



*Konzeption und Implementierung eines
leichtgewichtigen und autonomen
Regel-basierten Systems als eine
Realisierung von "Active Properties"
im Kontext von aktiven Dokumenten*

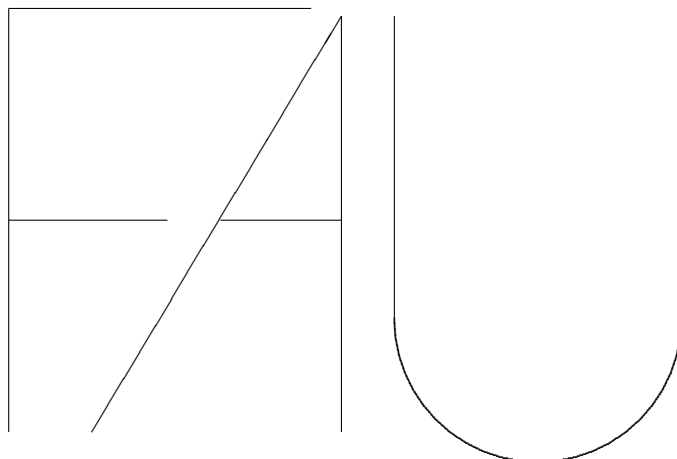
Diplomarbeit

Aneliya Todorova

Lehrstuhl für Informatik 6
(Datenmanagement)

Department Informatik
Technische Fakultät

Friedrich Alexander-
Universität
Erlangen-Nürnberg



Konzeption und Implementierung eines leichtgewichtigen und autonomen Regel-basierten Systems als eine Realisierung von "Active Properties" im Kontext von aktiven Dokumenten

Diplomarbeit im Fach Informatik

vorgelegt von

Aneliya Todorova

geb. 25.08.1983 in Burgas

angefertigt am

**Department Informatik
Lehrstuhl für Informatik 6 (Datenmanagement)
Friedrich-Alexander-Universität Erlangen-Nürnberg**

Betreuer: Univ.-Prof. Dr.-Ing. habil. Richard Lenz
Dipl.-Inf. Christoph P. Neumann

Beginn der Arbeit: 01.02.2010

Abgabe der Arbeit: 16.08.2010

Erklärung zur Selbständigkeit

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass diese Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Informatik 6 (Datenmanagement), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Diplomarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 16.08.2010

(Aneliya Todorova)

Kurzfassung

Konzeption und Implementierung eines leichtgewichtigen und autonomen Regel-basierten Systems als eine Realisierung von "Active Properties" im Kontext von aktiven Dokumenten

Zur Realisierung einer organisationsübergreifenden Prozessunterstützung im Gesundheitswesen wird ein Dokumenten-orientierter Ansatz favorisiert. In diesem Kontext beschäftigt sich diese Arbeit mit dem Integrationsentwurf eines leichtgewichtigen Regel-basierten Subsystems für aktive Dokumente. Aktive Dokumente stellen in sich geschlossene Einheiten dar, die die grundlegenden Bestandteile des Dokumenten-orientierten Ansatzes bilden. Das Subsystem, das die *active-properties* von einem aktiven Dokument implementieren soll, muss in der Lage sein das Dokument zu ändern und es zwischen lose gekoppelten, autonomen, heterogenen Teilsystemen zu verteilen. Der Akzent fällt auf die Konzipierung der *active-properties* anhand eines Regel-basierten Systems, das für das Triggern von Aktivitäten zuständig ist. Die Aktivitäten bezwecken die dynamische Gestaltung eines organisationsübergreifenden dezentralisierten Workflows durch das Ändern und Verteilen von Dokumenten.

Abstract

Design and Implementation of a Lightweight, Autonomous, Rule-Based System Which Realizes "Active Properties" in the Context of Active Documents

This thesis deals with the design of a lightweight subsystem, which realizes the active-properties in active documents in the context of a distributed and decentralized workflow approach. The subsystem is devised to react autonomously by utilizing a rule engine. The active documents are the self-contained artifacts of a document-oriented workflow approach, which combines the content-oriented and activity-oriented workflow paradigms in a hybrid solution. While the characteristics of this approach in the context of healthcare are explained, the focus is set on the conception and implementation of the active-properties subsystem as the autonomous decisive part of this workflow. The main functionality of the active-properties is the altering and distribution of the document they are part of, and hence the support for a dynamic process evolution within heterogeneous cross-organizational systems.

Contents

List of Abbreviations	ix
1 Introduction	1
1.1 Motivation and Challenges	1
1.2 The Role of the Active-properties	2
2 Methods	5
3 Basics	7
3.1 Distributed Systems	7
3.2 Process/Workflow	8
3.3 Active Documents	9
3.4 ECA-Paradigm vs. Inference Engines	9
3.5 Conclusion	10
4 Requirements Analysis	13
4.1 The Domain Model	13
4.1.1 The Artifacts: α -Doc and α -Cards	13
4.1.2 The Adornment Models	15
4.2 Overview: α -Flow Components	16
4.3 Functional Requirements Framework	20
4.4 Functional Requirements of the API	21
4.4.1 Motivating the Usage of a Rule Engine	21
4.4.2 Requirements of the API	21
4.5 Non-Functional Requirements	23
4.6 Conclusion	23

5	Rule Engines and JBoss Drools	25
5.1	Rule Engines	25
5.2	JSR-94	26
5.3	JBoss Drools	27
5.3.1	Subprojects Overview	27
5.3.2	Drools Expert	29
5.3.3	Drools Fusion	31
5.3.4	Drools Flow	32
5.3.5	Drools Guvnor	33
5.4	Conclusion	35
6	Proposed Solution: α-Properties	37
6.1	Architecture Overview	37
6.2	The α -Properties Interface	38
6.3	The α -VerVarStore Interface	39
6.4	The Update Service Interfaces	40
6.5	Conclusion	40
7	System Design: α-Properties	43
7.1	The α -Model	43
7.2	The α -PropsFacade Interface	45
7.3	The Events Classification	48
7.3.1	Application-specific Events	49
7.3.2	Technical Events	49
7.4	The Groups of Rules	51
7.4.1	Adding a New Content α -Card	52
7.4.2	Checking the Changeability of Adornments	55
7.4.3	Changing an Adornment	56
7.4.4	Changing Payload	57
7.5	Propagating Updates	58
7.6	Notification	60
7.7	Conclusion	61
8	Implementation Issues	63

8.1	The Rule Package	63
8.1.1	Queries	63
8.1.2	Globals	64
8.1.3	Functions	64
8.1.4	Rules	65
8.2	Distribution	67
8.3	Monitoring	69
8.4	Conclusion	70
9	Discussion	71
9.1	Delimitations in the Model	71
9.2	Prototype Optimization Options	72
9.2.1	Networking	72
9.2.2	Clearing up the Working Memory	72
9.2.3	Flow Features Elaboration	72
9.3	Rules Management, Dynamic Load of Rules and Rules Propagation . . .	73
9.4	Versioning and Variants Management	76
9.5	Participant Management	76
9.5.1	Access Control	76
9.5.2	Assignment of Tokens	77
9.6	Data Synchronization	77
9.7	Thread Synchronization and Race Conditions	78
9.8	Persistence	78
9.9	Network Security	79
9.10	Import and Export von "Process Templates"	79
10	Conclusion	81
A	The Rule Package	83

List of Figures

4.1	The α -Doc	14
4.2	The α -Flow Subprojects Overview	17
4.3	The Correlation between the α -Doc, the Editor, the Properties and the VerVarStore within the α -Flow	19
5.1	<i>JBoss Drools</i> System Components	28
5.2	Drools Guvnor: Architecture (<i>Source: [Dro10]</i>)	34
6.1	The α -Properties Architecture	38
7.1	The α -Doc and α -Card Models	44
7.2	The Payload Model	44
7.3	The <i>AlphaPropsFacade</i> Interface	45
7.4	The Initialization Methods	46
7.5	The Collaboration between the α -Properties, the α -Editor and the α -VerVarStore	48
7.6	The Event Model	50
7.7	The Addition of an α -Card	53
7.8	The Update Propagation Interaction	59
7.9	The Update Service Interfaces	59
7.10	The Event Listeners	60
8.1	The <i>Incoming</i> Pipeline	68
9.1	Dynamic Rule Loading Proposal	74

List of Tables

7.1	The Technical Events	51
7.2	The α -Props Groups of Rules	51
7.3	The Adornments: Default Values	54
7.4	The Changeability of the Adornments	55
8.1	The α -Queries	64
8.2	The Functions	64
8.3	The α -Props Rules	66
9.1	Dynamic Rule Loading Proposal: Options Comparison	75

List of Abbreviations

API	Application Programming Interface
USB	Universal Serial Bus
PDF	Portable Document Format
ECA	Event-Condition-Action
DBMS	Database Management System
DBS	Database Systems
EDA	Event Driven Architectures
DSL	Domain Specific Language
DRL	Drools Rule Language
JMS	Java Message Service
JAXB	Java Architecture for XML Binding
JPA	Java Persistence API
JTA	Java Transaction API
TSA	Treatment Structure Artifact
CRA	Collaboration Resource Artifact
JAXB	Java Architecture for XML Binding
BIRT	Business Intelligence Reporting Tool

JCR	Java Content Repository
GWT	Google Web Toolkit
Ajax	Asynchronous JavaScript and XML
BPMN	Business Process Modeling Notation

1 Introduction

IT support in healthcare is typically limited to intra-institutional processes. It lacks in patient data reconciliation and timely results upgrade throughout the therapy. Depending on the disease and its treatment the number of the involved parties is likely to rise, thus resulting in a complex and tedious communication among the participants. Therefore, an infrastructure is needed that reflects the patient's treatment process and the tracing of the attendant documents. In this thesis an approach for such a infrastructure will be presented focusing on just one part of it - the "active-properties". It is a part of the ProMed research project, which focuses on the distributed process support in medicine and healthcare.

1.1 Motivation and Challenges

The need of improvement in the healthcare information systems is inevitable. The classical paper-based treatment handling can no longer fulfill the arising requirements on such information systems. The lack of adaptability as well as the support of cross-organizational workflows have become important and challenging issues. The heterogeneous nature of the healthcare sector implicates distributed and demand-driven system solutions. At the same time, the autonomy of the participating parties must further on be granted. The conventional communication between the institutions involved in a treatment cycle does not offer enough flexibility and duly information availability. The escape of this traditional approach is required in order to move from the boundaries of the classical to the prospects and benefits of the evolutionary. The work in this directions promises a great amount of quality and efficiency increase in medical care ([CWW⁺06], [LR07]).

Treatment episodes often demand cross-organizational partner collaboration. But participants involved in a treatment process, as in the context of healthcare, have various origin. There are physicians from the primary sector cooperating with physicians from

the secondary healthcare sector and vice versa; health insurance funds and other reference physicians are also involved. They all need sufficient information about the status of the patient's treatment. Consequently, a decentralized approach for the inter-institutional process support in healthcare must be considered, which support multiple heterogeneous distributed participating systems.

Processes are the focus of developing such an infrastructure. Processes are suited to describe the overall control flow. But they tend to become very complex, that is why the scope of the required solution must be reduced in granularity. Shifting the control responsibility into the hands of the process actors themselves is considered. More specifically that would mean assigning control from one centralized management unit (or a single process participant, the initiator) to more fine-grained units like the artifacts that build the data units of exchanged information - the documents themselves. Consigning responsibilities and control to local components (á peer) has the advantages that specific circumstances and exceptional situations can be handled directly on site. Yet local decisions must be synchronized with the entire workflow as well. That's why local reactions must be distributed and taken care of on global basis.

