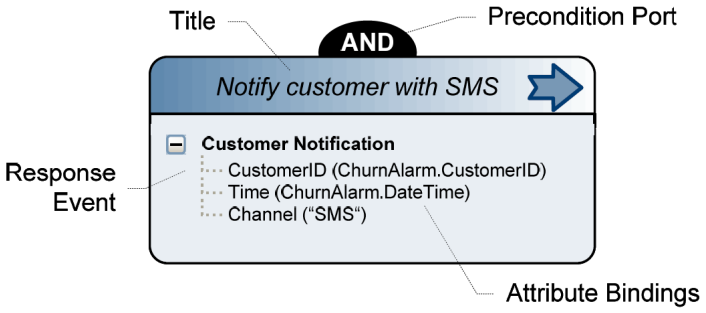

Figure 2.8: Response Event in SARI Rules [27]

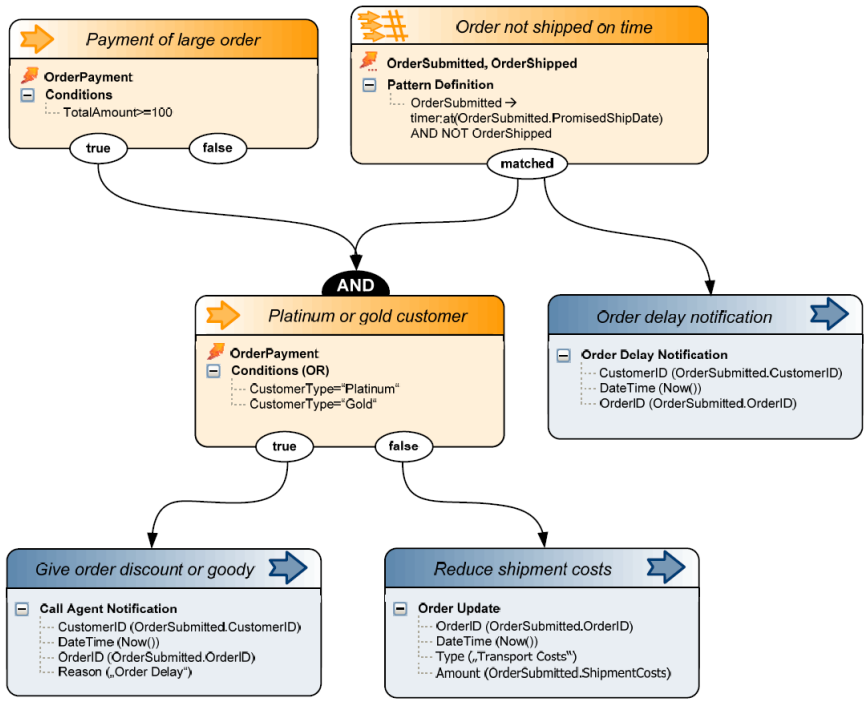

Figure 2.9: An example of interconnected elements of SARI Rules [27]

- The graphical model promotes the use of a graphical editor and thus dramatically increases the range of potential users.
- Preconditions can extend the scope of event conditions and help to keep them simple at the same time.
- False ports *facilitate the implementation of "otherwise" situations.* Event conditions can be reused instead of creating almost identical ones. This further helps to keep rule sets simple.
- Even more simplicity derives from the separation from the correlation process.
- The service orientation of SARI Rules helps to maintain flexibility for the rule processing itself as well as for any relying services.

The big steps taken towards the upkeep of simplicity and performance naturally don't come without a downside. SARI Rules certainly do not provide all the possibilities a full featured language like RAPIDE [11] would feature. However, one aim of this work is to show, that the concept of SARI Rules is still powerful enough to accommodate the biggest part of all common business situations and even situations beyond that.

2.5 Similar Concepts and Projects

Many concepts that are applied in Naiad and Naiad Rules are not totally new - not even their application in the field of Complex Event Processing.

Schiefer and McGregor [26] were not the first to value the power of correlated events. As early as in [19] the Alarm Correlation Engine (ACE) was presented. It features the correlation of causally related alarms and the triggering of rules upon these alarms. In this respect it followed the same principle as the SARI architecture. The ACE however was designed exclusively for telephone networks.

Senactive InTime [29] is another, commercial, product following the SARI paradigm. Like Naiad, it too correlates events into sessions before further processing them. Naiad shares some history with InTime, because in prior lifecycles, before aiming to be a standalone CEP product, Naiad was designed to analyze events that originated from InTime.

The Simple Event Correlator (SEC) by [33] is another tool that utilizes the correlation of events. It aims to be lightweight, open-source and platform independent, as it is written in PERL [32]. Its primary focus lies on network management. SEC is configured with regular expressions, which are then applied to incoming events. Those expressions form rule-like entities, making Vaarandi one of the first to introduce rules to Complex Event Processing. However, those rules where not enforced by a rule engine in our sense.

RuleCore [15] is a commercial real-time CEP server that features dynamic (runtime) management of event condition action (ECA) rules, causality tracking, semantic event correlation and a convenient user interface. In its concepts, Naiad is very similar to RuleCore, as it uses an ECA based rule engine, JBoss Drools [9], too.

Drools CEP [9] is the relatively new approach to extend the Drools Rule Engine with features of complex event processing, and thus trying to implement things “from the other side”. Though very interesting and promising, Drools CEP was not in a usable state at the time this work was done and therefore has not been reviewed further.

Internally Naiad uses XML-based events. One problem when trying to bring this approach and a conventional rule engine together is, that the latter was never meant to work with entities of variable and extendable nature. Chen discusses ways to use legacy CEP applications, that can only handle name-value pairs as attributes to the events processed, with structural event data, such as XML [4]. A similar approach is taken for Naiad Rules too.

Although pursuing virtually the same goals, event stream processing (ESP) is a slightly different approach to the topic then complex event processing. Esper [6], an open-source Java [31] based product, is a quite prevalent tool from that sector. It supports event pattern matching, event correlation based on these patterns and a

SQL-like query language. Esper has been ported to .NET [14] as NEsper. ESP is explained in section 1.2.3 in a little more detail.

Finally, works by Luckham [12] and Mühl, Fiege and Pietzuch [16] should be mentioned, where Patterns of CEP that this work tries to implement (see chapter 5) have been presented.

3 Naiad - an Event Correlation Server

The previous chapter has given an overview about the history and the becoming of Naiad. This chapter will illustrate the current state of Naiad and the features it provides.

3.1 XML based Events

To extract values out of an event's data container Naiad uses XPath [5] Expressions, which have already proved themselves very suitable in [26]. They are powerful enough to be applicable to almost all imaginable real-life data structures for events. Of course, this strongly suggests the use of XML as underlying data structure, because XPath libraries for XML are available as third-party products and appropriate adapters have already been included in Naiad's core. To hook up Naiad to any system that is already providing events in the XML format, one is only left with some configuration tasks and no programming whatsoever. Further, the extensible nature of XML perfectly complements the service oriented architecture of Naiad.

There are of course disadvantages about XML based events. They can be roughly summarized as XML events being very "clumsy" to handle. More on this can be found in [4]. On the other hand Naiad supports the processing of events in any other data structure than XML as well, just as long as data can be extracted using XPath expressions. The only thing that must be done to accomplish this, is to write an adequate implementation of the IEvent interface provided in Naiad's core, which is not too much of an effort.

XPath is not only used for the plain extraction of data from events, but also for defining the attributes after which events should be correlated. Optionally it can be used to define an event's *type*, if that information is available in the underlying XML structure. In the example of listing 3.1 the type's XPath would be `//@type`.

```
1 <event id="68" timestamp="35100" type="evtWorkpieceOut">
2   <payload key="WorkpieceId">medium_part1</payload>
3   <payload key="ComponentName">DS7</payload>
4 </event>
```

Listing 3.1: An event from the SAW domain

Events in Naiad, besides all the attributes they carry within themselves, are tagged with a special attribute, its *type*. When scanning for fitting correlation sessions for an

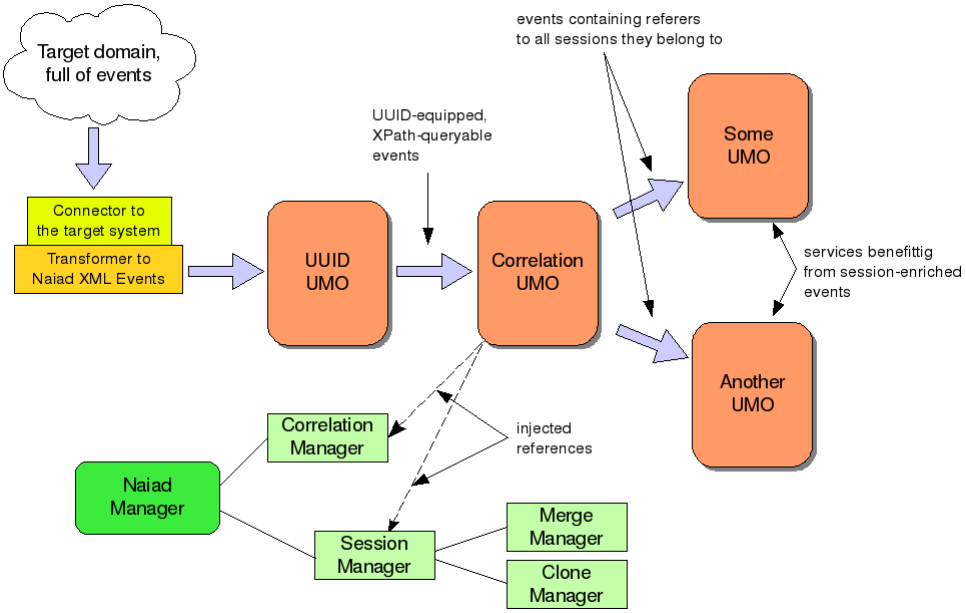

Figure 3.1: An overview of Naiad's architecture

incoming event, the type of an event is used to make a first distinction whether the event is a candidate member for that session or not. Except for that the type of an event has no inherent special meaning in Naiad and doesn't have to be used at all. However, in most occasion it is very convenient to have a mean of rough distinction, before any attributes of an event need to be examined.

3.2 Correlation and Sessionhandling as Core Features

Figure 3.1 shows a basic layout of Naiad's core. First, Naiad taps into any system via one of Mule's various out-of-the-box *connectors*. Ideally this is done via subscribing to any external service's notifications, implementing an event-based interaction model [16], and thus leaving the monitored system completely untouched. Alternatively a custom connector implementing Mules interface for connectors can be written.

Incoming events are then processed by a *transformer*, that puts their data into Java classes implementing the IEvent interface, which knows an events type and can query its data with XPath expressions.

It is possible to use multiple connectors and adapters at the same time, just as long as valid IEvents are the output.

The IEvents then go into a *UuidUMO* that equips them with a unique identifier in the UUID format[1]. The identifier is later used for (de)serialization, but is a convenient

[1] `http://tools.ietf.org/html/rfc4122`

help in bugtracking as well.

Uniquely identified events then go into the *CorrelationUMO*, where all the magic happens. Though, following the principles of SEDA, many instances of the CorrelationUMO may exist at the same time, they share the same, injected[2] references to a global *Correlation Manager* and *Session Manager*. To be precise, they only share a reference to a global *Naiad Manager*, from which they can request references to a Correlation Manager or Session Manager via injected identifiers of those. The Correlation Manager and Session Manager are instantiated by the Naiad Manager from its configuration.

The Correlation Manager first returns all *Correlation Matches* fitting an incoming event (see section 3.3 for more on CorrelationMatches). The list of Correlation Matches is then sent to the Session Manager, which returns a list of (identifiers of) *Sessions* with which the event is enriched.

If needed (which might be the case when using so called *bridged correlations*, see section 3.3) sessions get merged prior to that. Because sessions serve as data containers for the correlations an event belongs to [26], they might contain data that needs special attention on merging. For this reason multiple *Session Merge Managers* may be injected into the Session Manager. Normally a Session Merge Manager would handle the merging of one special date in the session.

Except for correlating and merging sessions another task of the Session Manager is the isolation of session access. Due to the SEDA model events might be processed concurrently and side effects of this approach need to be prevented [26]. The `MemorySessionManager` implemented by Marian Schedenig solves that problem by locking sessions for write-access and sending threads to sleep that are waiting for write-access.

In any case, a session is cloned before it is handed out by the `MemorySessionManager`. Per default this is done via (de)serialization. But same as for the merging of sessions, some contained data might again need special attention for cloning. Hence multiple *Session Clone Managers* may be injected into the Session Manager.

It should be mentioned, that it is possible to have multiple different CorrelationUMOs, Correlation Managers and/or Session Managers per runtime environment. Wiring them together, which is freely possible in either a serial and/or parallel fashion, opens up the doors for a number of scaling and performance boosting strategies. The precise structure and useability of such strategies depends strongly on the domain at hand and lies out of the scope of this introduction.

3.3 Configuring Naiad: Correlations and Bridges

How correlations and sessions are handled in Naiad is configured in a single XML document. On the one hand Session Managers can be defined and Session Merge

[2] dependency injection aka inversion of control (IoC)

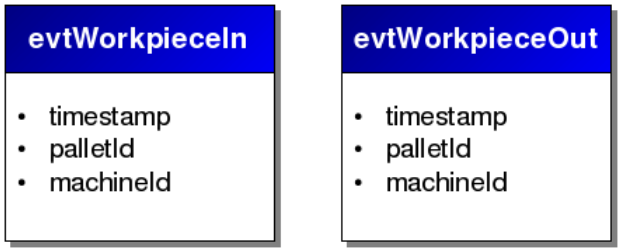

Figure 3.2: Two sample event types with the attributes they contain

Managers and Session Clone Managers can be injected into them. On the other hand Correlation Managers can be defined, each with the following elements:

- An **identifier** for the Correlation Manager.
- **Correlation Items** are defined for certain event types. Each Correlation Item holds one or more **xPathSelectors**, that point to certain values in the events data. If an incoming event's type matches the type defined for the Correlation Item, a *Correlation Match* is created, which holds information about the Correlation Item it belongs to and about the values behind the Correlation Item's xPathSelector for that event.
- Multiple Correlation Items are grouped into **Correlation Sets**. If two ore more events have equal Correlation Matches, that is if they have equal values behind the xPathSelectors of their matching Correlation Item, and their Correlation Items are grouped into the same Correlation Set, they end up in the same session and thus are correlated.
- Multiple Correlation Sets can be grouped into **Correlation Bridges**. If an event belongs to two ore more different Correlation Sets, for which two different sessions existed so far, and those Correlation Sets belong to the same Correlation Bridge, this event becomes the *bridging event* for those sessions, and they are merged into one. With this technique events can be correlated, that do not share any values according to which they could be correlated directly.

There are a couple of less important additional features and configuration options, which are described in [25].

Like always, an example helps to demonstrate things. Let's assume that machines in our assembly line generate (among others) events with attributes as shown in figure 3.2.

The `evtWorkpieceIn` event signals that a certain pallet has entered a certain machine at a certain point in time. The `evtWorkpieceOut` event signals that the machine has finished on the workpiece and the pallet has been released from the machine. Those events are obviously related to each other in some way as they share the same `machineId` as well as the same `palletId`. If we correlated them via the `machineId`,

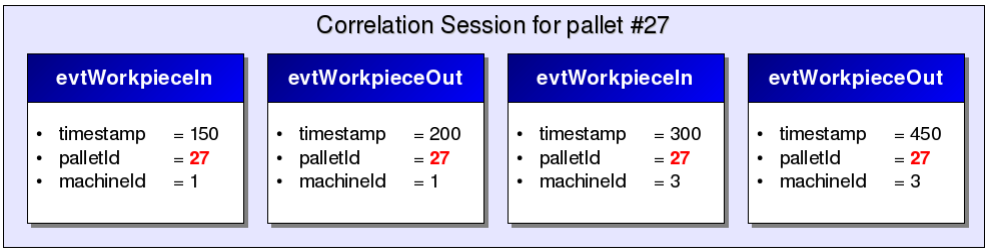

Figure 3.3: Events of the same workpiece correlated into one session

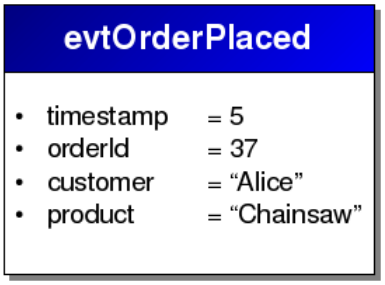

Figure 3.4: The event created when Alice placed an order

all such events from the same machine (but caused by different pallets!) would end up in the same session. This could be something really useful, if somebody wanted to find out things about the machine.

For our example we go the other way and correlate those two events by the palletId. Needless to say, all `evtWorkpieceIn` and `evtWorkpieceOut` events caused by the same pallet will now end up in the same session (figure 3.3), regardless of the machine where they were caused.

This is what a single Correlation Set would do in Naiad. Now let's assume that the product that is assembled on this pallet was ordered by Alice. When Alice placed her order the following event shown in figure 3.4 was generated.