Taking these challenges into account has led to the proposal to define long-running processes using rules, as rules can model behavior and reason about large diversity of data and events. The rules are incorporated into the artifacts themselves. This approach affects the entire workflow by shifting the control to the fine-grained level of documents. The distributed execution of rules completes the requirements.

1.2 The Role of the Active-properties

In this thesis a module that implements rule management within documents and contributes to their autonomous behavior is conceived. It is responsible for the local behavior of a document in the context of the distributed workflow. The concept of *active-properties* comes from an active document management system called *Placeless Documents*, which implements the idea of properties that "actively contribute to the functionality and behavior" of the documents they are bound to ([DEH⁺00]). The role of the active-properties in the context of a cross-organizational workflow is therefore to realize the navigation in

the process: it works its way forth in the workflow, is in charge of changes and controls access to the artifacts.

The active-properties are part of the active document and are responsible for its liveliness. They stand for a function or a particular behavior that is assigned to this document. They react upon changes, made inside the document or coming from the outside, by triggering response actions respectively. The active-properties can be implemented in various ways. Integrating a rule engine in it presents one possibility. These properties, which are alone responsible for the behavior of their residence, allow for an ordinary document to be autonomous. It thus helps itself through the whole process, by modulating it on the fly; without implying the exact actors and events to be known in advance.

In the context of healthcare, a document-based IT supported workflow is considered and active documents are favored for the fulfillment of its objectives. This thesis concentrates on the concept and implementation of such active-properties for document-based artifacts used as the units of information exchange in the healthcare sector during a patient's treatment.

2 Methods

In this thesis a possible design and implementation of active-properties in the context of active documents for a document-based workflow called α -Flow is proposed. The thesis is arranged in seven main chapters. Beginning with an overview of the basics, there will be given a brief description of what active documents are. In addition, the conventional Event-Condition-Action (ECA) paradigm known from active Databases and Event Driven Architectures (EDA) will be opposed to the concept of Inference Engines in order to compare both paradigms in search of a solution for the implementation of the active-properties. The α -Flow is a workflow proposal, whose main characteristic is its distributed heterogeneous nature. Hence, some basic features of workflows and distributed systems will also be explained.

Chapter 4 explains the artifacts that constitute the α -Flow as well as the prerequisites for the subsystem that is to be designed. In chapter 5 the JSR-94 specification for rule engines is outlined. A more detailed description of the JBoss Drools platform is given as well, explaining its components, architecture and principles. By offering a better understanding of how rule engines are designed, this outline motivates for utilizing one in the active-properties part of active documents, which are the subjects of the α -Flow approach.

In the following chapters 6, 7 and 8 the design and implementation of the prototype are described in detail. Chapter 6 illustrates the modules of the α -Flow and the place the active-properties have in it. The module that realizes the active-properties is called α -Properties. Its responsibilities, as well as the solution principles rule engines offer and the services expected to be provided are elaborated. The next chapter amplifies the system design of the α -Properties. Based on the proposed architecture and the declaration of the interfaces for the other modules, which depend on the α -Properties module, the proposed solution is detailed. In chapter 8 some selected details about the implemented prototype are explained. These include the application-specific rule package

conception and some further proposed solutions for the realization of the functional requirements concluded in chapter 4.

The last chapter deals with several points of extensibility for the implemented system. These issues concern the devised prototype as well as some further aspects that are considered important not only for the α -Properties but as a whole in the context of the α -Flow. Future work can build upon the efforts described in this thesis and the given reference implementation. A brief evaluation of the achieved goals is included in the conclusion.

3 Basics

In this chapter some basic concepts are introduced. An overview of distributed systems and their characteristics, a snippet of the traditional ECA paradigm in contrast to inference engines and some key features of active documents are represented. There are a bunch of approaches when it comes to organizing artifacts in workflows. These classical approaches will be contraposed to the hybrid character of the α -Flow concept.

3.1 Distributed Systems

A *distributed system* is represented by multiple autonomous computers, that communicate through a computer network. Its main characteristic is that all involved units are autonomous, but their interaction serves a common goal. Scalability and extensibility are important requirements for distributed systems. Scalability describes the capability of the systems to cope with continuously changing demands. Shared resource transparency and synchronization of exchanged information units rank among the features of distributed systems as well.

Communication and exchange of data take place among the involved units. Synchronization concerns arise regarding the latter. As Leslie Lamport describes in his paper such problems can be solved with the adoption of logical clocks [Lam78]. Lamport studied causality in distributed systems and eventually introduced the *Lamport timestamps* together with a general mechanism for managing replication. Summarized, a logical clock counts the order of events and is able to define a "happened-before" relationships between two events. An event is defined as the sending or receiving of a message by a process. Lamport defines that two events within the same process always have a fix order. If a message (event *a*) is sent by one process, than it is stated that *a* has happened before the receipt of that message (event *b*) by the other process. Events are assigned numbers, that are incremented in the order of their occurrence. While the innovative conclusions

of the Lamport timestamps offer a solution, synchronization and management of multiple replicas are still major issues in distributed systems.

Propagating updates of information about shared resources is an important feature of distributed systems. When designing a distributed system, some aspects in this regard must be considered. Such aspects are the format of the propagated changes, the policy of distribution and the definition of the set of the recipients. If a change occurs, there are the following possibilities to upgrade the rest of the concerned parties in a distributed system: 1) the altered object itself is sent out; 2) only a message, stating the invalidity of the old version, is sent; or 3) the operation procedure that led to the changes is propagated enforcing the same result on each existing replica. There two paradigms how such changes are distributed: either they are pushed to the others, or they get pulled from the source. Last but not least, the set of the affected peers needs to be defined: changes can be either broadcasted to all peers involved in the distributed system or multicasted to a selected subset of them.

Due to the complex nature of distributed systems some security issues arise to their requirements. Among others, integrity, availability and confidentiality are accentuated. Integrity defines that changes to assets (such as hardware, software and data) can be made only in an authorized way and improper alterations or access should be detected and taken care of. Confidentiality concerns the authorization and rights management of users, who manipulate these assets. The availability aims to assure that the peers or certain services they offer are constantly available.

3.2 Process/Workflow

A *process* or a *workflow* describes the order in which a series of steps needs to be executed. Two types of workflows are distinguished: entity-oriented and activity-oriented workflows. The entity-oriented (or content-oriented) workflows focus on the state of an entity. Entities are always in one specific state. If certain conditions are met, transitions from that state to another out of a set of possibilities take place. Such systems place the content object (for example a document) in the center of the workflow process. Each entity has an initial state and a destination state for each transition. Activity-based workflows concentrate on the activities that have to be done. As soon as an activity has

been completed, the process definition specifies what are the next activities that need to be done. The central point is not the document, but a task. In this approach process definitions can be describe without involving any kind of content.

Aiming a solution for the inter-institutional heterogeneous nature of healthcare, both approaches are considered as insufficient. Traditional workflow management systems consider predefined workflow schemata and are mostly instantiated by a central enactment unit. A comprehensive prospective conceptualization of a workflow, which is ad-hoc, decentralized and operates with initially unknown set of actors, states and transitions, is needed. Further, support for content work and support for coordination should also be regarded separately ([LED⁺99], [NL09]). The document-oriented approach proposed in [NL09] addresses these requirements.

3.3 Active Documents

An *active document* is a document that allows a direct interaction with itself ([NL10]). Therefore, documents become active documents if they are assigned with active-properties (see 1.2). Active documents allow infrastructures to adapt to variations in application demands, or provide an infrastructure for interactive document applications. For example, they can be used in a distributed infrastructure, in which activity is directly associated with documents, rather than being locked inside applications, that are invoked to process them ([DEH⁺00], [HM00]). Besides, active documents utilize context information and distributed resources to support users ([WKJ⁺01]).

3.4 ECA-Paradigm vs. Inference Engines

Event-Condition-Action (ECA) rules are a general formalism for modeling the functionality of an active Database Management System (DBMS). An active database is a database with the event monitoring scheme for detecting manipulation activities of data, and automatically executing actions in response when certain events occur and particular conditions are met. Active databases support the creation of triggers ([DKM86], [MD89]).

ECA rules generalize mechanisms such as assertions, triggers, alerters, database procedures, and production rules. Events are typically database operations (such as

delete, update, insert), temporal events, signals from user processes, a combination of these. Conditions are retrieved from defined queries over the persisted data in the database. Actions are normally the invoking of a program or a procedure inside the database ([DBM88]).

ECA Rules can be applied for *distributed* active Database Systems (DBS) as well. Event detection and event processing in the form of ECA rules are distributed across the network and can be configured into event handlers that implement specific policies ([KL98], [HS09], [ACJZ08]).

Inference engines on the contrary follow a different approach. They derive conclusions from an initial set of given facts, stored in a knowledge base. Inference engines define a subset of reasoning engines. They apply heuristics, based on logic and statistics to known facts or presumed hypotheses in order to draw conclusions or prove an assumption. Commonly, in inference engines a *forward-chaining* or a *backward-chaining* algorithm is used (see also section 5.1) to derive implications. By this approach, the separation of knowledge (in the form of predefined rules) from the control unit (the inference engine) is distinguished. An inference engine is an important component of expert systems ([Jac98]), software agents or rule-based systems. However, reasoning over facts and formulating certain conclusions based on the retrieved knowledge does not necessarily leads to the firing of specific actions.

3.5 Conclusion

In this thesis a decentralized document-oriented workflow approach is explained. The thesis aims to design a component, which will contribute to this approach by implementing the active-properties of the documents. Therefore, an overview of some basic topics like distributed systems, processes and active documents was given in this chapter. In addition, a comparison between two paradigms: traditional ECA rules and inference engines, was made in order to cover the possibilities for an approach, that best suits the pursued objectives of the active-properties module to be designed. Inference engines do not regard the invoking of actions as a reaction to the derived conclusions, whereas the traditional ECA paradigm offers actions. The ECA rules on the other hand reach decision breakpoints in a simple way without being able to cover complex correlations

of events and facts. A hybrid combination of both paradigms is needed, in order to be able to take complex input knowledge under consideration, match it with conditions and rules in order to reach a conclusion, and as a result be able to trigger a corresponding reaction.

4 Requirements Analysis

This chapter provides the requirements for the intended component. The main idea of the α -Flow is that active documents should become the primary means of information exchange within a treatment process. Furthermore, they should offer ways to be systematically classified. Main assumption is that the sets of participants and workflow steps cannot be initially known. This feature must be taken under consideration as well as non-functional requirements like the light-weight of the subsystem. A coarse overview of the artifacts in the α -Flow and its components is also expounded below. The role of the adornment models and the structure of the artifacts are described as well.

4.1 The Domain Model

The domain model can be explained through its two-fold meaning - processual and documentary. The abstract term of an α -Flow illustrates a dynamic-shaped workflow, where on the other hand the documentary approach adopts the concept of electronic documents, called α -Docs, as the primary means for information transfer. The α -Flow has no foreseen structure, as it progressively evolves it describes the past and the next steps of a treatment episode by means of coordination artifacts.