The manager of the company does not yet now how much to charge Alice for the order, but It would be a good hint for him, if he knew how long each machine was occupied for it. Unfortunately, we cannot correlated the evtOrderPlaced event directly to the `evtWorkpieceIn` and `evtWorkpieceOut` events, because they share no attributes for which we could create Correlation Items. Luckily, when the order was sent to the factory, a pallet was assigned to Alice's order and a corresponding `evtPalletAssigned` event was created. Enter: Correlation Bridges!

We can now add a Correlation Item for the `evtPalletAssigned` event to the Correlation Set we already had. This way the `evtPalletAssigned` event ends up in the same session as all other `evtWorkpieceIn` and `evtWorkpieceOut` events for that pal-

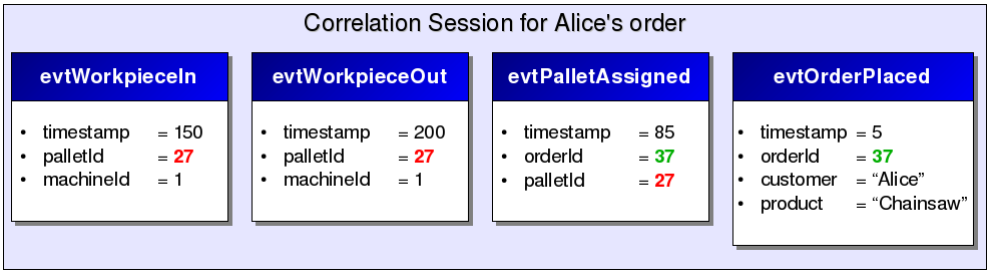

Figure 3.5: A session containing all events relevant to Alice's order

let. Then we create a second Correlation Set with two Correlation Items: One for the `evtPalletAssigned` event, but this time with the xPathSelector pointing to the `orderId`. The other one for the `evtOrderPlaced` event. The only thing left to do now, is to enclose both Correlation Sets in a Correlation Bridge.

Now, though at first two different sessions will be created, they are going to be merged when the `palletAssigned` event is processed, because it belongs to two different session of the same Correlation Bridge. Now the `evtOrderPlaced` event is correlated into the same session as all `evtWorkpieceIn` and `evtWorkpieceOut` events for Alice's order are, and any downstream metric UMO will have a walk-over in calculating the total machine usage for Alice's order. Figure 3.5 depicts that case.

The XML document configuring Naiad for this simple example, could look like listing 3.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<eventServer statusEndpointAddress="vm://stateChange">
  <correlationManager name="correlationManager">
    <correlationBridge identifier="allEvents">
      <correlationSet identifier="palletEvents">
        <correlationItem identifier="workpieceIn" eventType="
            evtWorkpieceIn">
          <xPathSelector>//palletId</xPathSelector>
        </correlationItem>
        <correlationItem identifier="workpieceOut" eventType="
            evtWorkpieceOut">
          <xPathSelector>//palletId</xPathSelector>
        </correlationItem>
        <correlationItem identifier="palletAssigned" eventType="
            evtPalletAssigned">
          <xPathSelector>//palletId</xPathSelector>
        </correlationItem>
      </correlationSet>
      <correlationSet identifier="orderEvents">
        <correlationItem identifier="orderPlaced" eventType="
            evtOrderPlaced">
          <xPathSelector>//orderId</xPathSelector>
        </correlationItem>
        <correlationItem identifier="palletAssigned" eventType="
            evtPalletAssigned">
          <xPathSelector>//orderId</xPathSelector>
        </correlationItem>
      </correlationSet>
    </correlationBridge>
  </correlationManager>
  <sessionManager name="sessionManager" class="at.ac.tuwien.ifs.
      naiad.core.session.memory.MemorySessionManager"
      correlationManager="correlationManager">
</eventServer>
```

Listing 3.2: Configuration of a bridged correlation in Naiad

4 Applying a Conventional Rule Engine to Complex Event Processing

Naiad Rules try to follow the principles of SARI (see section 2.4) very closely. However, for their current implementation a couple of simplifications and modifications have been made to them.

4.1 Types of Rules

Opposed to SARI, there are only two kinds of Elements within a set of rules: *Rules* and *ResponseEvents*.

A **rule** in Naiad Rules consists of the following parts:

- A unique name called *identifier*.
- One or more *patterns*.
- An optional *precondition*.

A **pattern** consists of

- an identifier,
- an event type,
- an optional *ancestorIdentifier* (see section 4.3),
- a optional list of constraints that together form a boolean expression.

A **precondition** consists of

- a type (AND or OR),
- an identifier for OR-preconditions only,
- one or more *preconditionItems*.

A **preconditionItem** consists of

- a *causingRule*'s identifier,
- an identifier for preconditionItems that belong to AND-preconditions only.

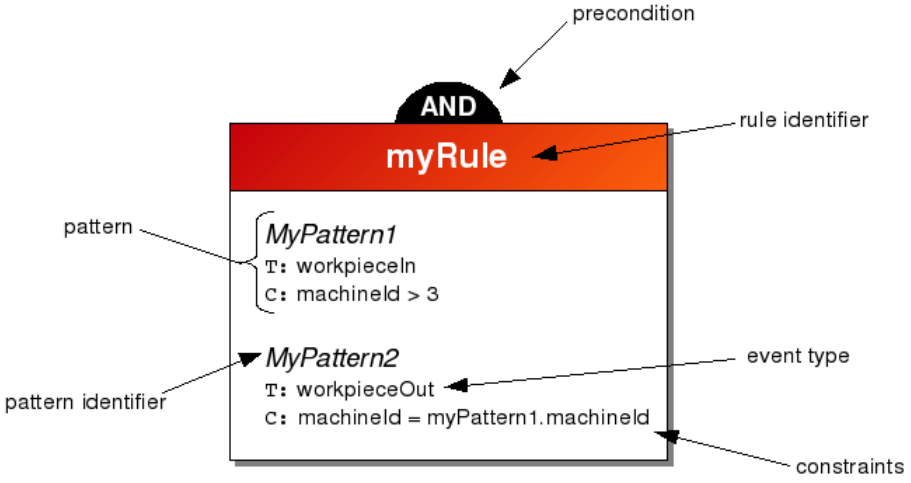

Figure 4.1: A simple example of a rule in Naiad

A rule triggers when there is any combination of events within a correlation session that matches the patterns it contains. Because a rule can contain just one pattern as well, this makes it a mix between event conditions and event patterns in SARI Rules.

A pattern is satisfied in Naiad Rules by any incoming event of the same type, whose attributes satisfy the pattern's list of constraints. Constraints may be *literalFieldConstraints*, which compare an event's attribute to a fixed value, or *variableFieldConstraints*, which compare attributes to other attributes of the same event or other event's that have matched other patterns of the same rule. Constraints may be connected by AND and OR *constraintConnectives* which may contain constraints and constraintConnectives themselves. This way very powerful boolean expressions can be built. In any graphical representation of course, constraintConnectives should be depicted with the help of parentheses and boolean operators. Big parts of the nomenclature for patterns and their subelements have been taken from Drools [9].

A rule that is equipped with a precondition can only be triggered once that precondition is satisfied. PreconditionItems are references to other rules, that must have been triggered for the precondition to be satisfied. Those rules are called a rule's *ancestors*. The difference between AND and OR preconditions is pretty self-explanatory and identical to SARI Rule's precondition ports. Though they are mentioned here for the sake of completeness, preconditionItems are only a technical requirement to implement links between rules and preconditions and should be hidden from the user in any graphical editor.

A rule can only be triggered once per correlation session, even if a later incoming event or alternate combinations of such would satisfy its patterns again. This restrictions not welcome in all situations, but it had to be made to solve the retriggering problem (see section 4.2).

The implementation of something like SARI Rule's false-port has been omitted for Naiad Rules. Though this would help to keep rule sets more simple, there is is nothing that could not be implemented without it. Therefore the decision was made to drop that feature for the first proof-of-concept version of Naiad Rules.

Examples of how to configure rules in Naiad can be found in appendix A.

Response events are discussed later in section 4.4.

4.2 Transforming Naiad Rules to Conventional Rules

Conventional rule engines, which work after the ECA (event condition action) principle, already do a lot of what we need in Naiad Rules. They "wait" for incoming events (sometimes called *facts*) and try to match them on the left-hand side of any rules in a rule set. If a match could be found, the right-hand side of that rule is executed. "Executed" in that respect varies a lot from one product to another. However, there are a couple of things about Naiad Rules that conventional rule engines do not support out of the box.

In this section it will be shown, how those things can be rewritten to fit into the left-hand side of a conventional ECA rule. Because for Naiad Drools [9] was chosen as rule engine, Drools' self-explanatory DRL (Drools Rule Language) syntax will be used to illustrate things. As an example listing 4.1 in a rule's left-hand side would match any *Event* with the value of the attribute machineId being 1 and the value of the attribute palletId being 7. Additionally this would bind the matching event to the variable *myEvent* for referencing it in other constraints of the same rule or its right-hand side.

```
1 $myEvent : Event(machineId == 1 && palletId == 7)
```

Listing 4.1: A basic example of Drools' pattern syntax

Constraints of patterns in Naiad Rules do function just like constraints in a conventional rule's left-hand side. So converting them to something a conventional rule engine understands is trivial. The binding variable is named after the patterns identifier.

To keep the concept of Naiad Rules compatible with a spectrum of rule engines as big as possbile, it could not be assumed that the rule engine of choice recognizes different types of incoming events (like Drools would). Therefore the event type defined in a pattern is transformed to be a part of the pattern's constraint. As an example, the only pattern of the rule in figure 4.2 would be rewritten as shown in listing 4.2.

Preconditions were another feature that needed special attention when transforming Naiad Rules. The solution to this were *InternalEvents*. Whenever a rule is triggered,

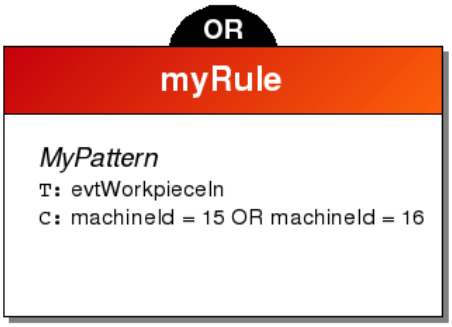

Figure 4.2: A simple rule with one pattern definition

```
1 $myPattern : Event(type == "evtWorkpieceIn" && (machineId == 15
    || machineId == 16))
```

Listing 4.2: An example of the transformation of a pattern's event type

a new InternalEvent, which is not visible from the outside, is created (see listing 4.6). InternalEvents contain the following pieces of information:

- The identifier of the rule that caused them.
- References to its ancestors (see section 4.3).

So if a rule needs to "wait" for one or more other rules before itself is allowed to trigger, this can be achieved by adding a pattern for each of these rules to the waiting rule's left-hand side. Following this suggestion the left-hand side of the rule dependingRule in figure 4.3 would be transformed to the code in listing 4.3.

```
1 $anotherPattern : Event(type == "evtWorkpieceIn")
2 $someInternalIdentifier1 : InternalEvent(causingRule == "myRule1
    ")
3 $someInternalIdentifier2 : InternalEvent(causingRule == "myRule2
    ")
```

Listing 4.3: An example of the transformation of a rule's AND precondition

If the precondition in figure 4.3 was an OR condition, the transformation would look like shown in listing 4.4.

Another problem was caused by the possibility that two already existing sessions got merged because of a bridged correlation. Again, because it was necessary to stay compatible with a spectrum of rule engines as big as possbile, it could not be assumed that a rule engine could handle the merging of two sessions in any predictable way. Therefore merging two sessions is done by adding all events and InternalEvents

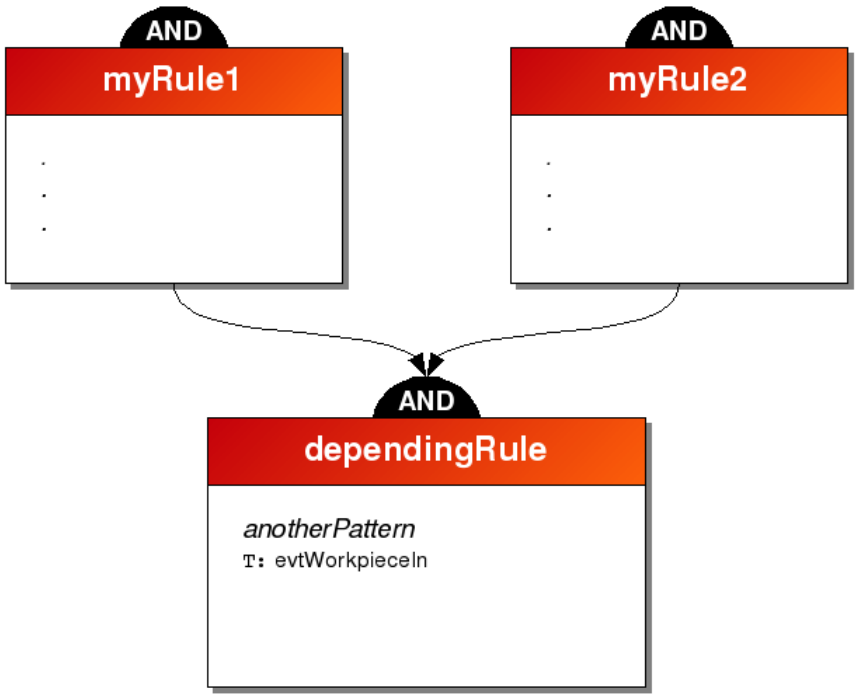

Figure 4.3: A rule's precondition port being connected to two other rules

```
1 $anotherPattern : Event(type == "workpieceIn")
2 $someInternalIdentifier : InternalEvent(causingRule == "myRule1"
      || causingRule =="myRule2")
```

Listing 4.4: An example of the transformation of a rule's OR precondition

contained in either one into a freshly created session. As different rules might or might not have been triggered in either session, they might or might not have been retriggered in the new session.

Hence, to keep predictability, it was decided to prevent the retriggering of a rule entirely by adding the non-existence of its own InternalEvent as a constraint to each rule.

The rule dependingRule from figure 4.3 would therefore be amended with the code from listing 4.5.

```
1 not InternalEvent(causingRule == "dependingRule")
```

Listing 4.5: Code snippet of a rule that prevents retriggering

At first glance one problem still remains: if the same rule has been triggered in two different sessions that later get merged, the new session would contain two different

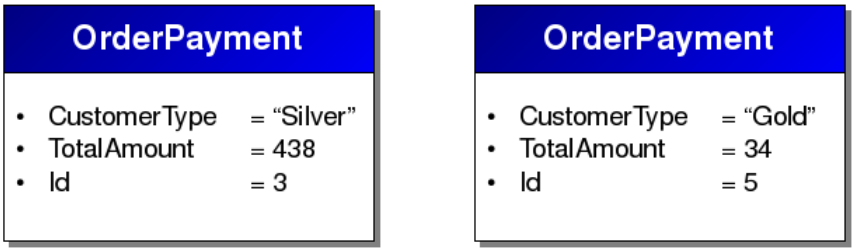

Figure 4.4: Two different events of type orderPayment

internalEvents for the same causingRule. If you took a peek into section 4.3 you would see, that it is not equal which of these would trigger any other rule "waiting" for an InternalEvent of that twice triggered rule.

On a second though this is no problem at all: if there are different sets of events triggering the same rule in different sessions that got merged later, those events could just as well have been inserted after the sessions have already been merged. In that case, the (unpredictable) order of the events incoming would decide, which set of events is then used by any depending rule. So, the possible unpredictability after a merge is just a transformation of an unpredictability that has always been there. If that is a problem, then there is probably a problem with the correlation criteria or rules themselves.

4.3 Ancestor Identifiers

You might have noticed, that not all parts of Naiad Rule's configuration have been discussed yet. The missing parts are about so called *Ancestor Identifiers*. To see what they are good for, let's take a look at the example of SARI Rules from [27], shown in figure 2.9 again.

Suppose that there were two `OrderPayment` Events in a correlation session, each with attributes as shown in figure 4.4.

Maybe having two different `OrderPayment` events in one correlation session would not make too much sense in that example, but let's just accept that fact and take a look at the `Platinum or gold customer` rule. Obviously the event with id of 5 would trigger that rule, while the event with id of 3 would not. However, `Platinum or gold customer` has the `Payment of large order` rule connected to its precondition port. `Payment of large order` on the other hand would only be triggered by the event with the id of 3, not the event with the id of 5. Let's suppose further, that `Order not shipped on time` has been triggered.

Price question: Does `Platinum or gold customer` trigger and thus activate the `Give order discount or goody` response event, or does it not? The answer: This

depends whether the `OrderPayment` event defined in the `Platinum or gold customer rule` refers to the `OrderPayment` event from the `Payment of large order` rule to which it is connected via its precondition port, or to any other `OrderPayment` event contained in the event cloud of that correlation session.

Now SARI Rules could include some extra definition about how to handle that case, but to my knowledge they don't. Further, any fixed definition about those kind of situations should be handled would reduce flexibility as business situations are imaginable where either interpretation could be the desired one.

In Naiad Rules *Ancestor Identifiers* aim to eliminate that shortcoming. They provide a way for rules to say "I explicitly want to refer to a event, that triggered the rule I have in my preconditions". But there is more to think about.

Take a look at the example in figure 4.5, which shows a rule set working on a correlation session containing all events from the same workpiece. The rule `Too much delay` tries to trigger when a workpiece has passed[1] machine 4 and machine 6 - let's assume 6 always comes after 4 in the assembly line - and the time between those events is bigger than a certain threshold. Obviously we need to subtract the timestamp of the event of machine 4 from the timestamp of the machine 6 event and compare the result to our threshold. But in `Too much delay`, how shall we know which of the ancestor's events is from 6 and which from 4?

To solve this question, Ancestor Identifiers in Naiad Rules are a way to even say "I explicitly want to refer to *this* event, that triggered *that* rule I have in my preconditions". The first thing needed is the already mentioned identifier for preconditionItems. With this a rule has a reference to any rule it depends upon. And because patterns already have identifiers in all rules, a rule has a reference to any pattern of any rule it depends upon as well.

An Ancestor Identifier can now be added to any pattern of our rule. It consists of two parts and looks like this: `preconditionItemIdentifier.patternIdentifier`. Figure 4.6 shows how the example from figure 4.5 can be completed using Ancestor Identifiers. The colored bars are just helpful to understand which elements refer to each other, they are not part of Naiad rules.

When defining an Ancestor Identifier for a pattern, no event type must be defined for that pattern. This would not add any informational value anyways.

As utilized in the `m6Pattern` pattern, it naturally is also possible to add constraints to any pattern that contains an Ancestor Identifier.

Please note that the given example could be solved without the concept of Ancestor Identifiers too, but then we would again end up with more complex and thus less

[1] "Passed" is not really accurate to what a `workpieceIn` event stands for, but that assumption is fine for our example.

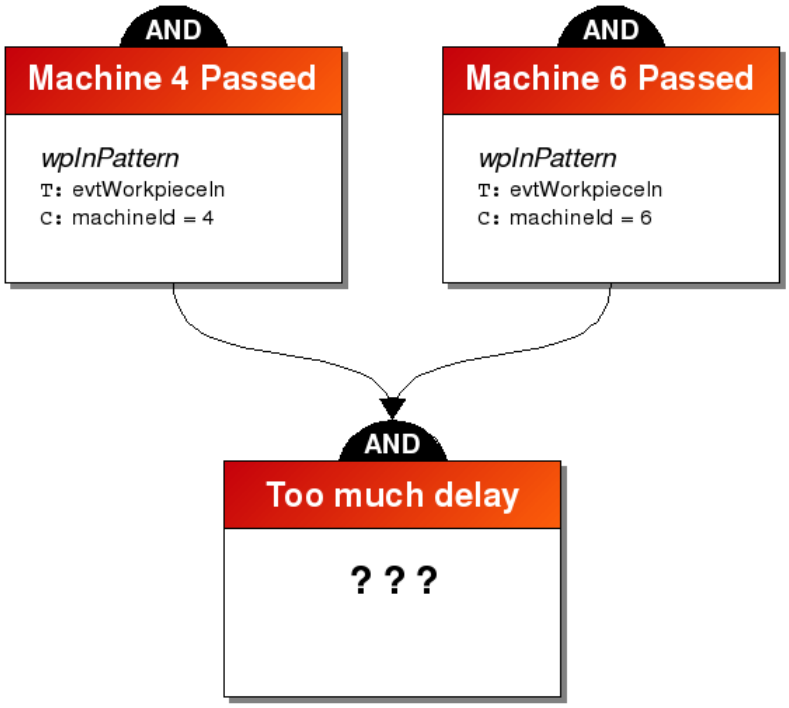

Figure 4.5: A set of rules illustrating the ancestor problem

efficient and less reusable rules. Hence we would not only end up with bigger, but more rules as well.

Another thing to mention is, that even if a pattern does not reference a specific ancestor (event) with an Ancestor Identifier, it still might be matched by that event, just because it defines the same event type and its conditions match. However, the pattern might just as well be matched by any other event in the correlation session.

Ancestor Identifiers can not only be of use, when the same type of event triggered multiple precondition rules, but when one single precondition rule is triggered by multiple events of the same type too. This is shown in figure 4.7.

When using OR preconditions not every preconditionItem gets its own identifier. Because only one of all connected rules will trigger a precondition port, it is enough to give an identifier to the precondition itself. Figure 4.8 depicts that case and even illustrates an example of how Ancestor Identifiers can be utilized to reuse rules. In that case it is `Processing too long`, which is able to detect too long delays from machine 6 as well as machine 8.

Ancestor Identifiers do only work on one level of precondition. It is however possible, to make explicit references to events that are away more than one hop in terms of

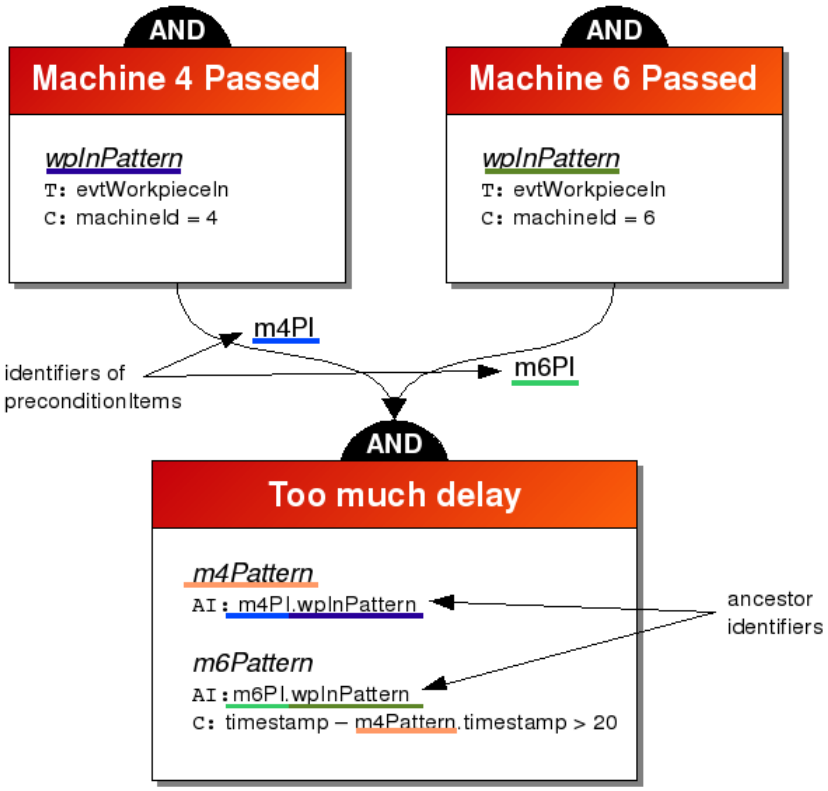

Figure 4.6: Ancestor Identifiers in Naiad Rules

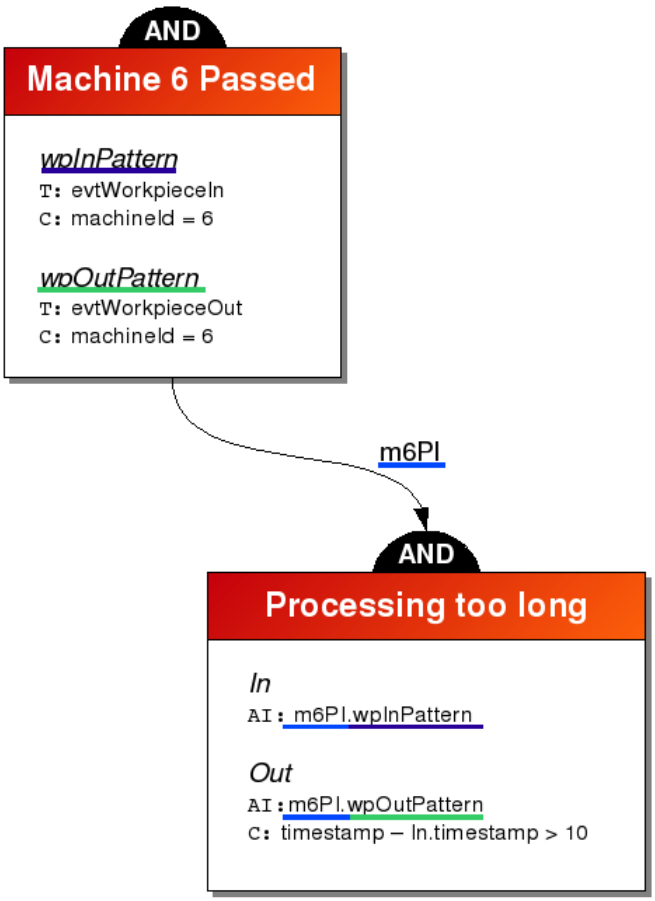

Figure 4.7: Ancestor Identifiers referring to different patterns in the same rule

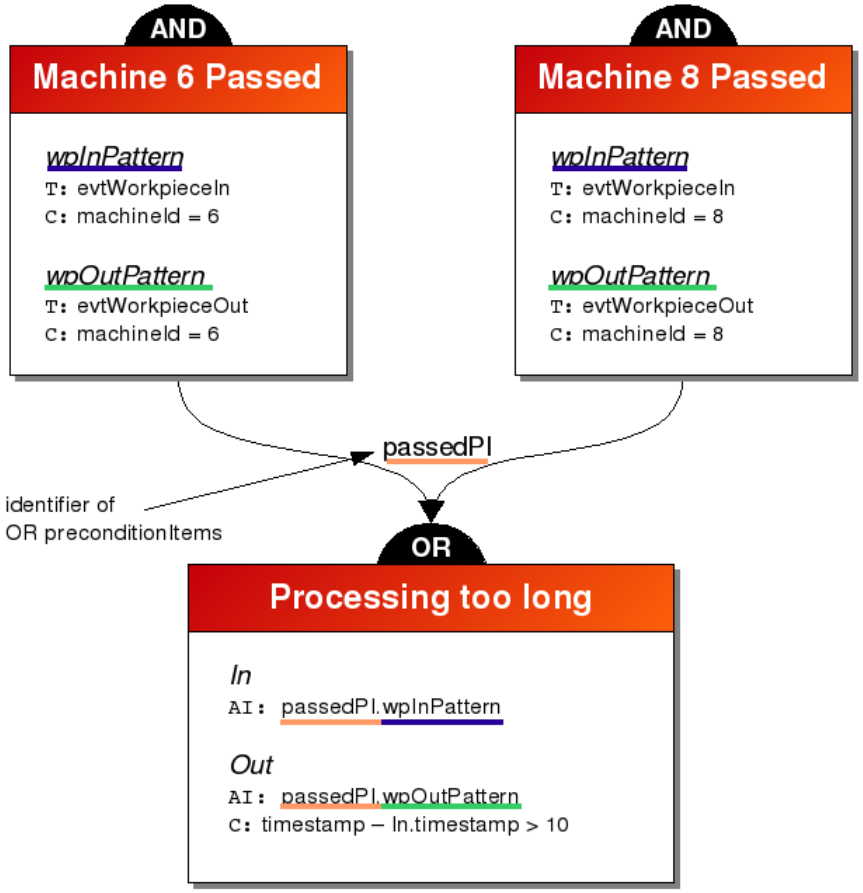

Figure 4.8: Ancestor Identifiers used with an OR precondition port

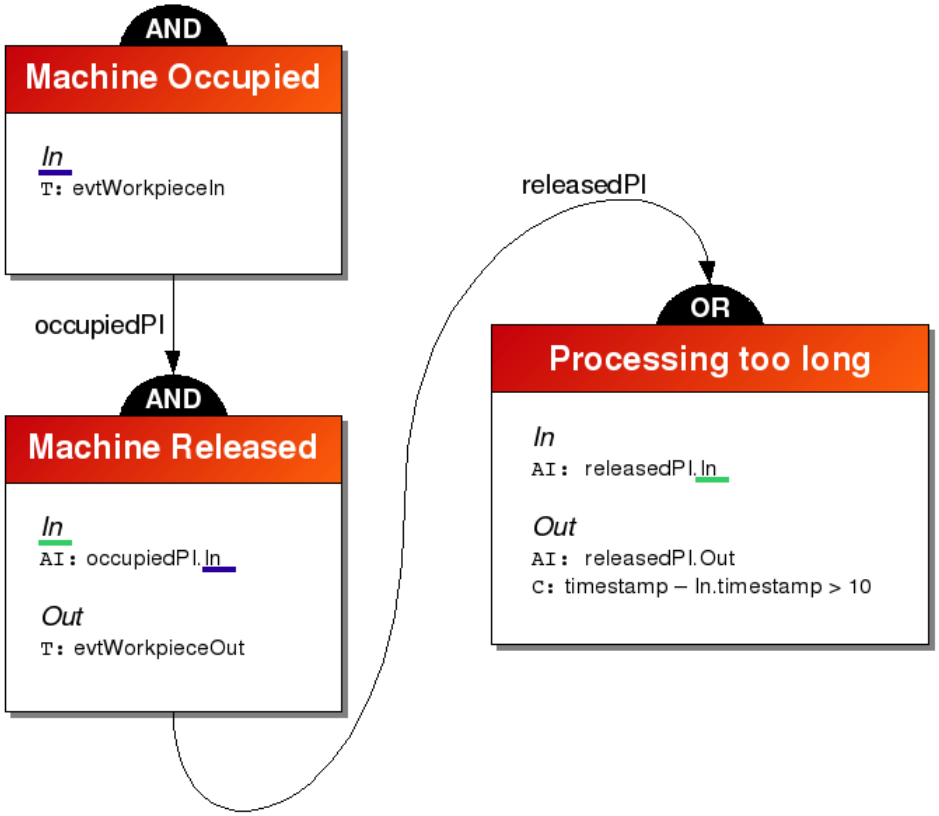

Figure 4.9: Forwarding of Ancestor Identifiers

preconditions by "through-connecting" them in intermediary rules. Figure 4.9 shows an example of that technique. Note that in the example `workpieceIn` events could get compared to `workpieceOut` events from different machines. To increase clarity, the accordant constraints have been omitted.

4.3.1 Transforming Ancestor Identifiers

Just like preconditions, Ancestor Identifiers have to be rewritten to normal constraints a conventional rule engine can understand.

It was already mentioned that Internal Events do not only contain the identifier of their causing rule, but also references to all events that were matched for triggering it. To keep the class for Internal Events generic, those references are stored in a Map. Listing 4.6 illustrates how this would look in Java.

In Drools, the parts of the left-hand side of a rule, which deal with internalEvents and Ancestor Identifiers, would look like shown in listing 4.7.