4.1.1 The Artifacts: α -Doc and α -Cards

A distributed process characterized by a particular goal and constructed of a bunch of distributed activities is called an α -Episode ([NL09]). There are many *episodes* within the α -Flow depending on the complexity of the medical condition that is treated. Such α -Episode is represented by one α -Doc. In the α -Flow approach, the α -Docs are assigned with active-properties. The term ' α ' itself relates to the active-properties bound to it. An α -Doc is a document and application at the same time.

relationships¹ between the content cards. The TSA card provides hence information about the overall workflow schema. The CRA-Payload on the other hand consists of a list of all participating parties relevant for the α -Episode. A participant can be any actor, who is somehow involved in the treatment episode. Its domain model wraps information about the actor themselves, the role they are taking in the workflow, the institution they are representing and the network characteristics of their workplaces. Both the TSA and CRA α -Cards are used for structuring content documents and consolidating coordination information ([NL09]).

In contrast to the coordination cards, the content α -Cards can be (initially) none, one or many. They have the same structure as the coordination cards, but are of a different fundamental semantic type (\rightarrow *content*). The content artifacts bear in their payloads treatment results in the form of medical papers. A content α -Card is created at first locally. The α -Card is not shown to other participants until it is marked as *public*. A public α -Card implies that each distributed replica of it is made visible to those who are concerned with its payload.

4.1.2 The Adornment Models

An α -Card includes a set of adornments for the payload. The adornment models reveal properties (or attributes) of the α -Cards, such as *validity* and *visibility*, as well as other process-relevant metadata ([NL09]). The adornments are defined in the α -Adornment-Descriptor.

Each α -Card has an identifier, which is unique for the entire α -Flow, and it can be given a fully descriptive name. As already mentioned all α -Cards split into two *fundamental types*: coordination and content. This classification is one of the attributes in the set of adornments for each α -Card. A further attribute the *syntactic payload type* contains the format of the payload. The owner of the card (the *subject*) and the patient (the *object*) are also recorded as adornments in the α -Adornment-Descriptor.

The model distinguishes between the visibility and the validity of an α -Card. In traditional database-centric approaches, these two properties are strictly related to one

¹ A *relationship* exists, if producing one card requires another card to be available. For example, a result report on diagnostic findings has dependency to a referral voucher in healthcare.

another. Therefore, objects made visible are assumed to be valid. In this application it is not implicitly the case. An α -Card can already be *public* (i.e. visible to other process participants), but its content (the payload) can still not be *valid*; and vice versa. An α -Card marked as invalid declares that its payload provides interim information.

Cards have a *semantic type*. An α -Card can be a referral voucher, a result report or the like. The set of possible semantic types is determinist, but extendable. It is constituted according to the paperwork conventionality for a therapy.

Along with visibility and validity of the card, one can mark if a new *version* and/or a new *variant* of the payload of the α -Card has been published. The versions and presumably the variants are managed additionally. A *versioning* attribute signifies if the card is currently under version control. Only if this attribute is set, the version counter is incremented. The version can be incremented only if a payload exists for the α -Card and respectively versioning has been enabled. There isn't any special management of variants for now, but such is intended to be devised in the future.

Further, there are two more tags for the status of an α -Card. They indicate if the card is marked as *deleted* or as *deferred*. If these attributes are set to false, than the α -Card is referred to as "in use" and it can be operated on. If the card is marked "deleted", all attempts for manipulation on it are declined. The card itself is never actually deleted, it is only preliminary disabled. A deferred card is a card whose fulfillment is not possible for indefinite time, yet the underlying impediment is not part of the model.

In addition, there is a label for *priority* which can be assigned to an α -Card. Possible values are high, normal or low. The priority attribute is important for the user at most. Cards with high priority are expected to be handled first. In this context a *dueDate* can also be set for determining a deadline for providing the artifact.

4.2 Overview: α -Flow Components

Although the α -Flow is still under construction, there are some components, which are already advanced enough to be entitled as being in their mature state. In the following a brief overview of the α -Flow modules, as they are currently conceived, will be given, with an accent on the active-properties component and its role in the flow.

A coarse overview of the α -Flow architecture currently includes seven basic components: the α -Editor, the α -Properties, the α -Model, the α -Injector, the α -VerVarStore, α -Institutions and α -Overlay-Network. A module outline is offered in figure 4.2. Most of the α -Flow components reside within the peer and are node-centric: the Editor, the α -Properties, the Injector and the VerVarStore module. The abstract α -Model is significant for all components, as it constitutes the domain presentation. The α -Institutions and α -Overlay-Network modules coordinate and keep track of all members in the network. Their roles in the α -Flow are comprehensive. In the following the basic functions of the α -Flow components are explained.

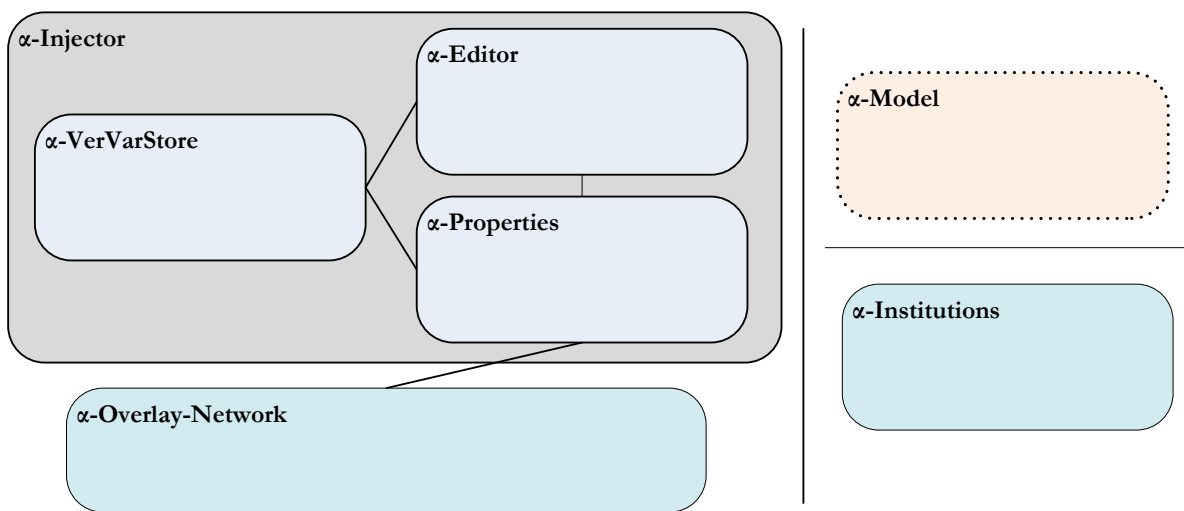


Figure 4.2: The α -Flow Subprojects Overview

In the *Model* subproject the artifacts, as described in chapter 4.1, are designed. Their representation and the prototype design of the adornment models are constituted here. This model of an α -Card is temporary. The coordination and the content card as well as the model of their payload are kept simple. They fulfill the minimum requirements for the domain model.

The α -Injector assigns a passive document (payload) active qualities. It prepares the passive payload with the set of adornments (as process-relevant metadata attributes) and then includes it to an existing α -Doc or starts a brand new α -Doc with the new document file as the first α -Card (i.e. its payload). This component *injects* life to passive documents. It enwraps the passive document and bestows it with its own editor and

logic (active-properties), in this way making it an integrated and active element of a distributed process.

A further component is the α -*VerVarStore*. It supports the payload administration of the α -Cards and the storage of the artifacts. The payload of either a content α -Card or the coordination α -Cards is governed by this module, managing versions and variants separately. It caches the changing payloads during run-time and persists the α -Doc, the α -Cards and their payloads locally on the disk at the end.

The α -*Institutions* component manages all the institutions, the potential actors and their roles in the entire α -Flow network. Its purpose is to support the search for available actors for a certain process role.

The α -*Overlay-Network* deals with the peer management. The participants are organized in the institutions they represent. Each institution has a *super-peer* - that is a server-like computer, which controls the flow between the peers and knows the super-peers of the other institutions. It allocates the recipients that are online, in order to enable a direct multicast to them. For all the peers, that are offline, this super-peer is always available, so they are able to fetch the newest updates on pull-principle basis on demand. The objective is to realize the distinctive offline characteristics of the application. The primary function of this component would be to provide a *Recipient List* to the α -Properties module on demand. In this list the network information of all super-peers of all institutions, that are involved in the specific treatment episode, is presented.

The α -Institutions and the α -Overlay-Network components are only conceptually elaborated so far and they are still the subject of future work. In order to provide the functions they have in the α -Flow, a simple substitute implementation was embedded in the prototype of the α -Properties.

The *Editor* allows the user to view, manipulate and edit the active document. The active-properties module (here called: α -*Properties*) lies underneath the Editor and is responsible for the most of the logic. Each α -Doc has its own Editor and its own α -Properties. Figure 4.3 outlines the correlation between the α -Episodes within an α -Flow per distinct patient. The episodes are mapped to α -Docs, which in turn are each equipped with its own editor (α -Editor), storage manager (α -VerVarStore) and logic (α -Properties). On the desktop of each participant reside a replica of the attendant α -Doc of every episode in which the participant is involved.

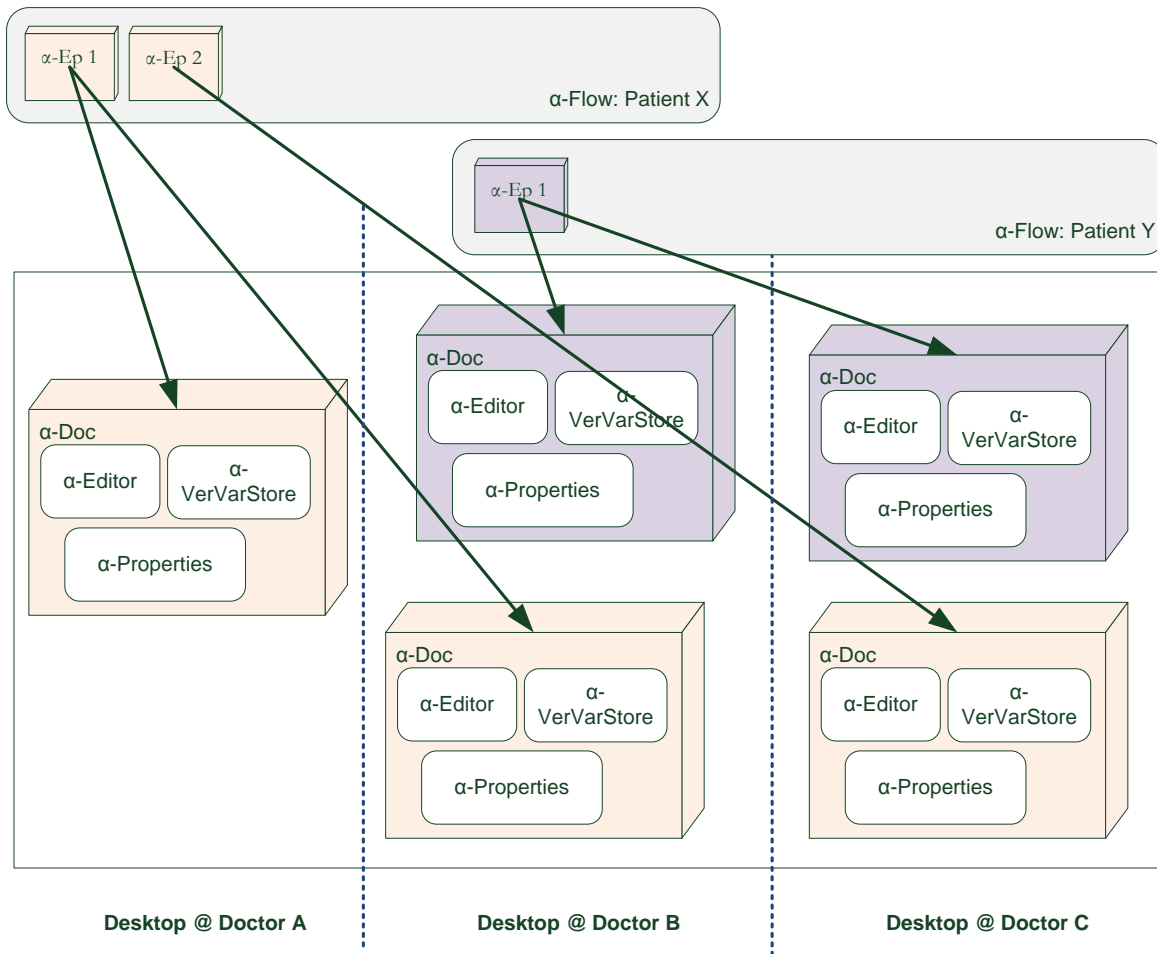


Figure 4.3: The Correlation between the α -Doc, the Editor, the Properties and the VerVarStore within the α -Flow

The α -Properties are the focus of this thesis. They represent this part of the α -Flow, which implements the most of the coordination logic. They are present in every α -Doc, and can only see and work with the artifacts of this container or the distributed replicas of it. The α -Properties are not global for the Flow, but address always only one specific α -Episode. There are three main tasks, this component is responsible for: 1) triggering the α -Card publication based on the visibility adornment; 2) enforcing the versioning of the α -Card depending on visibility and validity; and 3) consequently propagating changes to the other participants when the card is published ([NL09]).