```
1 // The constructor of InternalEvent takes the causing rule's
     identifier as argument.
2 InternalEvent internalEvent = new internalEvent("Machine
     Released");
3 // The variables In and Out were assigned to the events matching
      the patterns in the rule's left-hand side.
4 internalEvent.addAncestor("In", In);
5 internalEvent.addAncestor("Out", Out);
6 // The global insert() method adds the internalEvent to the
     event cloud of the current session.
7 insert(internalEvent);
```

Listing 4.6: The right-hand side of the `Machine Released` rule in figure 4.9

```
1 $releasedPI : InternalEvent(causingRule == "Machine Released")
2 $In : Event(this == $releasedPI.ancestor_In());
3 $Out : Event(this == $releasedPI.ancestor_Out());
```

Listing 4.7: Parts of the left-hand side of the `Processing too long` rule from figure 4.9

As you can see, the variable binding called `someInternalIdentifier` in listing 4.3 has now been renamed to the identifier of the preconditionItem (`releasedPI` in this case). Not very suprising, the keyword `this` refers to the event to be matched itself. The calls to `ancestor_*` really call helper methods that emulate calls to `getAncestor("*")`. This is a technical requirement of Drools, which is discussed in section 6.1.2.

The pattern for the `Out` event would also contain a constraint for the timestamps (as shown in figure 4.9), which has been omitted for demonstration purposes.

4.4 Response Events

Just like in SARI Rules, response events in Naiad represent events that should be created once all their preconditions are satisfied. Besides the precondition, which works just like any preconditions for rules (including Ancestor Identifiers), response events have an *identifier* and a *template name*.

In Naiad, the generation of response events is not a part of the rule engine, but a service of its own. The template name is just a hint for any connected event-generating service how to compose the new event.

Another important thing to mention is, that response events do not contain any explicit reference to the events that caused them. This makes sense in the context of Naiad, because instead of explicit causal relationships, related events are correlated into sessions. Response events may be re-fed into the system and, assuming an according definition of correlations, eventually will end up in the correlation session they were

created from. Again, this approach helps to keep things simple, memory consumption low and performance high.

For this, the data contained in the response event's ancestors is made available to that service, and thus an implicit definition of causal relationships is enabled. Configuration of such a service lies out of scope for this work. However, for the proof-of-concept version in Naiad, a template engine was chosen and a simple but well working configuration mechanism implemented. See section 6.2 for details about that implementation.

A detailed explanation of how to configure response events in Naiad can be found in appendix A.

5 Implementing Common Patterns of Complex Event Processing

As already stated, the theoretical basis for Naiad Rules, SARI Rules [28], is different from other classic approaches like for instance Luckham's Rapide language and its interpreters. Naiad Rules naturally comes with its own set of advantages and shortcomings, compared to such other approaches. Nevertheless, it is possible to implement some common features of CEP, like presented in [12], not explicitly, but at least implicitly using a couple of tricks.

Tricks naturally come with some loss in performance. This section just wants to show which patterns can theoretically be implemented, which can not and why. In many real-life cases it might even make more sense, to try to utilize Naiad's core assets instead of copying concepts from other systems.

In Luckham's approach Event Processing Agents (EPAs) are small, rule-based, action-processing[1] entities. Three types of EPAs are presented, which all share a common interface. Simplified, this interface defines an input of certain actions and an output of certain actions that an EPA generates following its rules. This is quite the same as the "blackbox-definition" of a Rule Set in Naiad Rules.

Two of those types, *Filters* and *Maps*, can be implemented in Naiad Rules as described in the next sections.

5.1 Filters

Filters are the simplest of all types of EPAs. Yet they are very important to the processing of a huge number of events. In such situations, it is always the best decision, to get rid of all those events irrelevant to our problem first. Filters have been identified as key features of CEP as early as in [19].

A filter marks a certain pattern of events. Events that match this pattern are passed through, events that do not match are dropped. In [12] Luckham explains three different types of filters:

Action Name Filters Such filters match, and thus pass, any actions with a certain name. In Naiad that would be any events of a certain event type.

Content Filters *use the contents of events to decided which ones to pass on to their output.* [12]

[1] For our purposes it is sufficient to set Luckham's notion of actions equal to our notion of events.

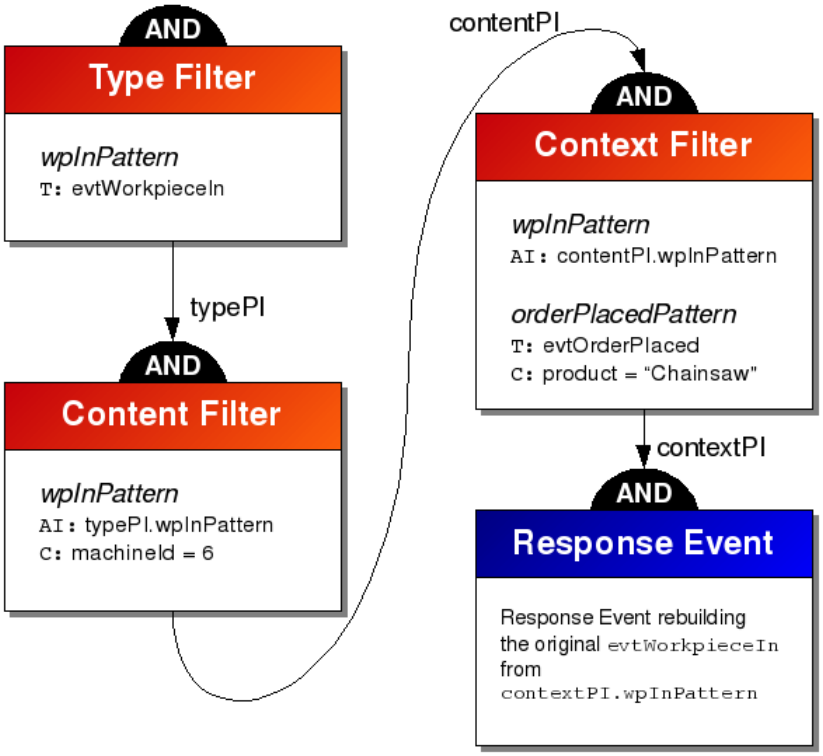

Figure 5.1: A Filter EPA in Naiad Rules

Context Filters *pass on subposets of their input that occur in a certain context.* [12] While *context* does not exactly mean the same, we can think of an event's correlation session as its context in Naiad Rules.

Naiad Rules' concept of Ancestor Identifiers makes it very easy, to depict all three variations of filters in one single example. Figure 5.1 shows that.

Any of the three rules in figure 5.1 could be connected to `Response Event` directly, thus implementing one of the three types presented by Luckham. The Response Event would need to completely rebuild the original `evtWorkpieceIn` event, copying the original event's attributes into a skeleton with placeholders. Listing B.1 shows this for Naiad Rules and the Velocity based Response Event Generator.

This obviously is unneeded work, because the original event already contains a XML based representation of itself. Another implementation of Naiad's Response Event Generator interface, like a *PassThroughResponseEventGenerator* could be an elegant and very simple solution to this. This would still create a new, totally identical event instead of passing the original one, but would nevertheless be much faster than the current approach.

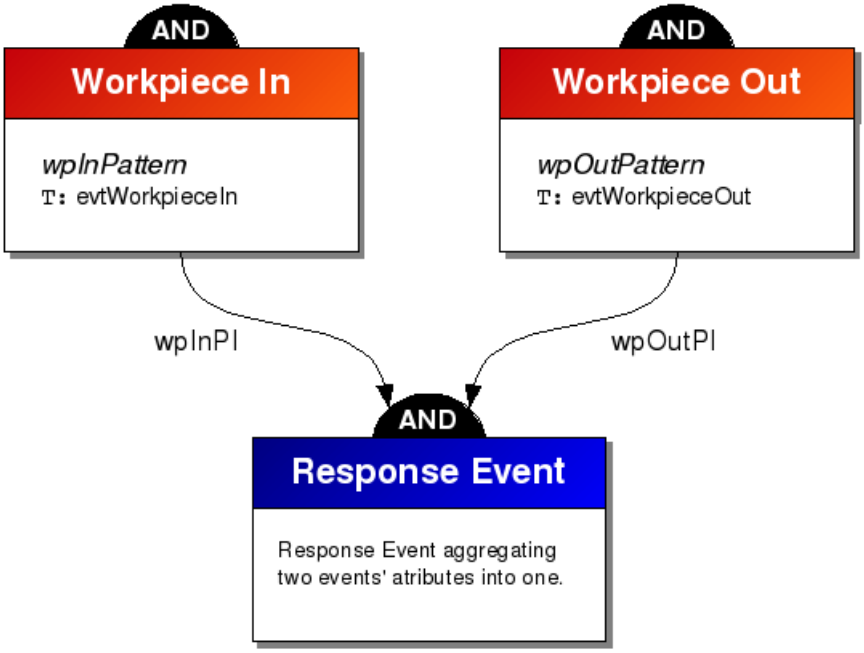

Figure 5.2: A simple Map in Naiad Rules

5.2 Maps

Maps are one core feature of any CEP system, because they provided the means to *aggregate* multiple events into less, higher-level events. This is one of the basic concepts of CEP as already shown in figure 1.4.

A simple example of a map is shown in figure 5.2. The Velocity based configuration of the Response Event can be found in listing B.2.

In Luckham's work, events generated by a Map are *causally related* to those events that triggered the execution of the map. Naiad does not have the idea of causal relation. Instead the forementioned session paradigm is used. Configuring Response Events in "Maps", one has to make sure that they contain the necessary attributes[2] to be allocated to the right sessions, if they are re-fed into the system (see section 4.4 too).

5.3 Event Processing Networks

Event Processing Networks (EPNs) are packages of multiple, relatively light-weight EPAs, communicating with each other through the events they produce. Alternatively, the same results as with EPNs could be achieved in one big set of Naiad Rules, which

[2] `palletId` or `machineId` in our example

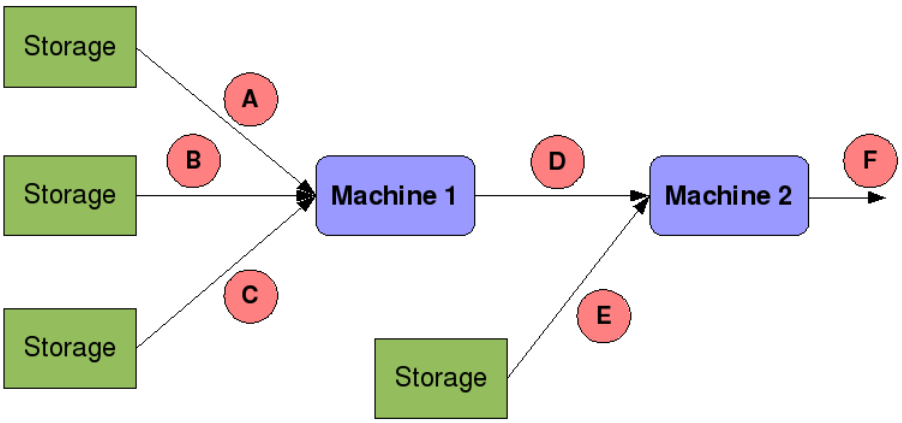

Figure 5.3: A simple assembly workshop

could perfectly make sense, depending on the case. However, Luckham identifies a couple of advantages of EPNs [12]:

- *Agents can be reused easily.* They might as well be taken from some generic EPA library.
- *EPNs can be composed dynamically.* The network could be modified on the fly, while the system is up and running, to reflect changing business situations.
- *CEP can be achieved in small steps.* This facilitates the distribution of the whole CEP system onto multiple physical systems and thus enhances scalability.

The next example from the SAW domain illustrates, how multiple Maps, each representing a single workstation (machine) in a workshop, can be connected together to form a EPN representing the whole workshop. Assume we have a very simple workshop as shown in figure 5.3. It consists of two workstations, each processing multiple input-workpieces into a single output-workpiece. For example, `Machine 1` builds product `D` out of products `A`, `B`, and `C`.

In our example we have two kinds of events:

Sensor Events Those are events generated whenever something "physical" happens in our workshop. We only need the `evtWorkpieceOut` event, signalling that a workpiece has left a machine. In our simplified example, this is equal to the workpiece entering the next machine - that is, we don't have any conveyor- or waiting-time between machines.

"Meta" Events Those events are generated from within the CEP system, whenever according sensor events occur. They contain the state of a workpiece. In our example, we want to utilize those events, to transitively compute a workpiece's cost. Note that technically they are normal events just like sensor events too - it is just their meaning to the example, that makes them special.

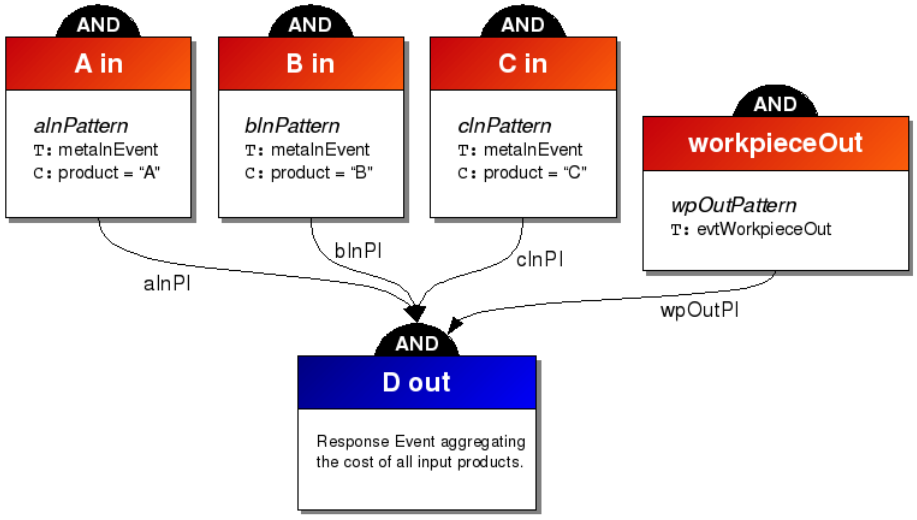

Figure 5.4: EPA of `Machine 1` from figure 5.3

Let's take a look at machine 1 and just assume for the moment, that (meta) events of type `A in`, `B in` and `C in` are correlated into a session, representing all events related to `Machine 1`. Figure 5.4 shows `Machine 1`'s EPA. Listing B.3 shows the configuration of the `D out` Response Event.

Once a sensor event arrives, that the workpiece has left machine 1, the `D out` event is created, which maps the cost of products `A`, `B` and `C` into the cost for product `D`.

The next node in our EPN would be some EPA representing the conveyor between machines 1 and 2. Since we dropped that for our example, its configuration is very simple as shown in figure 5.5. `D in`'s configuration is trivial as well, so it is not listed here.

The EPA for machine 2 (figure 5.6) looks very similar to that of machine 1. Again, we assume that a `E in` event somehow got merged into `Machine 2`'s correlation session. However, the `D in` event is the one generated by our conveyor's EPA. Again, the cost of product `F` is transitively computed from the cost of product `D` and `E`. The configuration of F out is similar to listing B.3.

Figure 5.7 shows an overview of our EPN and how the single EPAs are connected together. Additionally to the connections shown there, all EPAs receive the sensor events relevant to them.

This concludes the example. We have seen how multiple EPAs can be connected to form an EPN that transitively computes a workpiece's cost.

In Naiad, all those EPAs could run within the same virtual machine, or easily be distributed among different physical systems.

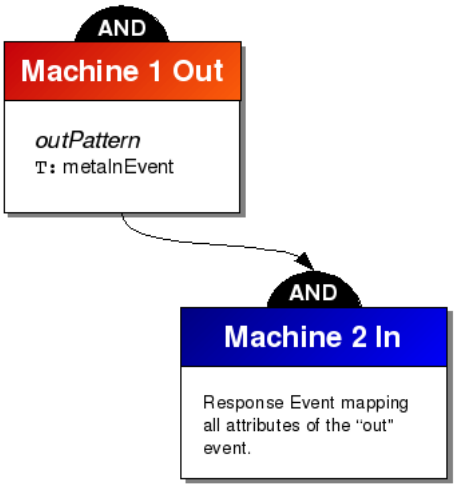

Figure 5.5: Simple EPA for conveyor in figure 5.3

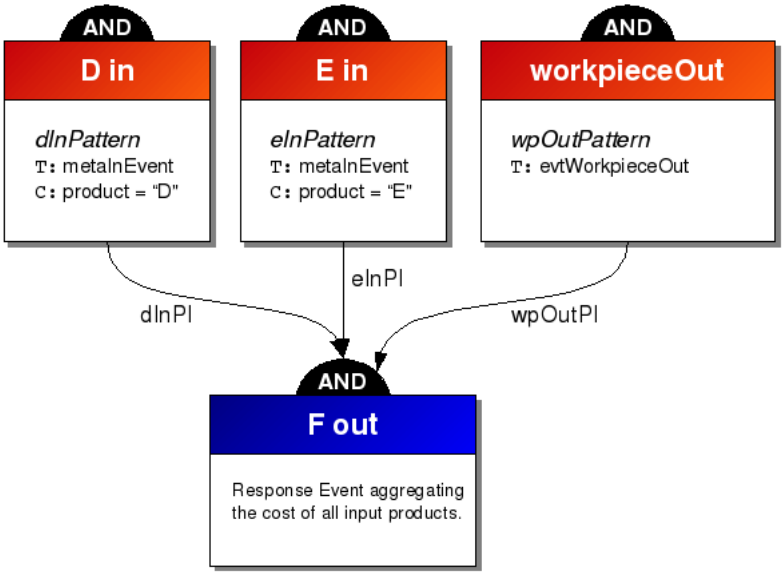

Figure 5.6: EPA of `Machine 2` from figure 5.3

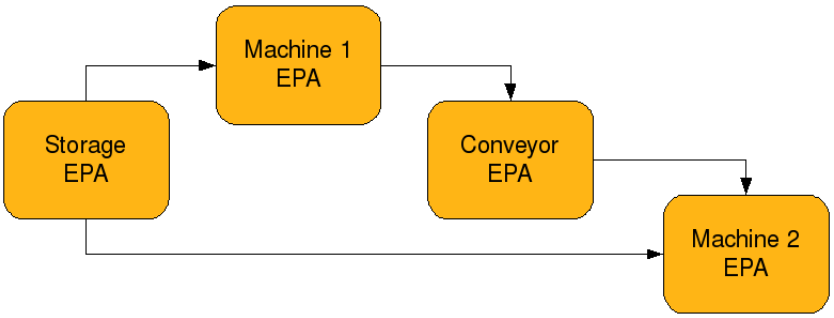

Figure 5.7: EPA layout for workshop in figure 5.3

5.4 Distributed Event Detection

Distributed Event Detection is described in [16]. It applies to CEP systems where the transfer of events between nodes (whatever nodes that might be) is either time consuming, expensive, or both. In such scenarios it makes sense to aggregate events close to their source, before huge and unnecessary amounts of events are sent through such expensive connections.

Suppose for instance, that we have two factories on either side of the globe. Naturally we would only send aggregate events of finished products, or even whole finished orders to our headquarters, but certainly not status messages from each and every single sensor.

However, there are cases where things are not so clear, and where one might not be able to consider these things before "going live". In [16] a number of *Distribution Policies* are discussed, which define how a CEP system would reorganize itself and its EPAs at runtime, to obey to a certain priority of goals.

This lies far beyond the scope of this work though. It shall nevertheless be noted, that the ESB-like nature of Mule, the framework underneath Naiad's hood, perfectly supports the quick reorganization of its components, and thus would facilitate the autonomous execution of such a thing as well.

5.5 Naiad's Limitations

There are of course very convenient patterns that can not be implemented with Naiad Rules (yet). The two most desirable are presented here. Possible strategies to implement them are later discussed in chapter 9.

5.5.1 Constraints

The third type of EPAs are Event Pattern Constraints or just Constraints. They can be seen as *special kinds of maps*, but opposed to maps their purpose is not aggregation,

but *detection* [12].

> Event Pattern Constraints have important applications in detecting violations of business or security policies in enterprise systems [12].

Luckham defines three kinds of constraints:

Never Constraints describe patterns that must never occur in a system.

Always Constraints contain patterns that must always occur in *some relationship to some other pattern.*

State Constraints require *some part of a system's state to satisfy certain conditions whenever a pattern of events happens.*

Implementing Never Constraints is trivial in Naiad Rules. Just model any pattern that must not occur as a set of rules and connect it to a "Constraint Violation" Response Event.

Always Constraints show where the limits of the current state of Naiad Rules are. As Luckham explains (see [12]) they require some kind of "event store". However, no such thing is currently provided for rule sets in Naiad Rules. On the technical side this would not be too complicated to accomplish, but providing a clean and understandable way of configuring such a feature is somewhat more complex. Section 9.1 discusses that aspect.

State Constraints require a rule set to have access to external sources like a database. Being technically simple as the forementioned "event store" for Always Constraints, such a feature implicates the same problems and yet to make considerations as well.

5.5.2 Cut and Join

Cut and Join is a technique described by Chen, Jeng and Chang in [4]. Their motivation was to cut relatively big and thus complex events into multiple smaller ones before they are processed by a correlation engine. The reason for this was, that most available correlation engines need to work with events consisting of unique name/value-pairs. The event in listing 5.1 obviously does not satisfy these requirement, as it contains multiple `item` tags of no predefined count.

The Cut and Join approach would create three new events, where all attributes are accessible through a unique name. Listing 5.2 contains an example.

```
<event type="purchaseOrder">
  <customer>
    <name>H. J. Simpson</name>
    <address>742 Evergreen Terrace</address>
  </customer>
  <items>
    <item>
      <product>Donuts</product>
      <quantity>10</quantity>
    </item>
    <item>
      <product>Ham</product>
      <quantity>1</quantity>
    </item>
    <item>
      <product>Duff Beer</product>
      <quantity>200</quantity>
    </item>
  <items>
</event>
```

Listing 5.1: XML representation of an event that can not be mapped to unique name/value-pairs

```
<event type="purchaseOrderItem">
  <customerName>H. J. Simpson</customerName>
  <customerAddress>742 Evergreen Terrace</customerAddress>
  <product>Donuts</product>
  <quantity>10</quantity>
</event>
```

Listing 5.2: XML representation of an event cut and joined from listing 5.1

Opposed to other correlation engines, Naiad's utilization of the XPath language enables it to cope with events like the one in listing 5.1 as well. At least as long as there is a predefined number of non-unique items per event. Unfortunately in our example and in most other real-life situations this is not the case. Naiad lacks the possibility to work with *sets* and *counts* of attributes of the same type. This feature is definitely something worth to work on in the future.

6 Connecting a Rule Engine to Naiad

This chapter will present some of the more technical details and problems that occurred, when the prototype for Naiad Rules was implemented. It will further explain the choice of technologies with a focus on "open source licenses" and a maximization of modularity and extensibility.

6.1 The Rule Engine - JBoss Drools

As workhorse for Naiad Rules the open source rule engine JBoss Drools [9] was chosen. In 2005 the up-to-then stand-alone project Drools federated with JBoss, but it is still being developed by a multinational team of programmers of varied origin, currently under the lead of Mark Proctor[1].

> Drools is a business rule management system (BRMS) [9].

The algorithm Drools uses under its hood is called ReteOO, which is an implementation of the Rete algorithm (see section 6.1.1) optimized for object oriented programming languages, especially Java.

When programming with Drools, most interaction will be made with the `WorkingMemory` class. Simplified, one could think of it as a bucket, where *facts* of any kind (Java objects of any class) can be thrown in.

Each WorkingMemory is connected to a `RuleBase` which contains the (business) rules how those facts should be interpreted and acted upon. A WorkingMemory's rules are not always evaluated, when a new fact is inserted, but only when its `fireAllRules` method is called.

A RuleBase contains an arbitrary number of *Rules*, each consisting of a left-hand side (LHS) and a right-hand side (RHS), which together are very much like any `if-then-else` statement in other programming languages. Listing 6.1 shows the (simplified) "Hello World" example of a rule from the Drools documentation.

Facts that are inserted into Drools WorkingMemories **must** follow some principles of Java Beans. To be precise, the must provide a getter and a setter method for each of their attributes, which is mentioned in any Rule of the WorkingMemory. For the ReteOO algorithm to function properly, the return values of the getter methods must not change. Drools does provide other means to safely change attributes of already inserted facts, but they naturally come with a cost in overall performance.

[1] `http://www.markproctor.com`

```
1 rule "Hello World"
2   when
3       $m : Message( status == Message.HELLO, $message : message
            )
4   then
5       System.out.println( message );
6 end
```

Listing 6.1: Simplified version of Drools' "Hello World" example

In the example of listing 6.1 there is a class `Message` which provides two methods named `getStatus` and `getMessage`. Further it contains a constant named `HELLO`. When `fireAllRules` is called on the surrounding WorkingMemory, the first Message with a status of `Message.HELLO` matches the LHS and is bound to the variable `m`. Additionally the return value of its `getMessage` method is bound to the variable `message`. The RHS of the rule is written in plain Java code and just prints out the text bound to that variable before. It is possible to user other pluggable dialects for a rule's RHS.

If a rule's LHS contains more than one *pattern*, Drools tries to find fitting sets of facts within all possible combinations of all facts that have been inserted into the WorkingMemory.

Additionally to rules a WorkingMemory can define *globals*. Those are references to Java objects that are usable from within all rules, but which are not used by ReteOO to find matches in the LHS of a rule. Globals are predominantly used to let rules communicate with the "outside world".

WorkingMemories can be of a stateful or stateless session. While the bucket analogy holds for stateful sessions, stateless sessions only operate on a fixed set of facts and throw them away afterwards. Naiad Rules only uses stateful sessions.

In Drools RuleSets can be modified dynamically, on the fly. This feature is not used in Naiad Rules yet, but keeps the door open for some interesting extensions to the current implementation.

The fastest way to configure Drools and create rules for it from scratch is its own rule language, which was used in listing 6.1 and prior examples. For this language Drools offers a convenient editor plugin for the popular Eclipse IDE[2] too. The rule language can be extended with selfmade domain specific languages (DSL).

Further Drools can load decision tables from popular file formats like Microsoft Excel's or OpenOffice's. Drools also comes with a web-based BRMS (business rules management system) targeted at a technically less experienced audience as shown in figure 6.1.

[2] `http://www.eclipse.org`

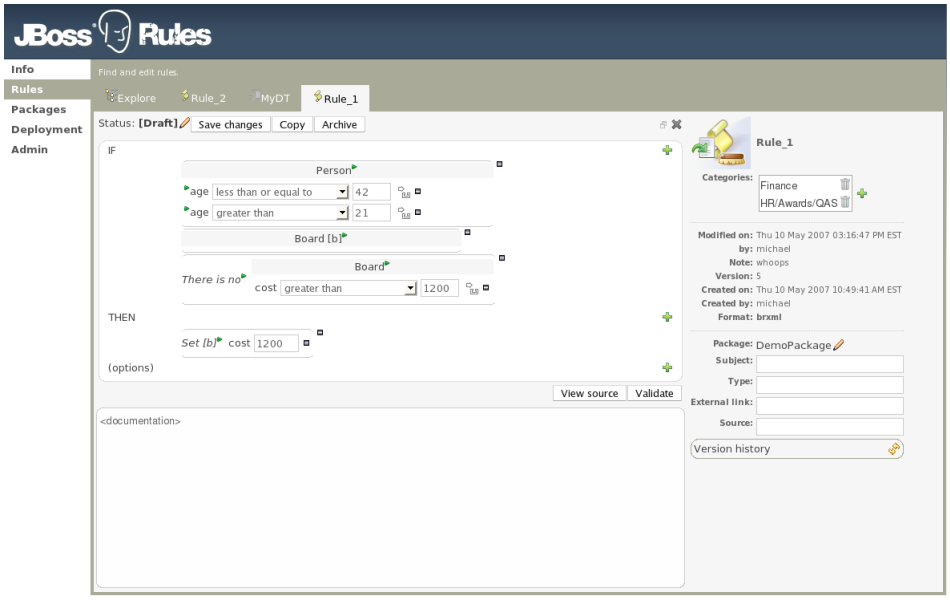

Figure 6.1: Screenshot of Drools' web-based BRMS [9]

The last (and probably most cumbersome) way to configure a RuleBase in Drools is via a XML document. In Naiad this way was chosen, because it is automated program code that transforms Naiad Rules' configuration to Drools' configuration, and XML is the best of all choices, when it comes to automatically creating highly nested data structures (like rule definitions).

Cumbersome, because up to now there is almost no documentation provided for the XML configuration of Drools. Further, the currently available stable version (4.0.7), which is used for Naiad Rules, contains a hard-to-find bug specific to the configuration via XML. At the time this is written, the Drools development team unfortunately failed to respond to any reports[3] about that bug. Therefore a temporary quick fix has been inserted into Naiad Rules.

Listing 6.2 shows the rule from listing 6.1 in XML format. This of course is only a basic example and does neither exhibit all features of Drools' XML configuration nor those used for Naiad Rules.

Starting from version 5, which is not a stable release at the time of writing, Drools will come with some CEP features (*Drools CEP* that is) out of the box.

There are of course a lot of other features packed into Drools, but they are not used in Naiad Rules, and therefore of no interest to this work.

[3] `http://article.gmane.org/gmane.comp.java.drools.user/11802`

```
<rule name="Hello World">
  <lhs>
    <pattern identifier="$m" object-type="Message" >
      <field-constraint field-name="status">
        <qualified-identifier-restriction evaluator="==">Message
           .HELLO</qualified-identifier-restriction>
      </field-constraint>
      <field-binding field-name="message" identifier="$message"
         />
    </pattern>
  </lhs>
  <rhs>
      System.out.println(message);
  </rhs>
</rule>
```

Listing 6.2: Simplified version of Drools' "Hello World" example in XML format

Another rule engine, that has been short-listed for use in Naiad Rules was Jess[4], which, just like Drools, has the Rete algorithm under its hood as well. It too would have been suitable for the task. However, Drools won the race due to Jess' less attractive licensing model. Jess is only free for academic use, but has to be licensed for any commercial application.

6.1.1 The Rete Algorithm

The *Rete Match Algorithm* was developed by Charles L. Forgy and presented in [8] and later [7]. As the name says, it was developed for fast and efficient matching of patterns in production systems. Its high-performance derives from the exploitation of two assumptions about the nature of production systems:

Temporal Redundancy This principle suggests that only a few pieces of data change in working memory from one cycle to another. Therefore, in short, each cycle Rete only watches the *changes* that happen to a working memory, rather then the working memory itself.

Structural Redundancy With multiple rules being applied to the same set of facts, it is very likely that the left-hand sides of these rules contain big portions of similarity to each other, and only differ in a view conditions. Rete benefits of that fact by compiling rules' LHSs before execution and thus discovering those similarities.

Rete is the Latin word for *net*, and it describes perfectly what the algorithm does. On compile-time it builds a a network of nodes, each representing a certain (atomic)

[4] `http://www.jessrules.com`

condition in the whole set of rules. New facts enter this network via a special *root-node* and are then sent and filtered through so called *input-nodes* until they finally arrive at a *terminal-node*. Each terminal-node represents a rule, and each fact arriving at a terminal-node represents a whole rule been satisfied.

There are of course a lot more things to know about how Rete networks are built and how facts propagate through them, but this is out of scope for this work. In [8] Forgy presents a hardware based approach for the Rete algorithm, which supposedly is able to boost the algorithm's performance by another 25% compared to a standard microprocessor based interpreter. This however does not add to this work as well.

Since 1979 a lot of improvements have been committed to, and a quite some alternatives have been derived from Rete. For instance Charles Forgy himself developed *Rete II* in the 1980's, which, unlike the original Rete algorithm, is a closed-source commercial product. Another spin-off is ReteOO, which is the engine used in JBoss Drools.

6.1.2 The Mapping Problem and its Solution

When integrating the Drools rule engine with Naiad, one major problem regarding structure of facts. As already mentioned, Drools requires facts to follow the Java Beans standard. In particular, any attribute of a fact, that is part of any condition of a rule must be accessible through a `getAttributename` and a `setAttributename` method. To provide such methods naturally is no big deal when writing a custom application for any special domain. However, Naiad Rules and Naiad in general aim to become an out-of-the-box and ready-to-use application for non-programmers. Hence, the burden of writing customized Java classes for each and every new domain was certainly not an option.

It should be noted that Drools in fact *does* provide other means to access a fact's attributes. However, those can not leverage Rete's assumptions of Temporal Redundancy (see section 6.1.1) and thus would dramatically influence the applications performance in a negative way.

In Naiad Rules' configuration the solution to the Mapping Problem is called *attributeMapping*. Listing 6.3 shows a `workpieceIn` event, as it would be represented in XML. `ComponentName` in this case refers to the machine that was entered by the workpiece.

```
1 <event id="79" timestamp="66880" type="evtWorkpieceIn">
2   <payload key="WorkpieceId">SW003</payload>
3   <payload key="ComponentName">DS3</payload>
4 </event>
```

Listing 6.3: XML representation of a `workpieceIn` event

If we wanted to create a rule similar to the `Machine 4 passed` rule from figure 4.6 Naiad Rules' XML configuration for that rule would look like shown in listing 6.4.

```
<rule identifier="Machine_4_passed">
  <pattern type="evtWorkpieceIn" identifier="wpInPattern">
    <literalFieldConstraint fieldName="machineId" evaluator="=="
        value="4" >
  </pattern>
</rule>
```

Listing 6.4: XML configuration of a rule in Naiad

Now how would Naiad Rules know, that the fieldName of `machineId` refers to the payload with key `ComponentName`? The solution is shown in Listing 6.5.