So far it is assumed that every participant, interested or directly involved in an α -Episode already has a copy of the corresponding α -Doc. How these α -Docs are going to be distributed is still an open issue, but it is considered to take place out-of-the-system; that is it will be send as an attachment in an e-mail, or will be physically transported with a USB¹ flash drive, or the like.

This chapter aims to introduce what is relevant for the design of the α -Properties module and what requirements it must meet. The first two sections gave an overview of the domain model and the place of the α -Properties in the context of α -Flow, whereas the next sections outline the concrete functionality this component must implement.

4.3 Functional Requirements Framework

The α -Flow project aims for an IT-supported document-based workflow within a distributed environment of heterogeneous systems. In the following, the functional requirements for the overall approach are outlined, with the special focus on the active-properties component. Many considered options are motivated by the idea of utilizing a rule engine for the execution of its functionality.

Distribution - First of all, the workflow takes place between participants that reside on physically scattered workplaces. Consequently, this should be considered in the design. It is expected that the system is able to distribute locally occurred changes to all concerned participants and accordingly receive such changes. Therefore, there is a need for a distributed rule execution. The propagation of change requests, artifacts and rules should be warranted.

Autonomy - The autonomy of the α -Docs is presumed. Having documents that are able to interact with themselves autonomously is what makes the communication on peer-to-peer level among distributed heterogeneous systems possible. Besides, α -Docs should have the ability to react entirely autonomous to the user's activities and be in control of the process progression.

Monitoring of changes - It is important to provide a monitoring module, which should be in charge of keeping a log of the occurred events. It has to be assured that

¹ Universal Serial Bus (USB)

actions triggered in the α -Properties subsystem reach the user in the form of notifications about relevant changes in the artifacts throughout the entire workflow.

There are some further aspects, that must be ensured as they follow as side effects to the main functional requirements. The first issue regards data **synchronization** among the distributed replicas of the artifacts. The second necessity concerns the management of different **versions** and **variants** of the artifacts.

4.4 Functional Requirements of the API

Beside the global functional requirements, there are some extra requirements for the Application Programming Interface (API) of the α -Properties. They are described in the following by reference to the consideration of utilizing a rule engine for the α -Properties module.

4.4.1 Motivating the Usage of a Rule Engine

The α -Document is postulated as an active document. As such, its autonomy and the ability to decide alone what happens to itself are its basic features. Applying a rule engine in the heart of it, that realizes this resoluteness, is very intuitive. Rule engines are capable of reasoning over facts and picking up responsive actions based on them. They allow for deploying application-specific rules dynamically and have scalable architecture. The latter has the advantage, that they can be implanted in a complex application, become easy a part of it and end up supporting its logic as well. Further, rule engines enable the handling of repetitive tasks. Taking these features into account has resulted in considering a rule engine for the system design of the α -Properties module, as these functionalities are presumed to be implemented in it.

4.4.2 Requirements of the API

Reasoning over facts, if user triggers event or if some other peer sends an event: A classification of events and the predefinition of the possible effects expected upon their occurrence by means of rules should be provided in order to represent the required behavior of the active documents. A package of rules, which meet the current

minimum functional expectations of the application, should be developed. The rule engine uses these rules in order to respond to user-triggered events. These responses should include reasoning over data, changing the data, inserting data in the workflow, propagating actions back to the user interface and propagating events out of the α -Doc instance to all its replicas.

The rules expect certain requests from the user. Upon their invoking, they are forwarded to the rule engine where the actual actions take place. The user is allowed to trigger the following events: changes in the adornments, the addition of a new content card, addition and changing of a participant, addition of a relationship between cards, attaching of payload to content cards, selection of an α -Card or the whole α -Doc for viewing or editing. The proposed rules must be designed with respect to these application-specific events.

Managing versions of the artifacts: In the declaration of the rules, versions and management of versions must be considered. Same intention is deemed for the variants of the artifacts, although the concept of variants is not fully defined yet and therefore cannot be implemented.

Changes get propagated: Another task of the α -Properties component is the propagating of changes through the network to all participants' nodes over to the concerned parties. Preparing the artifacts for the transmission (JAXB-Binding) and the transmitting to peers itself must be also realized, as a preliminary solution while the α -Overlay-Network module is not implemented.

The Editor is notified of changes: Beside the customary functions, the α -Properties must be able to send notifications about actions that take place within it to the Editor. It should be possible to give the Editor feedback, if something in the meanwhile altered the current state of the α -Document. The gathered information helps keeping a log of the occurred events.

Enables Queries over the current state of the artifacts per α -Episode (on fine-grained basis): Another functionality the prototype should provide is the query of artifacts in the pool of current facts. What is the state of the α -Doc, how many content cards it currently consists of and all kind of other information about the facts must be retrievable on demand.

Providing dynamical upload of rules: Using a rule-based system has the advantage that the application-specific rules are totally decoupled from the domain model and can

be replaced, altered or extended without any further ado. In this context providing a *dynamic* update of rule packages should be provided. The system ideally offers ways to load new knowledge dynamically at *run-time*.

4.5 Non-Functional Requirements

Apart from the functional features the system should offer, there are some non-functional requirements about how these features should be implemented as well. Such non-functional requirements are the lightweight nature of the application and the timely responses it should deliver.

Light-weight - In order to classify a system as lightweight some aspects must be present.

A lightweight system can mean different characteristics. In our context lightweight means on the one hand that it does not need any explicit installation, that is it is self-contained. Lightweight applications are characterized with quick start-up time and the usage of few or zero external requisites. On the other hand the application size must be held minor. Furthermore, lightweight systems can also mean to be easy to learn and work with, due to an intentionally limited scope of functions. Technically, light-weight is implied if the application includes few or zero internal dependencies. Lightweight systems are often very flexible, easy to tailor and extensible. Consequently, the fulfillment of these characteristics is presumed.

Instant reactions - By reasoning over facts and triggering of the action part of matched rules reactions are expected to take place instantly. The propagation of the events and artifacts should also follow promptly, even though the propagation is subject to network latency.

4.6 Conclusion

The goal of this thesis is to design and implement a prototype solution for the active-properties component of the α -Doc in the context of the α -Flow. The pursue of this goal implies the definition of the system requirements. This chapter offers a background review of the domain model and an overview of what the system should accomplish as well as its quality requirements. Since the interaction between the user (through an

editor) and the α -Properties is intense and the users actions are of great importance for the building of a knowledge base, some further application-specific requirements are also taken into account.

5 Rule Engines and JBoss Drools

The decision to apply a rule engine in order to meet the requirements was motivated by the advantages they offer in a dynamic environment. A rule engine can provide a flexible way to describe and modify rules over time. Furthermore, a rule engine provides a structure for factoring the logic out of the rest of the system, aiding the effort to separate concerns. Moreover, using a rule engine is preferable, if rules are likely to change over time due to the nature of the application. In this thesis the prototype to be designed fits such requirements. Therefore, a rule engine is to be utilized for it. In the following an overview of what rule engines are and how they are designed, using the example of the JBoss Drools platform, is discussed.

5.1 Rule Engines

Rule engines provide means for declarative programming and offer tools to indite rule sets outside the application code. Their purpose is to react upon input in order to produce some output. They can either execute interpreted rules directly or just delegate the interpretations to another application component responsible for the handling.

A rule engine is an interpreter for if/then statements, called rules. The *if*-part of the rule contains conditions, which must be fulfilled in order to trigger the second (*then*-) part of the rule: the actions. The input to a rule engine are the rule execution set and some data objects (called *facts*). The facts represent the current state of the application domain objects. The output depends on the input and can be one of the following: the same, but slightly modified data objects; some other new constructed objects or the invocation of functions, which appoint certain services, like sending a notification or propagating facts.

A rule engine typically has the following components. There is a *Knowledge Base* (or *Rule Base*), which contains all the predefined rules. Coexisting, there is a *Working*

Memory where all current facts are held. The rule engine operates on these facts when a rule is fired. There is also a *Pattern Matcher*, which decides, based on the facts in the working memory, which rules qualify to be applied. The *inference engine* is used to find out which from the chosen rules should be activated. The activated rules are then queued in the *Agenda*. The process of ordering the Agenda is called *conflict resolution*. Once the order of the Agenda is set, the first rule is fired. After a rule is fired the new state of the facts is again reasoned over, in case vital changes have occurred as a result of the triggered actions. The entire process is constantly repeated until there are no further rules assigned for execution. The main components of a typical rule engine can be seen among others in figure 5.1 below. The figure depicts their connection using the example of the *JBoss Drools* platform.

Rule engines either implement a forward- or a backward-chaining strategy, or sometimes both. The forward-chaining method is called data-driven, whereas the backward-chaining method is labeled to be goal-driven. They differ in the way they reason over facts. In forward-chaining mode all available facts in the Working Memory are taken under consideration for a start. Subsequently all applicable rules are triggered. Their actions are then used as input for the qualification of other rules until a certain goal is accomplished. Forward-chaining rule engines are suitable for deriving high-level conclusions based on simple input facts, thus resulting in the attempt to apply as many rules as possible. Whereas by the backward-chaining approach, the starting point is the goal (or the *hypothesis*). This goal is presumed by the inference engine, which then consequently tries to determine, if there are facts in the Working Memory to prove the truth of the asserted conclusion.

5.2 JSR-94

JSR-94 is the Java Specification Request for a Java Rule Engine API, [JSR02]. It defines a standard programming interface for developing and using a rule engine. It specifies the interfaces through which rule execution sets are invoked. Furthermore, it constitutes how these rule execution sets are loaded from external resources and registered for use. It does not determine the syntax of the rules and the semantics of their interpreting, and nor the mechanism by which rules are transformed for use by the rule engine.

The JSR-94 specification standardizes a set of fundamental operations, such as parsing rule sets, adding objects to the working memory, firing rules and getting objects from the engine as a result. The specification provides two main parts, a rule administration API and a rule runtime API. The first is used for loading and managing rule sets. Whereas the latter is for executing rule sets by using a rule session.