```
<xPathMapping>
  <mappingItem xPath="//payload[@key='ComponentName']" fieldName
     ="machineId" type="integer" />
</xPathMapping>
```

Listing 6.5: Sample configuration of Naiad Rules' attributeMapping

There must be only one `xPathMapping` element per ruleset. It may contain any number of `mappingItems`, each mapping a XPath expression to a fieldName to be used by the rules.

As an additional benefit, while other parts of Naiad can interpret an event's attributes only as string values, we can define an attribute's type within Naiad Rules. This comes in very handy when evaluators other then `==` are to be used in a constraint. Currently supported types are `string`, `integer`, `float` and `date`.

Another advantage is, that more meaningful names can be assigned to attributes than what they had in an event's XML representation. Just like in the example above. Needless to say, that this feature should be used very carefully in order to avoid any confusion.

Obviously the mapping could be done automatically as well. XPath expressions could be written into a rule's conditions directly and be secretly and internally mapped to some dummy-names only known to Naiad Rules and Drools. The presented solution was implemented nevertheless, because it definitely is easier to handle and bugs are easier to track down for the time being. Further, if automatic mapping would take place, some other mechanism must be found to define an attribute's type. However, this still is a topic of discussion, and the `xPathMapping` element might be dropped or replaced in a future version.

One more though about that arises, when one thinks about a - yet to develop - graphical editor for Naiad Rules. If we could rely on such an editor to always set the `mappingItems` correctly, we can leave the feature like it is and keep its advantages while hiding its disadvantages from the user at the same time.

What is technically happening for the mapping to take place is a different story. When Naiad Rules starts up, the class `Event` is dynamically extended to the class `Event_`*`NameOfRuleSet`*. This extending class provides getters and setters for all `mappingItems` defined and is then used to load incoming events in XML format into a correlation session's working memory.

Java does not support the dynamic creation or modification of classes or interfaces. Java's concept of reflection is not powerful enough either. The only solution was to create the new class' bytecode directly and overload the virtual machine's standard `ClassLoader` with a customized one for that job. To create the bytecode Object Web's ASM[5] is used.

Naturally, being quite low-level compared to standard Java coding, the development process of this part took its time and involved a good amount of time-consuming kickbacks. The result is not quite what could be described as intuitive and perfectly readable as well. Luckily all of this is encapsulated in its own classes and package, and can be replaced once a better solution comes up (which won't happen unless some other and better technology but Drools is found and chosen).

On the other hand, now that things are stable, the generation and loading of bytecode is fast and works like a charm.

6.2 The Response Event Generator - Apache Velocity

As already mentioned, the Naiad Rules itself does not know anything about how to create new events and where to publish them. Via injection Naiad Rules knows a *Response Event Generator (REG)* to which it can send the data contained in all ancestors of a ResponseEvent and the name of a template which should be used to create the event. The name of the template can be any string, just as long as the actual implementation of the REG understands it. Therefore, the definition of a ResponseEvent in Naiad Rules' XML configuration is relatively simple as shown in listing 6.6.

As REG in Naiad Apache's Velocity Template Engine [1] was chosen. Only a very lightweight wrapper class connects Velocity to Naiad, the biggest part of required features came out-of-the-box. A sample template for a Response Event in Naiad can be found in listing 6.7. Except for the data of all ancestors contained in the variable `$data`, the Velocity REG automatically provides a formatable variable containing the current time (`$now`).

[5] `http://asm.objectweb.org/`

```
<responseEvent identifier="myResponseEvent" template="
    myResponseEvent.vm">
  <precondition type="OR" identifier="cause">
    <preconditionItem causingRule="anotherRule" />
  </precondition>
</responseEvent>
```

Listing 6.6: Sample definition of a Response Event in Naiad Rules

```
<event type="myFirstResponseEvent" timestamp="$now.format('HH:mm
    :ss', $now)">
  <payload key="WorkpieceId">$data.cause.wpInPattern.workpieceId
     </payload>
  <payload key="anotherPayload">$data.cause.anotherPattern.
     anotherPayload</payload>
  <payload key="moreFun">some totally ancestor-unrelated value</
     payload>
</event>
```

Listing 6.7: A Response Event's definition for Velocity

It is not a requirement that events generated by the REG have to be in XML format. It is just very convenient, because Naiad already provides according parsers. More, it is even not a requirement for the REG to create Events at all. Being connected to any other components via flexible Mule, an REG could literally create any Java class out of the data submitted to it, just as long as all subscribing components can work with it or Mule provides proper transformers.

Of course such considerations should be made with care, because any such variation to the original theoretical concept of Response Events could very easily lead to confusion and error-proneness.

6.3 Integration

All services of Naiad are wired together using the Mule framework [17]. Naiad Rules and the Response Event Generator are no exception to that. Figure 6.2 shows a minimal setup of the components that make up Naiad Rules and how they relate to and rely on other parts of Naiad.

On startup the *Rule Agent* loads its configuration - its set of rules - and creates an according ClassLoader (see section 6.1.2). At the same time the REG is instantiated too, but doesn't do anything yet. Like the Naiad Manager, Rule Agents and REGs are implemented as *Mule Agents*. Agents are globally available services, that exist outside the flow of events and are not managed by SEDA principles. A *SessionMergeManager* and a *CloneManager*, both able to handle Drools' WorkingMemories, are created for

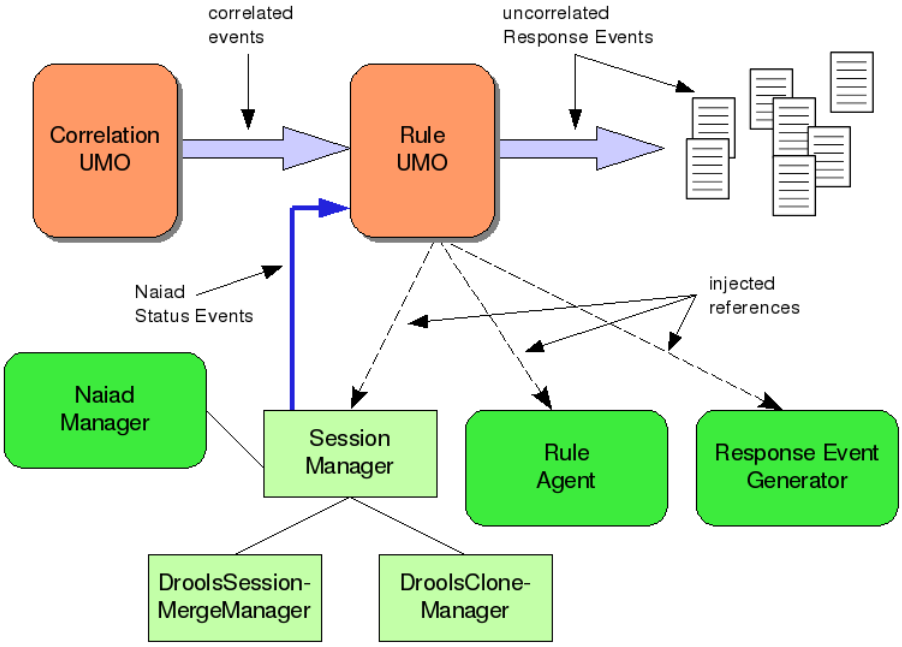

Figure 6.2: Integration architecture of Naiad Rules

the Session Manager.

At runtime, events that have been correlated into sessions enter the *RuleUMO*. Via injection the RuleUMO knows its Session Manager[6], its Rule Agent and the Response Event Generator assigned to it.

When a new event enters the RuleUMO, all the sessions it belongs to are checked out from the Session Manager for write access. If a session already contains a Drools WorkingMemory the new event is inserted and the `fireAllRules` method is invoked. If a session does not contain a WorkingMemory, one is created and stored in the session before the same procedure is applied.

If the evaluation of the new event resulted in the creation of a response event, its ancestor's data and its template name are sent to the REG, which then creates an according *Response Event* and returns it to the RuleUMO. The UMO then publishes the event at its *OutboundRouter(s)*. An arbitrary number of Response Events can be generated in each cycle.

What services wait at the other side of the router is not relevant to the RuleUMO and configured in Mule. That configuration might even change during runtime without affection the RuleUMO in any way. However, because the Response Events issued by the RuleUMO are neither correlated nor equipped with a UUID, it is probably a best practice to send them through a UuidUMO and a CorrelationUMO first. This might be the same CorrelationUMO where their ancestors have been processed ("*re-feeding*

[6]To be precise, just like the CorrelationUMO the RuleUMO only knows the Naiad Manager and requests references to the Session Manager from there.

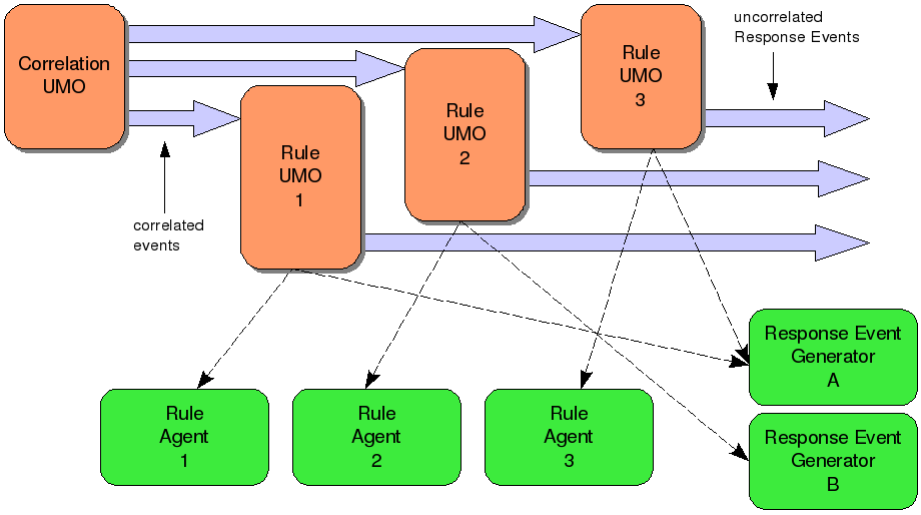

Figure 6.3: Naiad Rules architecture with multiple RuleUMOs

events"), or a completely different one, with its own configuration and correlation criteria, representing a higher level of aggregation in the whole CEP system.

The RuleUMO is a listener for *status events* from the Session Manager too. These include notifications about merged or deleted sessions, upon which the RuleUMO acts accordingly (e.g. invoking the `fireAllRules` method for freshly merged WorkingMemories).

It is perfectly possible to have multiple independent Rule Agents and/or REGs at the same time, each with its own set of rules or templates. Figure 6.3 illustrates that in a more complex example. Due to the extensive use of interfaces, these could even be totally different implementations.

6.4 Testing

Testing of Naiad is automatically done with version 4 of the JUnit testing framework [3]. Naiad Rules follows this approach too. About 90 percent of the code that makes up Naiad Rules are covered by conventional JUnit Tests. Some of them make use of lightweight mock-up classes as some classes can not be tested standing alone. Those mock-up classes and their usage are rather conventional and not to fancy either.

However, the remaining ten percent are of utmost importance for Naiad Rules' functionality and well-behaving. Those are the parts, where Naiad Rules' configuration files are transformed into Drools Rule Sets and where those Rule Sets are applied to incoming Events upon which ultimately Response Events are created. As this functionality is the result of all parts of Naiad Rules working together, and being connected with each other through Mule, it can only be tested within a complete, running Mule application.

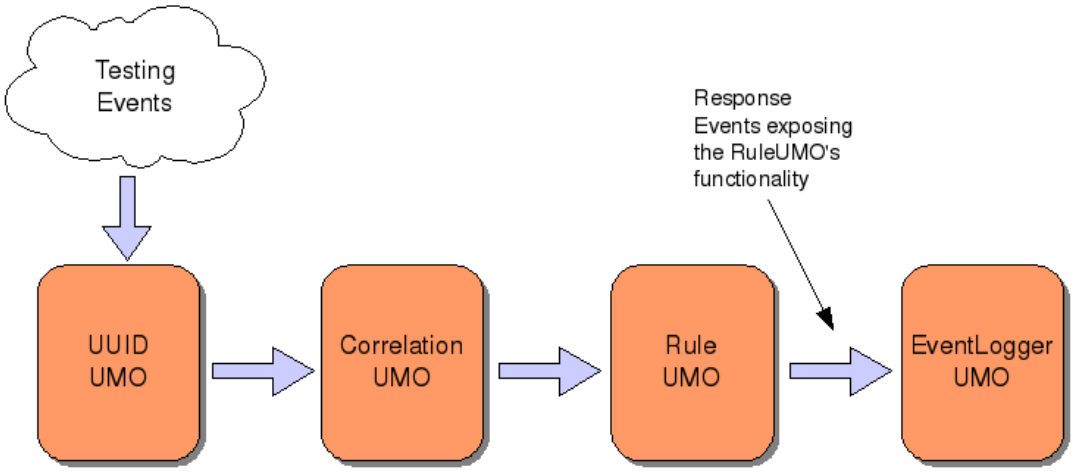

Figure 6.4: Naiad's Mule setup for Unit Testing

Luckily there is a way to automatically test this functionality with JUnit Tests too. Figure 6.4 shows an overview of the setup used for this. As only the UMOs and their connections are relevant, other parts, like Mule Agents, have been omitted in the figure.

Key to this approach is the `EventLoggerUmo`. In this class all fields and methods are defined as `static`, thus allowing this class to bypass any duplicates Mule, following the SEDA principle, might create. Whenever a Response Event leaves the `RuleUmo` it is sent to the `EventLoggerUmo` where it is stored.
A JUnit Test would then test some functionality of Naiad Rules as following:

- Start an instance of a Mule server configured as shown in figure 6.4. The configuration of the `RuleUmo` holds only very few rules and Response Events, that test for some specific behavior.
- Send a small number of events to the server. Those events have been designed to trigger respectively not to trigger the rules loaded into the `RuleUmo` in a specific way.
- Fetch all events from the `EventLoggerUmo` that have been stored there since the server started. Examining those events a test can now automatically decide if it has succeeded or failed.
- Shutdown the server.

Needless to say, that the set of rules and the event cloud used for a test have to be selected very carefully. Otherwise expected behavior in terms of observed Response Events could occur accidentally.

6.5 Bottlenecks

The most critical bottleneck for Naiad Rules - and probably for Naiad in general - is the Session Manager. To be precise, the (de)serialization that is done whenever a session is checked out or back in, which is very often. Because Drools' WorkingMemories are stored as attributes of a Session, they have to be (de)serialized just like the rest of the session. Additionally, a WorkingMemory of Drools does not implement Java's `Serializable` interface. Thus, simply put, "deserializing" is more like "creating a new WorkingMemory and consigning properties", which makes that procedure even more time-consuming than true (de)serialization of all inserted facts (!) would be in the first place.

Another theoretical bottleneck is, that the Rule Agent and the REG are implemented as Mule Agents and therefore shared between all RuleUMOs. Luckily this is not too critical in practice, because there are almost no `synchronized` operations between those agents and the UMOs. So calls from an UMO to an agent in most cases still run in the UMO's thread without blocking or otherwise affecting threads of other UMOs.
Additional care must be taken, when RuleUMOs are to be distributed among multiple virtual machines. In that case each of those virtual machine must have its Rule Agent(s) and REG(s).

At the moment, distribution is only a theoretical topic for further research anyways, because the central bottleneck, the Session Manager, still lacks support for that. Further, adequate Mule Transformers have to be written yet, to send events across the boundaries of one virtual machine. That however is a trivial task, once the Session Manager supports distribution.

6.5.1 Using Multiple Session Managers

The importance of the (de)serialization problem grows, when one considers, that it is not only the RuleUMOs that check out sessions. Pretty much any service in the CEP system, existing or yet to be developed, relies on or works with the correlated sessions - and, by the way, the correlation service itself does too. That means, that whenever a session is checked out by *any* service, *all* attributes of that session, whether they are used by that service or not must be (de)serialized in the process. Further, the bigger the number of services is, the more likely it is that a session is already checked out and a service is blocked until it is checked in again, resulting in even more performance loss.

The first approach that came to our minds was some mechanism that allowed the check-out of only certain attributes of a session. This way access to sessions would only block thread if already checked-out attributes would be requested by that thread. Further, not all attributes would have to be (de)serialized.

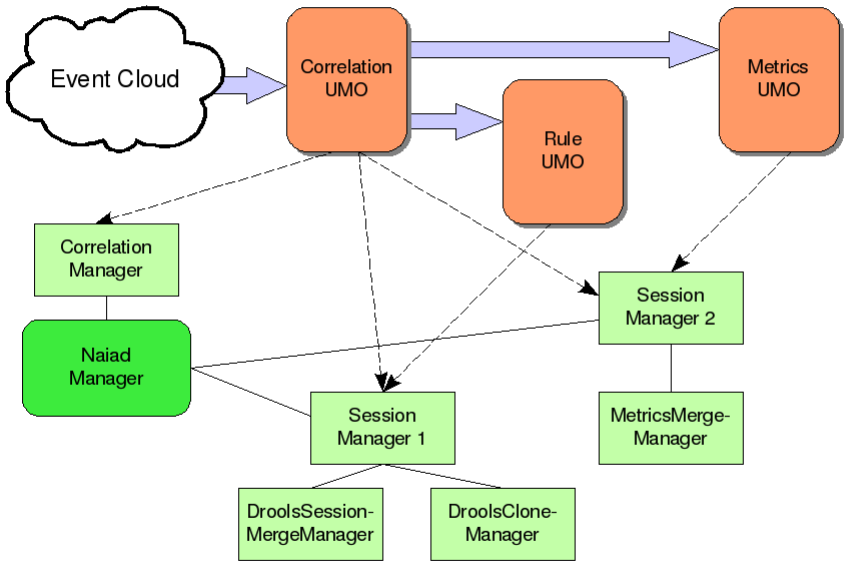

Figure 6.5: Naiad architecture using multiple Session Managers

That suggestion induced too many new questions about concurrency and integrity of sessions and threads, and was quickly rejected.

Instead Marian Schedenig came up with the idea of using multiple Session Managers with the same Correlation Manager, each one holding the attributes of just a few or maybe even just one service. Figure 6.5 gives an example. Please note that components that are actually necessary for Naiad Rules, but that are of no use for the example have been omitted.

In this example, we have some other metrics computing service (MetricsUMO) besides our RuleUMO. They work on the same set of correlations, but do not share any of their own ressources. When a new event enters the CorrelationUMO and its CorrelationMatches have been identified, those matches are sent to *two* Session Managers instead of only one. Up to that point both Session Managers hold identical sets of sessions. When the events propagate further through the system, they eventually arrive at the RuleUMO and the MetricsUMO. Each one of these can now use its own Session Manager and thus its own sessions to store attributes, without blocking the

other service or having to wait for the (de)serialization of attributes that are of no use to it.

What makes that concept so attractive is, that neither the Session Managers nor the RuleUMO and MetricsUMO know anything about it. In Naiad UMOs get references to their Session Managers via injection anyways and Session Managers do not know anything about "the outside world" at all. The only implementational work that needs to be done, is to enable the CorrelationUMO to work with more than one Session Manager - which is trivial. Besides that, it only has to be taken care that every Session Manager is hooked up with the proper Clone- and Merge-Managers for the attributes that will be stored in its sessions. Configuration editors, which have to be developed yet, could do that.

Additionally we gain some sort of distributability with this, even though, as mentioned, things are not truly distributable yet. The price we pay naturally is slightly more usage of memory as well as some overhead in transmission. And because so many components and only a few settings have to be modified, this can be easily applied to existing Naiad configurations.

7 Results and Findings

7.1 Development

Naturally it took some time and a couple of design-evaluate-redesign cycles to develop the whole basic concept of Naiad Rules and how it should be integrated into Naiad's services. After that task was finished however, development of the architecture in detail and coding of Naiad Rules was finished surprisingly fast.

Because Naiad Rules is more like a *wrapper* for a conventional rule engine, rather than a piece of *working* code, only 4 interfaces and 2 classes, which are relatively simple in their design, make up the generic part of it[1]. The other parts are specific to the integration of Drools and the Velocity template engine. The biggest part of those is the loading and parsing of Naiad Rules' XML configuration and its transformation to Drools rules. This part was the most time consuming of all planned steps of implementation.

The most time-consuming task however, was finding out about and tracing a bug in Drools and writing a workaround for it. The second most time-consuming part was the writing of the dynamic generation of bytecode for facts of incoming events using ASM. To be honest, it was not really the writing itself, but the study about how to write bytecode for the Java Virtual Machine.

To conclude this with a final statement: Development time for the integration of a conventional rule engine into a CEP application was short - certainly *much* shorter than any from-the-scratch solution. But it could have been even shorter, if it were not for problems and obstacles due to the chosen rule engine. To be fair, any other rule engine probably would have come with its own set of equally time-consuming oddities and challenges.

7.2 Configurability

The configuration of Naiad Rules and the Velocity based Response Event Generator has shown to be very easy, fast and powerful. It proved to be very easy to adjust the rule set to meet changing business requirements too. At least for somebody who is familiar with the features, options and syntax. Unfortunately this is the problem.

Because unless you spent a couple of hours studying Naiad's manual and trying out some sample-configurations, you just can not be familiar with it. All those interleaved options and parameters contained in an existing set of rules presented to somebody not

[1] Those numbers include all code for the Response Event Generator

familiar with them are just too confusing - not even speaking of creating a completely new set of rules from the scratch. Additionally the nature of Naiad Rules is just too far away from anything else a normal user is familiar with, so learning by trying and intuition is not an option either.

This once more stresses the importance of a graphical editor that can guide new users through creating rules and assists them in the process.

7.3 Performance

Naiad Rules and its performance was tested with live data from the SAW simulator (see section 1.5). The SAW team provided a JMS[2] connector, where a running instance of the simulator's Runtime Interface would publish all events which were useful for further processing using CEP.

However, to be independent from varying consumption of system resources by the simulator (if both applications were run on the same computer) or varying network bandwith (if both applications were run on different computers) and to be able to test in reproducible environments, events from various testruns of the SAW simulator have been recorded to files. A light-weight "event-player" was written, that was able to read those files and send the events they contained to Naiad per JMS. The speed of the playback could be modified, so that real-time, sub-real-time and super-real-time loads could be simulated.

For the specifications of the testsystem used refer to appendix C.

First, as a most simple scenario, Naiad Rules' performance as a filter (see section 5.1) has been tested. With only one rule filtering for one event type only, Naiad was able to process approximately 250 incoming events per second at a constant rate. When adding another filter, that rate dropped to roughly 200 events per second.

Interestingly, adding some more filter rules did not minimize that throughput anymore. Adding some Literal Constraints to those rules did not either. Further examination of the recorded data revealed, that the limiting factor for the testsetup, which was similar to the one shown in figure 6.2, was the CorrelationUMO, not the RuleUMO. Even the use of quite "creative" but realistic filter rules did not increase the RuleUMO's consumption of resources to a level where it would have become the limiting factor and decrease overall throughput.

Even if the number of handled events per second would decrease with more and more complex filter rules being added, it still would be considerably larger than the number of events[3] generated by an average assembly workshop.

[2] Java Message Service - `http://java.sun.com/products/jms/`

[3] That is events provided through the simulator's JMS connector, not all its internal events.

However, filtering is more or less just a supporting feature of complex event processing. Therefore after testing Naiad Rules with those simple scenarios, a somewhat more complex one was chosen: Like figure 5.2 shows, a map should be implemented, that integrated `evtWorkpieceIn` and `evtWorkpieceOut` events into new `evtWorkpieceProcessed` events (see listing B.2).

Surprisingly, doing this unveiled another performance issue with a more conceptual nature, rather than a technical one. It is described in section 7.3.1.

7.3.1 The completenessTimeout Problem

If you look at figure 5.2 it is clear, that for achieving a map like that, correlating all `evtWorkpieceOut` and `evtWorkpieceProcessed` events of the same machine does not do the trick: during a working shift, there is not only one pair of `evtWorkpieceOut` and `evtWorkpieceProcessed` events per machine, but hundreds if not thousands. If they all were correlated into the same session, it would not be possible to find those pairs that belong together.

Things were a lot easier, if the events contained some ID of the workpiece that they are about. Then we could correlated by the `componentName` (ID of the machine) *and* the ID of the workpiece. Unfortunately this is not what we can get from the SAW simulator (see listing 3.1) and we would end up with a lot of memory-consuming sessions anyways. But luckily Naiad offers a feature for just that case - at least sort of.

Each Correlation Item (see section 3.3 about how to configure correlations in Naiad) can specify a `minOccurence` attribute. If specified, Naiad counts the occurrences of this Item per session and marks the session `COMPLETE` once the count equals the specified value.

Sessions with this mark are then deleted after a certain amount of time, which is specified in a surrounding Correlation Set's or Correlation Bridge's `completeness-Timeout` attribute.

Listing 7.1 shows an example. All sessions created from the `workingProcess` Correlation Set are deleted two seconds after a `evtWorkpieceOut` event was added to the session.

Now this sounds exactly like what we were looking for: our mapping rule can rely on the fact, that all events in a session not only belong to the same machine, but to the same workpiece, because if a session is deleted after the workpiece has left the machine a new one is automatically created when the next workpiece enters the machine. Further, sessions get deleted after a workpiece has been processed, and don't clutter up our precious memory.

The problem with this lies in the detail of high-frequency domains in terms of incoming events per time unit. Given the testsetup similar to the one in figure 6.2 it is clear, that the interval for the Completeness Timeout must not be shorter than the

```
1 <correlationSet identifier="processingOfWorkpiece"
     completenessTimeout="2">
2   <correlationItem identifier="workpieceOut" eventType="
       evtWorkpieceOut" minOccurence="1">
3     <xPathSelector mandatory="true">//payload[@key='
         ComponentName']</xPathSelector>
4   </correlationItem>
5   <correlationItem identifier="workpieceIn" eventType="
       evtWorkpieceIn">
6     <xPathSelector mandatory="true">//payload[@key='
         ComponentName']</xPathSelector>
7   </correlationItem>
8 </correlationSet>
```

Listing 7.1: Configuration of a Correlation Set with Completeness Timeout

time it takes Mule to propagate an event from the CorrelationUMO to the RuleUMO plus the time it takes the RuleUMO to process the event. If it were shorter, the session would have been deleted even before the RuleUMO would try to check it out.

On the other hand, the interval must not be too long either. In this case, subsequent events might be added to a session they don't belong to, because that session was not deleted yet.

As already denoted, it strongly depends on the domain at hand, whether the Completeness Timeout could be a problem or not. That is because the relevant factor is not the frequency of incoming events in general, but the frequency of incoming events, that accidentally might be added into the same session although they do not belong together.

In the SAW domain this definitely *is* a problem. If the strategy used to manage a a workshop aims to be optimal, machines must not have long idle times. That means, whenever a workpiece leaves a machine, the next one might and should enter it immediately afterwards. Candidate events for erroneous correlation therefore are `evtWorkpieceIn` events from one workpiece and `evtWorkpieceOut` events from another workpiece that just left a machine.

Naiad supports values for Completeness Timeouts only in steps of seconds. This makes sense, because due to the non-deterministic nature of SEDA and Mule (see section 2.2) any smaller value would be a mere lottery. Nevertheless, for testing that restriction was removed.

First, using a correlation configuration as shown in listing 7.1 and a mapping rule as shown in figure 5.2, the minimum value for the Completeness Timeout of the `processingOfWorkpiece` Correlation Set was determined, while simulating a relatively small load of one incoming event per second. A Completeness Timeout of 100 milliseconds was possible with that configuration at a constant rate. That is, no in-

coming event took more than that time to be processed from the CorrelationUMO to and including the RuleUMO.

Then, gradually increasing the number of incoming events per second, the maximum number was identified: 2.5 events per second was all that could be processed by Naiad without getting caught in the timeout trap and mixing up sessions. It should be noted, that relying on those numbers would be far to risky for any real-life production environment, as they are too close to error-causing settings to allow for any variance in a computer's available resources.

This unfortunate bottleneck is solely caused by Naiad's handling of correlations and sessions. Though the RuleUMO naturally takes its own share of a system's ressources, they are negligible small in comparison. Stand-alone tests of the RuleUMO suggest, that throughputs close to the 200 events per second of Naiad Rule's application as an event filter would be possible, if there were no external handicaps like the one presented in this section.

One promising and possibly quite simple solution to the problem could be, if the CorrelationUMO just ignored any sessions marked `COMPLETE`. This way a bigger Completeness Timeout interval could be set, which makes sessions available to the RuleUMO even if they need more time to be propagated through all UMOs by Mule. On the other hand such a big interval would not be a problem for subsequently incoming `evtWorkpieceIn` events, as the CorrelationUMO would create a new session for them. However, this would be a rather big interference with existing code and should be evaluated carefully before doing any rash changes.

What would be left, is the risk of one `evtWorkpieceIn` event "overtaking" a foregoing `evtWorkpieceOut` event in parallel SEDA instances of the same CorrelationUMO. However, *much* smaller intervals between incoming events and a huge load on the system (which would lead to an increase of the number of instances of the CorrelationUMO as well as a greater variance in processing intervals of events in that UMO) would be needed to make such a situation as likely as the problem regarding Completeness Timeouts.

8 Conclusion

Virtually all business processes in today's world can be seen as event-generating and -consuming activities. The role of Complex Event Processing, when trying to handle those vast amount of events produced by modern computer systems, cannot be underestimated.

The assumption was formulated, that the application of conventional rule engines onto pre-correlated sessions of events could be a powerful way to extend an existing correlation engine's possibilities with a minimum of implementational work.

The SARI model (section 2.4) is a basic foundation for the functioning of what is developed as Naiad Rules in this work. Easy to follow step-by-step instructions can be used to translate rules of that concept into rules that a conventional rule engine can understand and handle. Though it was planned from the start to use a specific engine, JBoss Drools [9], those instructions are generic enough to be applied to virtually any rule engine currently available, as long as it follows an Event-Condition-Action (ECA) approach.

Although not all details of SARI Rules have been implemented into Naiad Rules, the key benefits as identified by Rozsnyai in [23] were achieved:

- The nature of Naiad Rules facilitates the use of a graphical and thus easier to comprehend model.
- Preconditions allow the combination of multiple rules, thereby enabling their reuse for a gain in clarity and performance.
- The service oriented approach of Naiad enables easy integration of new services that depend on Naiad Rules or that Naiad Rules can depend on.

The lack of a graphical editor is of course the biggest shortcoming towards the full exploitation of all benefits. But as this has been kept in mind throughout the whole development phase of Naiad Rules, it is possible to create such an editor without requiring any modifications on Naiad Rules itself.

The SARI Model incorporates some shortcomings with ambiguous definitions of events in certain rule patterns. Those are resolved by introducing Ancestor Identifiers to Naiad Rules. They are a convenient and easily configurable way to distinctively refer to specific events in a theoretically endless chain of preceding rules. Being completed by this new concept, Naiad Rules is now powerful enough to handle a lot of real-life business situations out of the box, that is without requiring additional programming.

Hence more power has been given from the programmers to technically less experienced business analysts and domain engineers. This is even amplified by the fact, that the setup or modification of Naiad Rules' configuration is very quickly done - even without some intuitive graphical editor.

With some creativity Naiad Rules can be utilized to implement common features of CEP too, such as Filters, Maps, Event Processing Networks or Distributed Event Detection. It has to be admitted however, that all those patterns can only be achieved implicitly, through tricky combination of certain rules, which makes greater demands on the personnel trying to configure Naiad. Here, again, an adequate editor tool could be of great value as it could act as a translator between the explicit definition of CEP patterns and their implicit configuration in Naiad.

There are of course features that cannot be implemented, though there are promising approaches to make that possible with probably only minor enhancements to the core concept.

As a proof of concept, the ideas of Naiad Rules were successfully implemented into Naiad. Though some unforeseen problems needed to be resolved, it was shown that, just like it was assumed, this approach still was *significantly* faster to implement than any solution built from the scratch. It is totally non-intrusive, meaning that no modifications whatsoever (except for bugfixes) had to be made to the existing core of Naiad.

Naiad Rules has proved to be fast enough for many standard business situations. Unfortunately the correlation- and session-handling of Naiad itself constitutes a problematic bottleneck to the overall performance of the system, while the rule-handling part itself consumes comparatively few resources. Tests revealed that for certain domains and situations Naiad's performance is more than disappointing.

In the end it is naturally inherent to the presented solution, that it can never achieve the same grade of performance as any specifically developed solution. But that is the same with any generic concept in information technologies.

Nevertheless and because Naiad and Naiad Rules offer a couple of options for the distribution of their services, it presumably is fast enough for the greatest part of all common business situations. And even with costlier and thus faster hardware it probably still is the cheaper alternative.

Within their relatively low cost of implementation, setup and maintenance lies the biggest advantage of Naiad and Naiad Rules, as they provide a rich spectrum of possibilities out of the box nevertheless. Being simple to be set up for a specific domain in the first place, but offering space for further enhancements, they can serve as both, a quick and painless entry to the world of Complex Event Processing or a powerful tool in the struggle to conquer daily growing avalanches of data.

9 Outlook

9.1 Further Research Topics

Currently neither Naiad nor any of its core services are able to save their current state and "go to sleep". Solving this **initialization- and restart-problem** is the main key for crash recovery strategies, which are vital for many real-life usage scenarios.

Related to that is the **modification of rules at runtime**. Though Drools supports the modification of its rule sets on the fly, this can not be leveraged by Naiad Rules, because some way to handle all those "meta-constraints" (see section 4.2) in a predictable and reasonable way is yet to be found.

Though SARI Rules were targeted to be configured with a **graphical editor**, such a tool is still missing for Naiad Rules. Besides reducing the error-proneness of manually messing with XML documents, it could provide helpful features like the basic verification of a user's input and auto-completion. Further it would help to understand and manage large sets of rules much faster than without any graphical preparation. If you remember section 1.4.2, this was one of the key requirements on a CEP application.

Depending on the rules and the number of incoming events per time-unit, events might arrive out of order when their order is important. Even if they do arrive in order, their order might change due to the SEDA [38] model implemented in Naiad: Because some events might take more time to be processed than others, one event might "overtake" another one while they are in the same UMO, but in different threads. So currently Naiad can only provide *Best-Effort Detection*, but not ***Guaranteed Detection*** [16] of sequences of events. The latter however could be helpful or even necessary in a couple of situations.

In [12] Luckham presented ***parents of rules***. Those are like data-containers able to hold a *state*, which can be accessed and *modified* by a rule. Using Drools, this could be easily implemented using Drools' concept of globals, with the WorkingMemory being the parent of a rule. What still needs to be developed is a convenient way of how to express the definition of and the access to data objects in a rules parent in Naiad Rules' XML configuration.

However it is done, this feature would dramatically increase the potential of what can be done with Naiad Rules (see section 5.5.1), and possibly enable the calculation of metrics at the same time. Sums, averages, maximum/minimum values - they all require some kind of state for their calculation. Vecera [34] suggested to use the

correlation session as temporary storage for metric calculation. This is an alternative to using Drools' WorkingMemory, with some advantages and disadvantages, that have to be evaluated yet.

The count of some kind of event is another valuable information in a lot of domains. For example the count of error messages within a certain time interval [19]. This could be implemented with a stateful parent as well.

Being able to handle events that contain **sets of attributes** rather than only simple name/value-pairs would greatly improve Naiad's power as well. Section 5.5.2 gave an example.

In [24] the concept of **inheritance of event types** is presented. This should not be too difficult to implement in a Java environment, that knows about such concepts out-of-the-box, but would add great value to Naiad's power. Drools is able to check for types of facts using Java's `instanceof` operator, which makes things even more simple. However, some implementational work still needs to be done to use that feature (see section 9.2), and once again a convenient way to configure this has to be found yet.

[24] presents some interesting similar concepts, the usage of which in Naiad could be evaluated as well after inheritance has been implemented.

The solution to the **retriggering problem** (see section 4.2) is not a very beautiful one. It is more a sacrifice that had to be made to enable the predictable and clean merging of WorkingMemories and thus of correlation sessions. Another solution, which does not prohibit the repeated triggering of rules, but would still prevent the accidental triggering on a merge, would give a lot more freedom to what domain engineers can do using rules. At best it would be totally configurable, whether a rule is allowed to trigger once, limitless or for a fixed amount of times.

9.2 Ongoing Development in Naiad

One major task for the near future is to make the services of Naiad truly distributable. For this the Session Manager as a core feature must be distributable, and adequate Mule transformers must be written that enable the transmission of events from one virtual machine to another. The latter is a trivial task. As for the Session Manager efforts are currently made to follow an approach of space-based computing.

As described earlier in section 4.2, the check for an event's type is transformed to just another constraint in a pattern. But the Rete algorithm defines a special kind of node, that checks for a fact's type and that is always the first node a fact has to pass in the network. In Drools those nodes are called *Object Type Nodes*, and utilize Java's `instanceof` operator, which might make things a little faster. Dynamically creating bytecode for one Java class per event type is not too difficult, but as mentioned in section 6.1.2 quite complex and cumbersome, and has therefore be postponed so far.

Currently the calculation of metrics in Naiad is only possible with custom-written, for most parts non-reusable services. A reusable solution that could be configured without any programming skills and maybe with some graphical editor would be a very valuable addition to Naiad's core services. Studies are already made towards that direction.

Bibliography

[1] Apache Software Foundation. Apache Velocity. `http://velocity.apache.org`, September 2008.

[2] Apache Software Foundation. Lucene Webpage. `http://lucene.apache.org`, September 2008.

[3] Kent Beck, Erich Gamma, et al. JUnit. `http://www.junit.org`, October 2008.

[4] Shyh-Kwei Chen, Jun-Jang Jeng, and Henry Chang. Complex Event Processing using Simple Rule-Based Event Correlation Engines for Business Performance Management. In *CEC-EEE '06: Proceedings of the The 8th IEEE International Conference on E-Commerce Technology and The 3rd IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services*, page 3, Washington, DC, USA, 2006. IEEE Computer Society.

[5] James Clark and Steve DeRose. XML Path Language (XPath). `http://www.w3.org/TR/xpath`, November 1999.

[6] EsperTech Inc. Esper. `http://esper.codehaus.org`, January 2008.

[7] Charles Forgy. Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. *Artif. Intell.*, 19(1):17–37, 1982.

[8] Charles Lanny Forgy. *On the efficient implementation of production systems.* PhD thesis, Pittsburgh, PA, USA, 1979.

[9] JBoss Labs. Drools Webpage. `http://labs.jboss.com/drools/`, January 2008.

[10] Ralph Kimball and Joe Caserta. *The Data Warehouse ETL Toolkit.* John Wiley & Sons, 2004.

[11] David C. Luckham. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Ordering of Events. In *DIMACS Partial Order Methods Workshop IV*. Princeton University, 1996.

[12] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[13] David C. Luckham. Whats the Difference Between ESP and CEP? Online Article, August 2006.

[14] Microsoft. .NET Webpage. `http://www.microsoft.com/NET/`, September 2008.

[15] MS Analog Software kb. Rulecore Webpage. `http://www.rulecore.com`, January 2008.

[16] Gero Mühl, Ludger Fiege, and Peter R. Pietzuch. *Distributed Event-Based Systems*. Springer, 2006.

[17] Mule Source Inc. Mule Open Source ESB Webpage. `http://mule.mulesource.org/display/MULE/Home`, January 2008.

[18] Yefim Natis. Service-Oriented Architecture Scenario. Technical Report AV-19-6751, Gartner Research, April 2003.

[19] Wu Peng, R. Bhatnagar, L. Epshtein, M. Bandaru, and Shi Zhongwen. Alarm Correlation Engine (ACE). In *Network Operations and Management Symposium, 1998. NOMS 98., IEEE*, volume 3, pages 733–742, 1998.

[20] PostgreSQL Global Development Group. PostgreSQL Webpage. `http://www.postgresql.org`, September 2008.

[21] Chris Reade. *Elements of functional programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[22] Szabolcs Rozsnyai. Efficient Indexing and Searching in Correlated Business Event Streams. Master's thesis, Vienna University of Technology, 2006.

[23] Szabolcs Rozsnyai. *Managing Event Streams for Querying Complex Events*. PhD thesis, Vienna University of Technology, 2008.

[24] Szabolcs Rozsnyai, Josef Schiefer, and Alexander Schatten. Concepts and models for typing events for event-based systems. In *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 62–70, New York, NY, USA, 2007. ACM.

[25] Marian Schedenig. Leveraging ESB and SEDA Technologies for Complex Event Correlation. Master's thesis, Vienna University of Technology, 2008.

[26] Josef Schiefer and Carolyn McGregor. Correlating Events for Monitoring Business Processes. In *ICEIS (1)*, pages 320–327, 2004.

[27] Josef Schiefer, Szabolcs Rozsnyai, Christian Rauscher, and Gerd Saurer. Event-Driven Rules for Sensing and Responding to Business Situations. In *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 198–205, New York, NY, USA, 2007. ACM.

[28] Josef Schiefer and Andreas Seufert. Management and Controlling of Time-Sensitive Business Processes with Sense & Respond. In *CIMCA/IAWTIC*, pages 77–82, 2005.

[29] Senactive GmbH. Senactive GmbH Webpage. `http://www.senactive.com`, January 2008.

[30] SpringSource. Spring Webpage. `http://www.springframework.org`, September 2008.

[31] Sun Microsystems. Java Webpage. `http://java.sun.com`, January 2008.

[32] The Perl Foundation. PERL Webpage. `http://www.perl.org`, September 2008.

[33] R. Vaarandi. SEC - A Lightweight Event Correlation Tool. 2002.

[34] Roland Vecera. Efficient Indexing, Search and Analysis of Event Streams. Master's thesis, Vienna University of Technology, 2007.

[35] Vienna UT Information & Software Engineering Group. IFS Webpage. `http://www.ifs.tuwien.ac.at`, September 2008.

[36] P. Vrba. MAST: manufacturing agent simulation tool. *Emerging Technologies and Factory Automation, 2003. Proceedings. ETFA '03. IEEE Conference*, 1:282–287 vol.1, Sept. 2003.

[37] Pavel Vrba and Vladimir Marik. Simulation in agent-based control systems: MAST case study. Technical report, Rockwell Automation Research Center, 2005.

[38] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP 2001)*, pages 230–243. ACM Press, October 21–24 2001.

A Configuration of Naiad Rules with Drools and Velocity

A.1 Connecting Naiad Rules to the Correlation Server

Before any rules can be processed, one or more rule agents have to be instantiated and connected to Naiad's Correlation Services in the Mule configuration as shown in figure 6.2. Listing A.1 exhibits a minimal but complete sample configuration of Mule, including all agents, connectors, transformers and UMOs necessary for a working Naiad server.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <mule-configuration id="EventServer_SAW" version="1.0">
3
4    <description>
5     This mule configuration file configures all necessary umo, transformer and
          endpoints to be used by the event server
6    </description>
7
8    <agents>
9      <agent name="naiadManager" className="at.ac.tuwien.ifs.naiad.core.
           NaiadManager">
10       <properties>
11         <property name="configFile" value="naiad.xml"/>
12       </properties>
13     </agent>
14     <agent name="ruleAgent" className="at.ac.tuwien.ifs.naiad.core.rule.drools.
           DroolsRuleAgent">
15       <properties>
16         <property name="configFile" value="naiadRules.xml"/>
17       </properties>
18      </agent>
19      <agent name="responseEventGenerator" className="at.ac.tuwien.ifs.naiad.core
            .rule.velocity.VelocityResponseEventGenerator">
20        <properties>
21           <property name="typeXPath" value="//@type"/>
22        </properties>
23     </agent>
24   </agents>
25
26   <connector name="jmsConnector" className="org.mule.providers.jms.JmsConnector"
         >
27     <properties>
28       <property name="connectionFactoryJndiName" value="ConnectionFactory"/>
29       <property name="jndiInitialFactory" value="org.activemq.jndi.
           ActiveMQInitialContextFactory"/>
30       <property name="specification" value="1.1"/>
```

```
31        <map name="connectionFactoryProperties">
32          <property name="brokerURL" value="vm://localhost"/>
33          <property name="brokerXmlConfig" value="classpath:activemq.xml"/>
34        </map>
35      </properties>
36    </connector>
37
38    <transformers>
39      <transformer name="ActiveMQTransformer" className="at.ac.tuwien.ifs.naiad.
          core.transformer.ActiveMQToXMLEvent" returnClass="at.ac.tuwien.ifs.naiad
          .core.event.IEvent">
40        <properties>
41          <property name="XPath" value="//@type" />
42        </properties>
43      </transformer>
44    </transformers>
45
46    <model name="SAW_EventServer">
47
48      <mule-descriptor name="UuidUmo" implementation="at.ac.tuwien.ifs.naiad.core.
          umo.UuidUmo">
49        <inbound-router>
50          <endpoint address="jms://MyDestination" transformers="
              ActiveMQTransformer"/>
51        </inbound-router>
52        <outbound-router>
53          <router className="org.mule.routing.outbound.OutboundPassThroughRouter">
54            <endpoint address="vm://uuid"/>
55          </router>
56        </outbound-router>
57      </mule-descriptor>
58
59      <mule-descriptor name="CorrelationUmo" implementation="at.ac.tuwien.ifs.
          naiad.core.umo.CorrelationUmo">
60        <inbound-router>
61          <endpoint address="vm://uuid"/>
62        </inbound-router>
63        <outbound-router>
64          <router className="org.mule.routing.outbound.MulticastingRouter">
65            <endpoint address="vm://correlated"/>
66          </router>
67        </outbound-router>
68        <properties>
69          <property name="correlationManagerId" value="correlationManager"/>
70          <property name="sessionManagerId" value="sessionManager"/>
71        </properties>
72      </mule-descriptor>
73
74      <mule-descriptor name="RuleUmo" implementation="at.ac.tuwien.ifs.naiad.core.
          umo.RuleUmo">
75        <inbound-router>
76          <endpoint address="vm://correlated"/>
77          <endpoint address="vm://stateChange"/>
78        </inbound-router>
79        <outbound-router>
80          <router className="org.mule.routing.outbound.OutboundPassThroughRouter">
81            <endpoint address="vm://ruleResponseEvents"/>
82          </router>
83        </outbound-router>
84        <properties>
85          <property name="sessionManagerId" value="sessionManager"/>
86          <property name="ruleAgentId" value="ruleAgent"/>
```

```
87          <property name="responseEventGeneratorId" value="responseEventGenerator"
                />
88        </properties>
89      </mule-descriptor>
90
91    </model>
92
93  </mule-configuration>
```