A *rule session* is a runtime connection between a client application and a specific rule engine and is always associated with a single rule execution set. A rule session reserves resources for itself and thus must be explicitly released when not needed. A rule session can be *stateful* or *stateless*. Within stateful sessions a prolonged interaction with the rule execution set is allowed; input objects can be progressively inserted into the session and output objects can be queried repeatedly. Besides, the session is kept alive during the entire interaction until it is deliberately disposed.

A *rule execution set* is a collection of rules for solving a specific task. It has some metadata, such as a unique name and a description. Rules should be written in a standardized way, but up until now there is no such standard. Normally, they are written in descriptive languages.

5.3 JBoss Drools

In this thesis *JBoss Drools* was used to represent the rule engine aspect of the α -Properties. JBoss Drools is JSR-94 compliant and it extends the standard with several additional features. The JBoss Drools is a platform for behavioral modeling. It does not only offer an inference engine (cf. 3.4) but provides a reacting system, which triggers actions, effects changes and persists data, as well. JBoss Drools combines three modeling techniques - Rules Management, Process Management and Complex Event Processing.

5.3.1 Subprojects Overview

Drools consists of four modules aiding to realize these concepts: *Expert*, *Fusion*, *Flow* and *Guvnor*. They are all integrated in the same engine. Consequently, when a new feature is added to one paradigm, it is accessible to the others as well. Depending on what functionalities are needed one may use one, more or all of the paradigms. Their separation is more logical than technical.

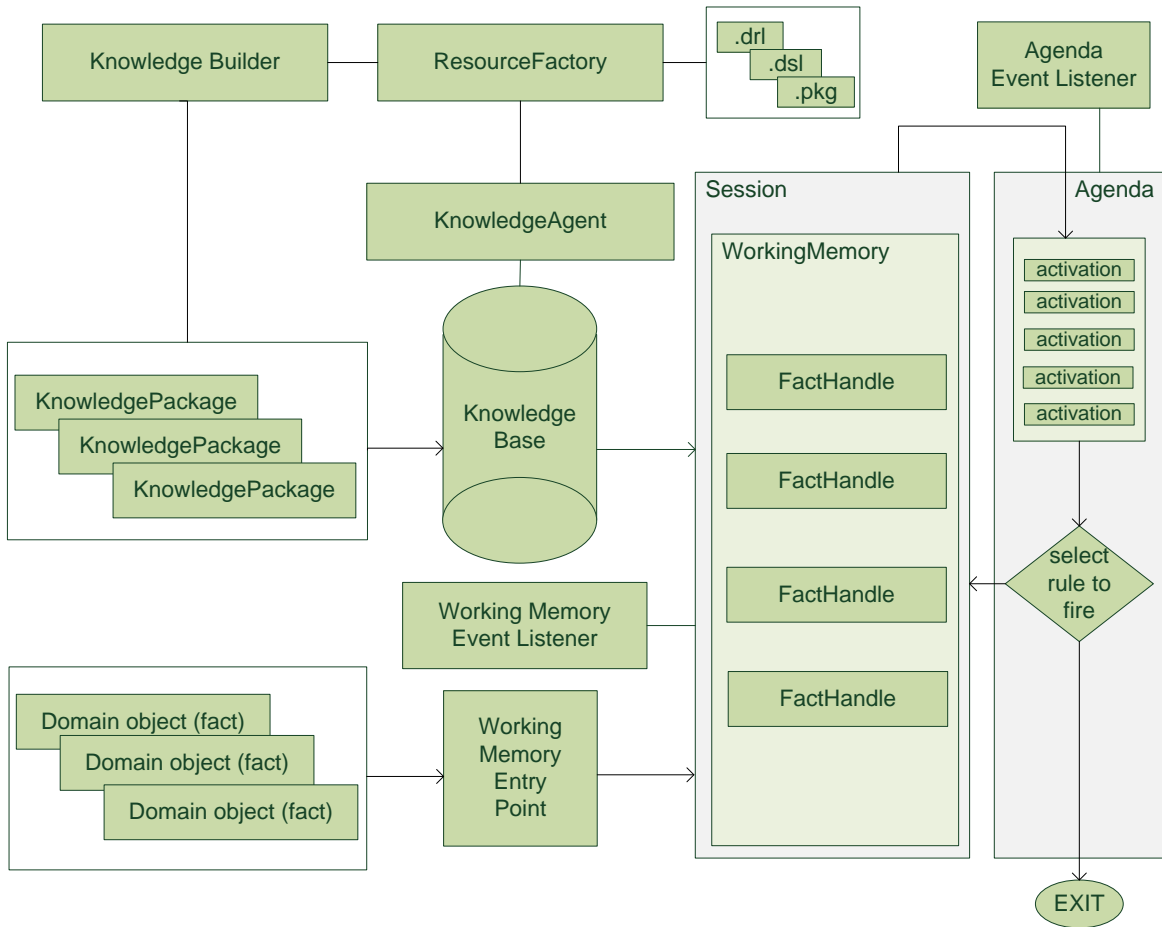


Figure 5.1: JBoss Drools System Components

Drools *Expert* represents the actual rule engine. It defines the basic features such as how to build a Knowledge Base, how to write the rules, how to add Event Listeners on both the Working Memory and the Agenda, how to include a Knowledge Agent, allowing extending the Knowledge Base on demand, and last but not least how to configure all these components. Drools *Expert* serves as a basis for the α -Properties subsystem.

The *Fusion* project concerns temporal reasoning and event processing. Rules are written that monitor events over periods of time, adjust average values, obtained through repeated event occurrences, and react upon excession of predefined thresholds. Based on the gained values certain actions get executed.

The *Flow* project depicts activity-oriented workflows and defines control flows for processes, in which the occurrence of certain states or the accomplishment of a task decides on the further steps in the workflow. This reasoning is achieved through integration of rules within the workflow.

The *Guvnor* represents a centralized Knowledge Base repository, where a great number of rules, models and processes can be managed. It offers a web based GUI and an editor hence enabling non-technical domain experts to operate on the rules intuitively. Moreover, all the artifacts can be versioned and their access can be controlled as well.

A detailed view of the features, which these subprojects implement, is given in the following subsections.

5.3.2 Drools Expert

Drools Expert defines all actual features of the rule engine. An overview of the components of the rule engine is outlined in figure 5.1. It has a Knowledge Base built with the help of the *Knowledge Builder*, which parses resources (knowledge definitions) and builds *Knowledge Packages*. A Knowledge Package is a collection of compiled knowledge definitions, such as rules, processes and models. Knowledge Packages are self-contained and serializable, and they form the basic deployment unit. These packages constitute the Knowledge Base.

The Knowledge Base does not contain any data. From the Knowledge Base many Sessions can be created and they are the units that retain the actual facts. There are two types of sessions: *StatefulKnowledgeSession* and *StatelessKnowledgeSession*. Their meaning is the same as discussed in respect of the standard in section 5.2.

The Working Memory can be partitioned. The *WorkingMemoryEntryPoint* provides the methods for the inserting, updating or retrieving of facts into (a partition of) the Working Memory. There is one default WorkingMemoryEntryPoint, but in case of complex event processing, as it is provided by Fusion, there are respectively as many *WorkingMemoryEntryPoints* as partitions.

Apart from the components, which are common for all JSR-94 compliant rule engines there are some further terms in Drools worth describing. An extract is given in the following. The selection includes concepts that are either specific for Drools or particularly relevant to the designed prototype.

Rule Package - Rule packages are defined by the user. A rule package is a collection of imports, globals, functions, queries and the rules themselves, specified in a file. They get written in one file, because they have semantically something in common, i.e. the solving of a specific task. The package represents a namespace, which must be kept unique. A single Knowledge Base may contain more than one package built on it. Functions in rule packages are used in order to import semantic code into the rule file and in that way separate it from the application logic. Queries can be formulated in order to retrieve desired status information from the Working Memory. The definitions of the rule packages in Drools can be written in either XML, DSL¹ or DRL². The rule packages are actually the rule execution sets known from the JSR-94 specification.

Groups - Rules can be arranged into groups. There can be *Agenda Groups*, *Activation Groups* and *Ruleflow Groups*. Agenda Groups allow to partition the Agenda providing more execution control. Only rules in that Agenda Group, which has acquired the focus, are allowed to fire. The MAIN group has per default the focus of the Agenda. If no other group has the focus set on it, the rules in the MAIN group are matched for hits. In an Activation Group only one rule can fire, and after that rule has fired all the other rules in this group are cancelled from the Agenda. Rules in a Ruleflow Group can only fire when the group is activated. The group itself can only become active when the elaboration of a ruleflow diagram reaches the node representing the group ([Dro10]). Ruleflow Diagrams concern Drools Flow.

Conflict Resolution - Conflict resolution is needed if there are multiple rules activated on the Agenda. As firing a rule may have side effects on the state of objects in the Working Memory, it is essential for the rule engine to know in what order the rules should be fired. The default conflict resolution strategies employed by Drools are: *salience* and LIFO (last in, first out). Setting salience (or *priority*) is manual, in which case the order of the activated rules is in the control of the user. Rules with high priorities are given higher salience numbers than the other rules. Rules with

1 Domain Specific Language (DSL)

2 Drools Rule Language (DRL)

higher salience will be preferred. If no salience is given, the rules are assigned a Working Memory Action counter value. All rules created during the same action get respectively the same value. The execution order of such rules is arbitrary. In general, it is recommended to assign rules with custom salience in order to enforce a particular order, which is thus ensured to be always kept.

Event Listeners - Event Listeners provide a monitoring component for the application. They observe the activities in the Working Memory and the Agenda and can be used for debugging or logging.

The Knowledge Agent - The *Rule Agent* (or *Knowledge Agent*) monitors resources (knowledge definitions). They are defined in a *change-set* file, where their locations are stated. The Knowledge Agent provides automatic caching and loading of these resources. It can update and rebuild the Knowledge Base, in case the monitored resources change. It is performed on pull principle, taking place any given period of time. The pull-period and some other configurations are made through a property-file. Changes of the change-set file itself are however not reflected.

The Pipeline - The concept of pipelines helps the automation of getting objects into and out of the Working Memory. Services like *Java Message Service (JMS)* and transformations of the transported objects for Smooks¹, JAXB², XStream and jXSL³ are provided as well.

5.3.3 Drools Fusion

Fusion extends the Drools Rule Engine with the concept of *events*. Drools Fusion is the module responsible for enabling event processing capabilities. It selects a set of interesting events in a *cloud* or *stream* of events. According to the Drools' definition, an event is "a record of a significant change of state in the application domain" [Dro10]. Events (in Drools) are referred to as a special type of fact, which has a temporal aspect. Events are characterized by temporal constrains and relationships. The detection of their

1 An extensible framework for building applications for processing XML and non XML data (CSV, EDI, Java etc) using Java.

2 Java Architecture for XML Binding (JAXB)

3 A small Java library for writing Excel files using XLS templates and reading data from Excel into Java objects using XML configuration.

occurrence or their absence can be integrated in the condition part of rules, queries and processes. Fusion accomplishes to detect potential relationships (patterns) between the events and takes appropriate actions based on them.

If over a certain period of time events are considered old enough, they are retracted from the session. This way a memory management is granted, and the release of any resources reserved by this event is accomplished. Teaching the rule engine when enough time has passed can be done explicitly or implicitly.

Drools Fusion uses *Sliding Windows* for capturing events within moving windows of interest. Rules can be created in order to aggregate values over a period of time. There are two types of sliding windows: sliding *time* windows and sliding *length* windows. The sliding time windows consider events that happen within a certain slice of time, whereas the length-based windows regard the last x occurrences of events. Due to the timestamps every event can be associated with a sliding window. Use of sliding windows is only possible if the event processing mode is set to *stream*. In *cloud* mode there is no concept of "now" and therefore events can not be compared, although they still have their timestamps. In *stream* mode the events are time-ordered and the streams themselves are synchronized through the use of a session clock. *Cloud* is the default processing mode.

5.3.4 Drools Flow

The Flow module realizes a seamless combination of process control flows and rules. Rules decide which steps in a process should be invoked and hence make it possible to leave space for decision support within processes, particularly if exceptional cases occur. In general, the intention of Drools Flow is very similar to the intention of the α -Flow concept: rules are used to dynamically shape a workflow. The difference between Drools Flow and the α -Flow in regard to activity-oriented workflows is amplified in [NL09].

As rules are used to handle specific exceptional steps in a process, changes in behavior on such crucial parts are made without having to change the process itself. If required adaptations or altered application requirements arise, modifying the rule packages will accomplish the necessary agility. Most of all, definitions of rules and processes in Flow are done separately, thus, evolving steps in the life cycle of rules do not affect the life cycle of the processes and vice versa. Further, Flow makes use of the concept of *human*

tasks. It is possible to define rules, which assign actors to concrete human task(s) in a process. Modelled processes can be added to a knowledge package (just as rules).

Another feature of Flow is the possibility to monitor and log processes and use that information to detect anomalies or create reports about users' activities. The reporting implementation is based on Eclipse BIRT¹ plugin. It allows the creation of reports, the insertion of charts, report exports on web pages and defining specific data sets of interest. Last but not least, Drools Flow can execute processes defined using the BPMN² 2.0 XML format the same way as it executes processes using the custom RuleFlow format.

5.3.5 Drools Guvnor

Guvnor is a stand-alone web-based GUI, with editors and tools for rule management. It has its own centralized repository, where rules, models, processes and other assets are stored. Furthermore, Guvnor offers versioning of all assets in the repository and therefore the conduction of a complete history on them. The Drools Guvnor subproject provides a guided editor for domain experts with no technical background. They are offered an understandable centralized application for elaborating and modifying logic in the form of rule and process definitions (in the context of the domain they are familiar with) through a graphical interface. Additionally, a proper access control is granted for the users who participate in the logic modeling, with respect to the scope they are allowed to interfere with.

In Guvnor *categories* can be defined for grouping artifacts. Categorizing assets contributes to a better navigation among them and supports the search. Furthermore, categories can be used to control visibility in order to restrict access or just hide features for certain users. Snapshots of knowledge packages can also be created. They are copies of the entire package.

Guvnor provides building and deployment of rule packages from multiple assets. The binary packages (*.pkg) are available via URLs or files and hence are made accessible for other applications, which need them for their Knowledge Bases.

1 Business Intelligence Reporting Tool (BIRT)

2 Business Process Modeling Notation (BPMN)

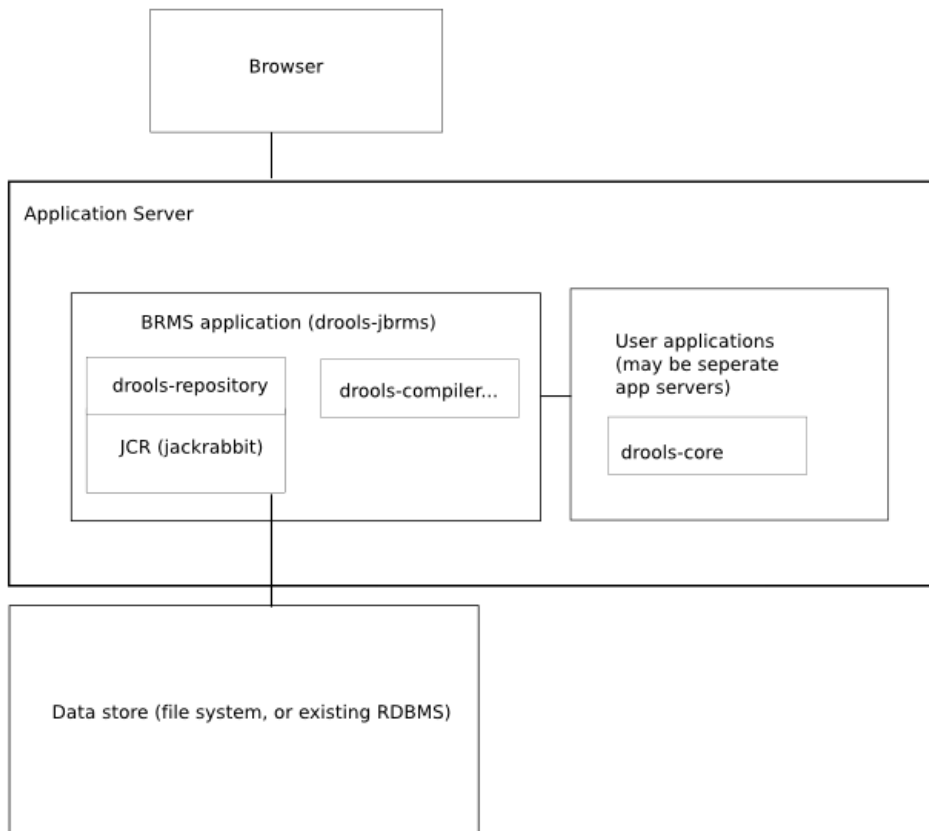


Figure 5.2: Drools Guvnor: Architecture (*Source:* [Dro10])

Figure 5.2 outlines the main components of Guvnor. It is deployed as a war-file, which provides the user interface over the web. Drools is Ajax¹-based and constructed with the GWT² widget toolkit. It uses Apache *Jackrabbit*, an implementation of the Java Content Repository (JCR)³, for the data storage management. This part is highly configurable, as one can use any persistence solution under the content repository implementation - a file system, a database, a WebDAV repository, etc. The versions of assets are stored in the content repository as well, along with the rules, models and processes definitions; so are the snapshots. JBoss Seam is used as the component framework.

1 Asynchronous JavaScript and XML (Ajax)

2 Google Web Toolkit (GWT)

3 The standard of Java Content Repository is defined in the JSR-170 and JSR-283

5.4 Conclusion

In this chapter the background and the mechanisms (functional principles) of rule engines were discussed. Rule engines offer a way to externalize application logic and hence help a system scale better in regard to often changing requirements. The principle of inserting knowledge to systems and letting them make decisions and conduct (re)actions with the help of predefined patterns has the advantage of merging many software solutions for similar behavior into one system.

In addition, the JBoss product *Drools* was examined and the main features of its modules were outlined. From many other existing implementations of a rule engine, the Drools platform was chosen for the prototype of this thesis. The choice was initially motivated by the presence of a Knowledge Agent in Drools, whose concept allows the dynamical load of new rules or the extension of rules. The availability of a monitoring module as the Event Listeners was also regarded. Due to the distributed nature of the α -Flow, concepts like Drools Pipeline were deliberately sought for the propagation of artifacts and rules. Moreover, the Drools platform is an open-source product, which has reached a level of sustainability and maturity, and has conceived a wide palette of complex features.

6 Proposed Solution: α -Properties

This chapter focuses on the α -Properties architecture. More specifically the interaction of the α -Properties with the α -Editor and the α -VerVarStore is regarded. The momentary realization of the communication, as a simplified solution for some of the α -Overlay-Network tasks temporarily assigned to the α -Properties, is also exposed.

6.1 Architecture Overview

The α -Properties, the α -Editor and the α -VerVarStore are three modules of the α -Flow that always reside within the peer. As already explained each α -Doc has its own Editor and Properties module, staying in close relationship to each other and to the VerVarStore respectively. The system design for their interaction is outlined in figure 6.1. Here it is graphically described how these modules are connected as well as what are the concrete components of the α -Properties module in particular. As concluded in previous chapters the JBoss Drools platform is used to realize the rule-based part of the α -Properties. As a result many Drools components are integrated in the system design.

Two interfaces delegate the data flow between the α -Properties and the Editor and between the α -Properties and the VerVarStore. The Editor is responsible for the initialization of the active-properties module. Upon opening of an α -Doc via the Editor, the active-properties component is started. The Editor provides the α -Properties with the current artifacts of the α -Doc before beginning to work with them. The state of the Working Memory, where these artifacts are imported into, is set up every time anew and is kept alive only for the time the Editor is opened (using a stateful Session). When the Editor is closed, the session is disposed and the Working Memory deallocated.

in the Working Memory. Through this interface the Drools instance is started and configured and eventually gets shut down. Additionally, the current model is loaded, inserted in the Working Memory and ready to work with. Here the Editor's Observers are added and the same instance of the α -VerVarStore component used from the Editor to persist the data is passed to the α -Properties.

Users open the α -Document and edit its content α -Cards, sending for every changed adornment or creation of a new content α -Card a request event to this interface. As a result, the rule engine within the active-properties reasons over the incoming event, triggers some rules and effects changes. Eventually the Editor gets notified about the occurred changes and it refreshes its view to the current state of the α -Document. Furthermore, changes of the payload of the coordination cards can be invoked through it. For example, a relationship between two α -Cards can be defined or a new participant can be dynamically added to the set of actors. The new participant gets hence immediately involved in the workflow. The configuration of the listening port for the application is conceived to ensue also dynamically. An initial port is set upon starting the α -Properties, but it is possible to change it at run-time by updating the participant with a new port. This standardized interface represents a facade for the α -Properties component upon the Editor module.

6.3 The α -VerVarStore Interface

The *VerVarStore* regulates the storage and loading of the payloads of α -Cards. It can store payload to a specific version or variant of the α -Card or load the payload to a specific version or variant. It thus realizes versioning of the payloads. The VerVarStore implements versioning (and ideally variants) management in a rudimentary way for now. On hard-coded directory basis all versions build up a hierarchy within the home directory of the application; they are saved under a new folder, named after the number of the new version. The same applies for the payloads of both the coordination and the content cards. The structure of the α -Doc is also persisted there, as a XML-file.

The *TSA-Payload* and *CRA-Payload* are treated in a special way, because they are shared between all participants involved in the α -Episode. In general each participant works with a replica of these payloads, which perpetually requires their synchronization.

Each time a participant, a relationship or a content card are added, the payload of the respective coordination card changes and its version is incremented. As long as the content card is not under version control, that is is the card is not visible and invalid, the payload is saved under one and the same initial folder and it gets overwritten every time the payload changes.

Both the Editor and the α -Properties operate on the same instance of the VerVarStore. Permanent writing is triggered through the Editor, which flushes the artifacts and their payloads at run-time. At the same time it caches the artifacts in order to grant faster access during uptime. The serialization of the α -Doc in a XML-file is not done until application termination. The α -Properties module on the other hand addresses the VerVarStore, if triggering a rule affects the payload of an α -Card: then these changes must be stored.

6.4 The Update Service Interfaces

In order to propagate changes from one Drools instance to the α -Properties component of all the concerned parties, the changes should be transferred through the network. There are two interfaces that realize the distribution: *UpdateServiceSender* and *UpdateServiceReceiver*. Each application has one fixed listening port for incoming event streams. This port is used to broadcast the events and via this port it is amenable in case another application wants to send events to it. The current prototype implements it the following way: the application fetches the ports and hostnames of all participants from the CRA-Payload on demand. In the future work a special module, the α -Overlay-Network, would arrange for the CRA-Payload of every α -Document the needed network information and support the data transfer.