Listing A.1: Mule configuration for a sample Naiad server

Three parts of listing A.1 are important for Naiad rules:

- The **RuleUmo**, which listens for events leaving the CorrelationUmo and status-events, and which publishes Response Events on the `vm://ruleResponseEvents` endpoint. The names of the Session Manager, Rule Agent and Response Event Generator it should use are injected here too.
- The **Rule Agent** that loads and manages a set of rules. All configuration of rules is done in the separate file `naiadRules.xml`, which is explained in detail in the next section.
- The **Response Event Generator**, implemented as Mule Agent. Because it creates `IEvent`s, it must know a XPath selector which points to the type of a Response Event within its XML representation.

A.2 Defining Rules

All rule-related configuration in Naiad is done in a file separate from any other settings. This helps a little to keep things organized. Further, if multiple Rule Agents are configured each one has its own file for configuration, facilitating quick distribution among different virtual machines at a later point in time.

The general skeleton for Naiad Rules' configuration is shown in listing A.2.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ruleAgent>
3
4    <xPathMapping>
5      <!-- All XPathMappings go here. 3 examples follow: -->
6      <mappingItem xPath="//payload[@key='WorkpieceId']" fieldName="workpieceId"
           type="string" />
7      <mappingItem xPath="//@timestamp" fieldName="timestamp" type="date" />
8      <mappingItem xPath="//@id" fieldName="eventId" type="integer" />
9    </xPathMapping>
10
11   <!-- All definitions of rules and Response Events go here -- >
12
13 </rule>
```

Listing A.2: General skeleton for Naiad Rules' configuration

Listing A.3 shows a sample definition of a rule (`myFirstRule`), that exhibits all features for such.

An AND-precondition depends on two other rules (`aRule` and `anotherRule`). Those are named `firstConditionAI` and `secondConditionAI` to be referenced in any Ancestor Identifiers of this rule.

The pattern named `productFinishedPattern` matches any event of type `evtProductFinished`, for it does not contain any constraints.

The pattern named `workpieceInPattern` references to any event that triggered the rule `aRule` and was identified as `workpieceInPattern` there as well, and matches the pattern's additional constraints.

Constraints can be nested in cascades of `andConstraintConnective`s and `orConstraintConnective`s.

Constraints can either be `literalFieldConstraint` comparing an event's fields to some constant, or `variableFieldConstraint`s comparing fields to a field of the same or another event. Whether the former or the latter is the case is determined by the absence of presence of a '.' in the `variableName` attribute.

```
1  <rule identifier="myFirstRule">
2    <precondition type="AND" identifier="cause">
3      <preconditionItem causingRule="aRule" identifier="firstConditionAI"/>
4      <preconditionItem causingRule="anotherCondition" identifier="
           secondConditionAI"/>
5    </precondition>
6    <pattern type="evtProductFinished" identifier="productFinishedPattern"/>
7    <pattern ancestorIdentifier="firstConditionAI.workpieceInPattern" identifier="
         workpieceInPattern">
8      <literalFieldConstraint fieldName="workpieceId" evaluator="==" value="1234"
           />
9      <andConstraintConnective>
10       <orConstraintConnective>
11         <literalFieldConstraint fieldName="workpieceId" evaluator="==" value="
               1234" />
12         <literalFieldConstraint fieldName="workpieceId" evaluator="==" value="
               5678" />
13       </orConstraintConnective>
14       <variableFieldConstraint fieldName="timestamp" evaluator="&lt;"
             variableName="productFinished.timestamp" />
15     </andConstraintConnective>
16   </pattern>
17 </rule>
```

Listing A.3: A sample rule definition in Naiad Rules' configuration

Listing A.4 shows a sample definition of a Response Event in Naiad Rules. For a Response Event, only a precondition and a template name have to be defined. The template name can be any string that can be interpreted by the connected Response Event Generator (see listing A.1 about how to connect a REG).

```
<responseEvent identifier="myResponseEvent" template="myResponseEvent.vm">
  <precondition type="OR" identifier="cause">
    <preconditionItem causingRule="aRule" />
    <preconditionItem causingRule="anotherRule" />
  </precondition>
</responseEvent>
```

Listing A.4: A sample definition of a Response Event in Naiad Rules' configuration

A.3 Defining Response Events

With the Velocity based Response Event Generator of Naiad Rules, Response Events are directly created from XML templates, where placeholders point to attributes of triggering events. Listing 6.7 has already shown a simple template.

The variable `$data` holds all preconditionItems of a Response Event. Those in turn hold all events that triggered the preconditionItem, identified by the matching pattern's identifier. Finally all xPath-mapped fields of those events can be accessed from them. Additionally the REG provides a variable named `$now`, which provides the current timestamp with an optional formatting feature.

Naturally the Velocity based REG could have created events of any form, not just XML, but with events represented as a XML document lots of already existing code in Naiad's core could be used.

B Configuration of Common Patterns

B.1 Filters

```
<event type="evtWorkpieceIn" timestamp="$data.contextPI.wpInPattern.timestamp">
  <payload key="palletId">$data.contextPI.wpInPattern.palletId</payload>
  <payload key="machineId">$data.contextPI.wpInPattern.machineId</payload>
</event>
```

Listing B.1: Configuration of the Velocity based REG for example in figure 5.1

B.2 Maps

```
<event type="evtWorkpieceProcessed" timestamp="$data.wpOutPI.wpOutPattern.
    timestamp">
  <payload key="palletId">$data.wpInPI.wpInPattern.palletId</payload>
  <payload key="machineId">$data.wpInPI.wpInPattern.machineId</payload>
#set($busyTime = $data.wpOutPI.wpOutPattern.timestamp - $data.wpInPI.wpInPattern
    .timestamp )
  <payload key="busyTime">$busyTime</payload>
</event>
```

Listing B.2: Configuration of the Velocity based REG for example in figure 5.2

B.3 Event Processing Networks

```
<event type="metaOutEvent" timestamp="$data.wpOutPI.wpOutPattern.timestamp">
  <payload key="machineId">1</payload>
  <payload key="product">D</payload>
#set($machine1cost = 10)
#set($cost = $data.aInPI.aInPattern.cost + $data.bInPI.bInPattern.cost + $data.
    cInPI.cInPattern.cost + $machine1cost)
  <payload key="cost">$cost</payload>
</event>
```

Listing B.3: Configuration of the Velocity based REG for example in figure 5.4

C Testsystem

The testsystem used for this work was a mobile notebook for personal use at the low end of up-to-date computers:

- Intel Centrino 1.7 GHz
- 1 GB RAM (512 MB reserved memory for the Java VM)
- 5400rpm HDD
- Operating System: Ubuntu 8.04.1 hardy with 2.6.24-21-generic i686 Linux kernel