6.5 Conclusion

In quest of a solution for an autonomous rule-based subsystem in the context of active documents, the α -Properties module was developed. It is a part of the α -Flow concept and is responsible for the manipulation of the active documents and the coordination of their distributed replicas in a decentralized environment. The α -Properties module

cooperates directly with the α -Editor and the α -VerVarStore. Their interaction was expounded in this chapter. The self-reliance of the α -Properties is ensured through a rule engine, deployed to handle events and actions on its own. In the implemented prototype some functions of the α -Overlay-Network were shifted to the α -Properties. Nevertheless the vision is to factor them out in the α -Overlay-Network component, once its boundaries and its concrete role in the flow are more specifically determined.

7 System Design: α -Properties

In this chapter the system design of the proposed solution is amplified. The prototype of the proposed solution was implemented in a way to fulfill the functional requirements from section 4.3, 4.4 and 4.5.

An *Event Model* was considered for the events. They were categorized and grouped according to the functional application-specific requirements. Additionally, the corresponding actions to these events were defined in the form of rules, which were also classified semantically. Models for the α -Adornment-Descriptor, the α -Doc, the α -Cards and the structure of their payloads were designed as well.

For the realization of *monitors* for the changes, the Drools Working Memory Listeners are applied. Additionally, a Drools incoming pipeline is used for the implementation of the distribution of updates.

Few initial assumptions and limitations were made for the system design in order to grant prototype simplicity and prove a point. These assumptions include an "always-on" semantics for the connectivity of the peers in the network and the proposal of an initial rule package, which reflects simple application scenario measures. Besides, security and synchronization issues are momentarily left out. Nevertheless, the basic functionalities of an active-properties component are provided.

7.1 The α -Model

The designed domain models for the α -Adornment-Descriptor, the Payload of the α -Cards and the α -Doc are displayed in figures 7.1 and 7.2. The member fields of the `AlphaCard` class represent the adornments, defined for the α -Adornment-Descriptor. The `AlphaDoc` class declares the structure of the α -Doc. It consists of a unique `episodeID`, has a descriptive `title` and a list of all α -Cards.

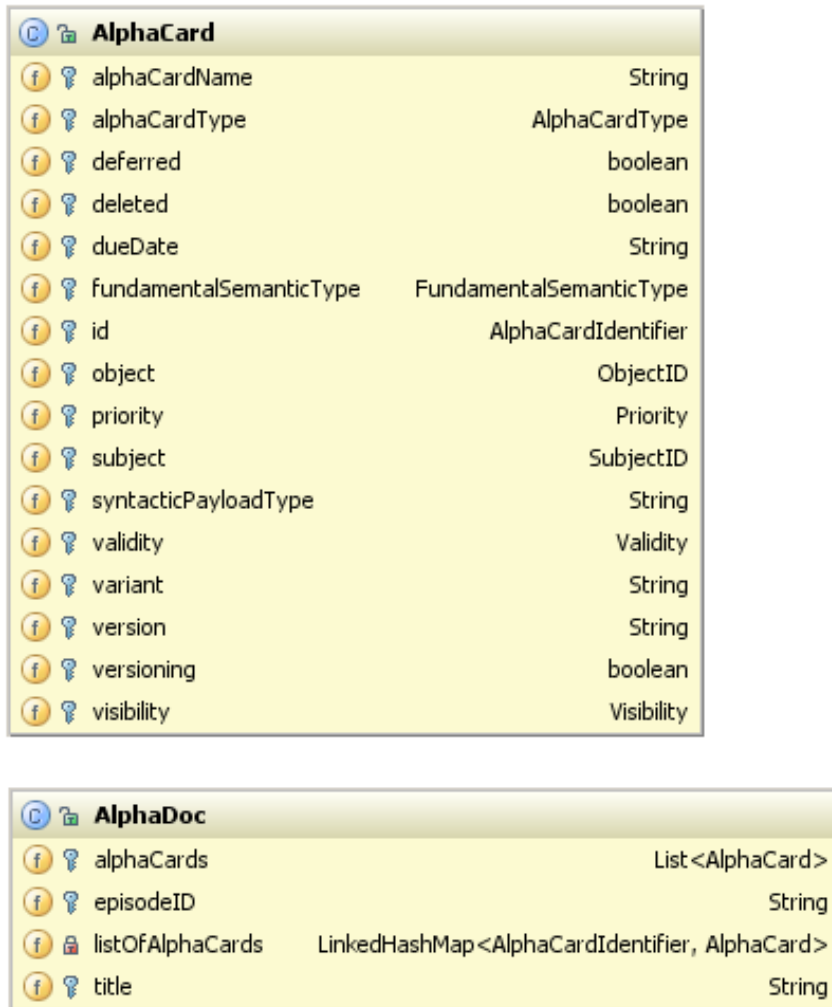


Figure 7.1: The α -Doc and α -Card Models

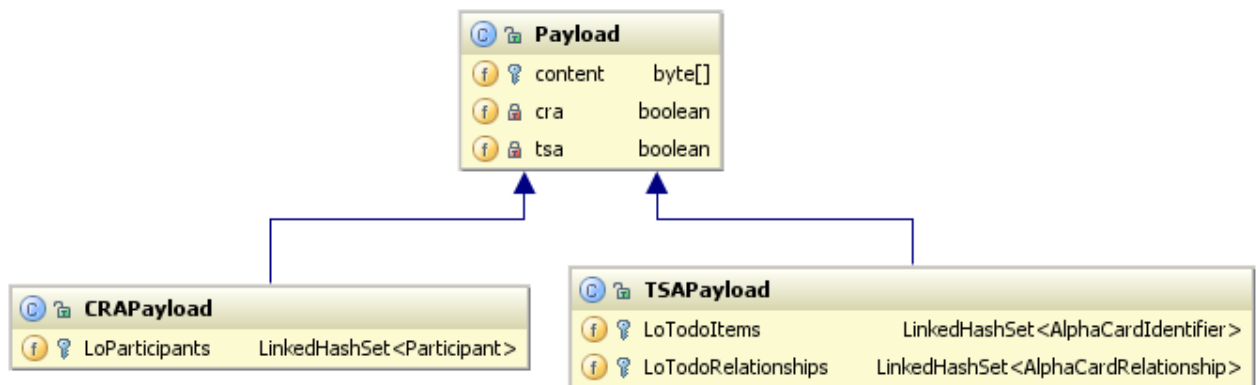
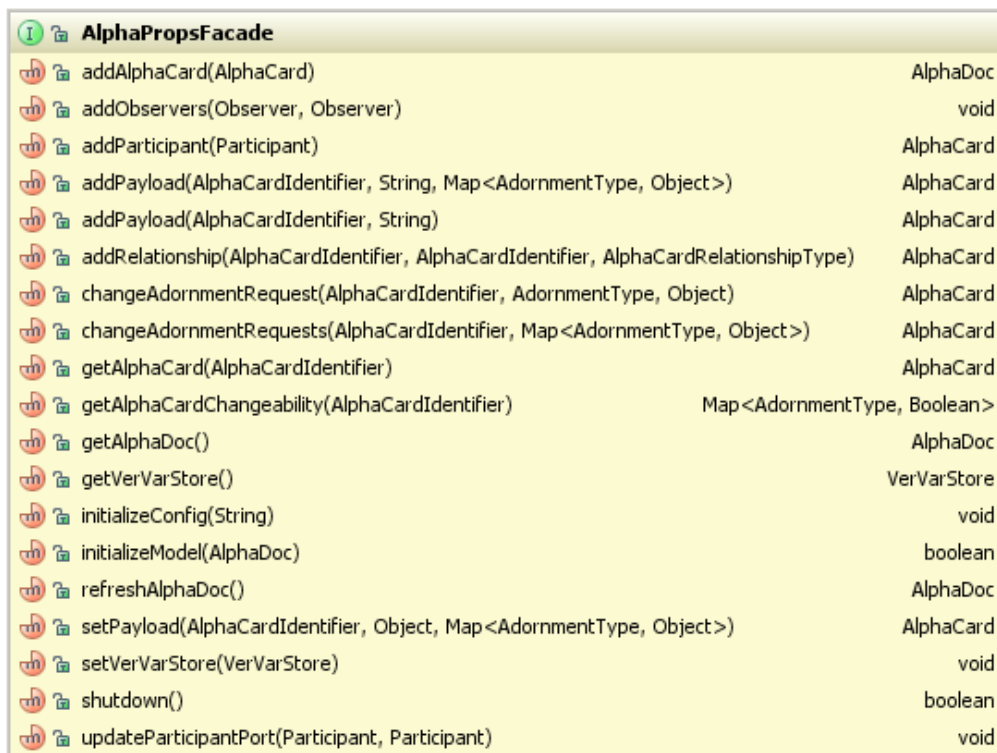


Figure 7.2: The Payload Model

As the payloads of the coordination α -Cards have a special structure, their models are declared separately, derived from the super class `Payload`. The payload of the content cards is so far stored in a binary form (as its format might be Microsoft Word, PDF¹, HL7², CDA³ or the like) and it does not have any special features.

7.2 The α -PropsFacade Interface

Figure 7.3 displays the methods the *AlphaPropsFacade* interface offers to the α -Editor module. The methods reflect the functional requirements for this API, which were elaborated in detail in section 4.4, and the functionality proposed in section 6.2. The



Method Signature	Return Type
<code>addAlphaCard(AlphaCard)</code>	<code>AlphaDoc</code>
<code>addObservers(Observer, Observer)</code>	<code>void</code>
<code>addParticipant(Participant)</code>	<code>AlphaCard</code>
<code>addPayload(AlphaCardIdentifier, String, Map<AdornmentType, Object>)</code>	<code>AlphaCard</code>
<code>addPayload(AlphaCardIdentifier, String)</code>	<code>AlphaCard</code>
<code>addRelationship(AlphaCardIdentifier, AlphaCardIdentifier, AlphaCardRelationshipType)</code>	<code>AlphaCard</code>
<code>changeAdornmentRequest(AlphaCardIdentifier, AdornmentType, Object)</code>	<code>AlphaCard</code>
<code>changeAdornmentRequests(AlphaCardIdentifier, Map<AdornmentType, Object>)</code>	<code>AlphaCard</code>
<code>getAlphaCard(AlphaCardIdentifier)</code>	<code>AlphaCard</code>
<code>getAlphaCardChangeability(AlphaCardIdentifier)</code>	<code>Map<AdornmentType, Boolean></code>
<code>getAlphaDoc()</code>	<code>AlphaDoc</code>
<code>getVerVarStore()</code>	<code>VerVarStore</code>
<code>initializeConfig(String)</code>	<code>void</code>
<code>initializeModel(AlphaDoc)</code>	<code>boolean</code>
<code>refreshAlphaDoc()</code>	<code>AlphaDoc</code>
<code>setPayload(AlphaCardIdentifier, Object, Map<AdornmentType, Object>)</code>	<code>AlphaCard</code>
<code>setVerVarStore(VerVarStore)</code>	<code>void</code>
<code>shutdown()</code>	<code>boolean</code>
<code>updateParticipantPort(Participant, Participant)</code>	<code>void</code>

Figure 7.3: The *AlphaPropsFacade* Interface

interface provides the facility to initialize the α -Properties module, as it is only started when the Editor is opened. Further, it provides methods for invoking data manipulation

1 Portable Document Format (PDF)

2 Health Level 7, <http://www.hl7.org>

3 Clinical Document Architecture

requests like changing an adornment value of an α -Card or adding of a new participant to the workflow and the like. The names of the methods are partly self-explanatory.

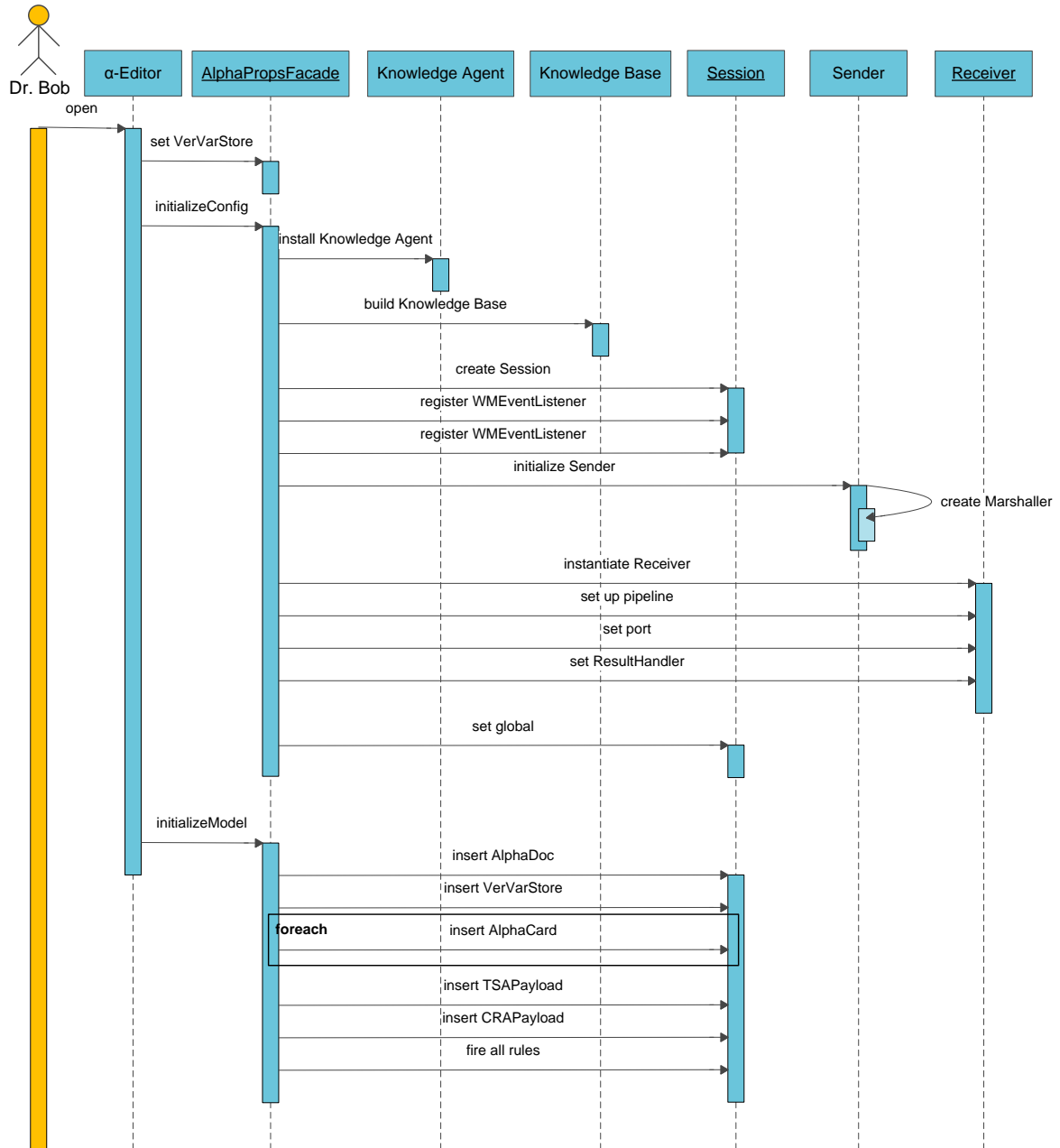


Figure 7.4: The Initialization Methods

Figure 7.4 shows a detailed view of what steps are taken when the Editor is opened, in regard to the α -Properties module. Invoking the `initializeConfig()` method means

the following. Within its call a Knowledge Agent is installed. A Knowledge Agent is considered, because it observes resources consistently and pulls changed resources on interval basis, thus enabling a dynamic rebuild of the Knowledge Base at run-time. Next step in the α -Properties initialization is the initial building of the Knowledge Base, whereupon a Stateful Session is created. Consequently, two `WorkingMemoryEventListeners` are registered for the session (more in section 7.6 below). The next two steps concern the initialization of the `UpdateServiceSender` and `UpdateServiceReceiver`. If globals (Drools) are used in the rule package, they should be set also now. As the `UpdateServiceSender` is considered to be imported as a global for the implementation of the α -Properties, this global is set next (see 8.1.2).

The second method, which the Editor invokes, is the initialization of the model. This method supplies the Session with the working artifacts (*FactHandles*). The α -Doc, the payloads of the coordination cards, all existing α -Cards so far and an instance of the `VerVarStore` are therefore inserted into the Working Memory (for the Session). When the user closes the Editor, it calls the `shutdown()` method of the *AlphaPropsFacade* where the Session is disposed, which eventually terminates the application.

The methods `getAlphaDoc()` and `getAlphaCard()` offer the facility to gain information about the current state of the artifacts in the Working Memory to the user. In fact, some *queries* are pre-defined for the main artifacts and they are invoked when the user makes such inquiries. The queries are amplified in the next chapter (see 8.1.1).

The Observers of the α -Editor register themselves through the interface by the Stateful Session in the α -Properties. There are so far two Observers: one regards any kind of events that happen to or about α -Cards and the other one is interested in changes or events that concern the adornments or the payload of the cards. If these Observers get notified of an event, the Editor responds by updating its insight of the artifacts through the `refreshAlphaDoc()` method.

A figurative example of the collaboration between the α -Properties, α -Editor and the α -VerVarStore in the process is given in figure 7.5. It describes the sequential method calls for two scenarios: changing a payload of an α -Card (above) and changing an adornment of an α -Card (below). As already explained, the user requests are propagated to the α -Properties, where rules are matched and actions like changing artifacts, storing a payload or forwarding of the change requests to other participants take place. Final persistence of the artifacts is realized at the end.

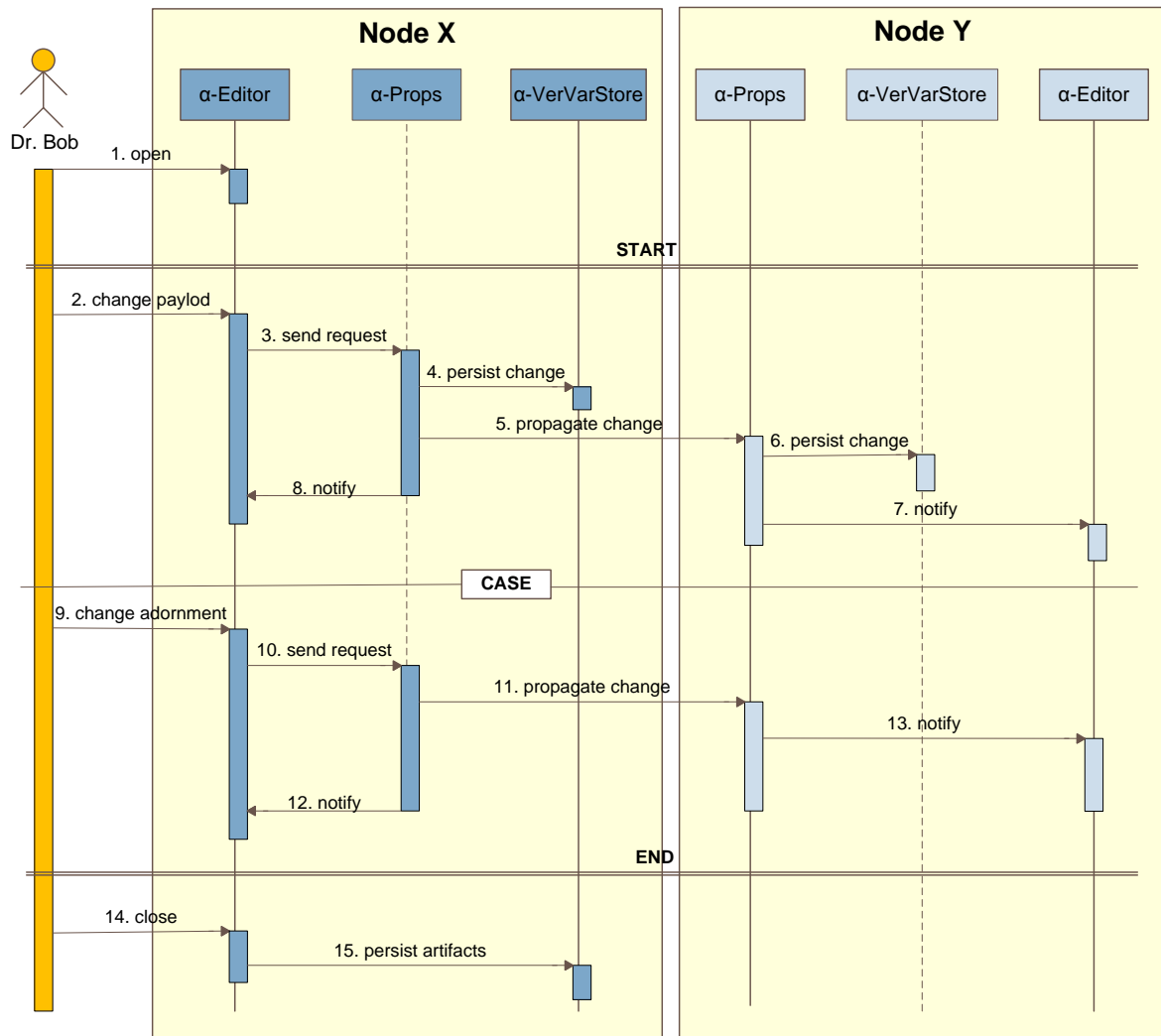


Figure 7.5: The Collaboration between the α -Properties, the α -Editor and the α -VerVarStore

7.3 The Events Classification

The term *event* is beheld two-fold here. There are the *application-specific* events, initialized by the user, and the *technical* events, which are created internally. In fact, the user events are transformed into technical events. The rule-based system in the core of the α -Properties expects these technical events, in the form of *facts*. Their insertion in the Working Memory activates rules from the Knowledge Base. The activated rules are then scheduled on the Agenda to fire and get eventually executed.

The events can be also classified according to how they are inserted into the Working Memory. The most of the events come from the outside, that is either from the user or through the pipeline; but there are some of them which are inserted internally into the Working Memory as a part of the action of a triggered rule.

7.3.1 Application-specific Events

The Editor sends *requests* to the α -Properties through the α -Properties interface. These requests represent events. The events define a finite group of requests, such as adding a content card, requesting an adornment change, setting a payload, appointing a relationship between two cards, adding or altering a participant or getting the current state of the α -Doc. The application-specific events reflect the functional requirements appointed for the user interface (see section 4.4).

7.3.2 Technical Events

The technical events are mapped according to the application-specific events and are created internally. They represent facts, that can be inserted into the Working Memory. They are simple POJOs, that are created from the application-specific events. These new objects are inserted into Working Memory. Their goal is to address exactly the rules that should fire. These POJOs have as member variables the information needed in order to trigger a certain rule. So upon insertion, this information addresses a rule, the rule is fired and, as the artificial fact is not needed any more after that, it is eventually retracted from the Working Memory. This last part happens as a step of the action part of the rule. It is important that no technical facts remain in the Working Memory without serving their purpose of triggering rules, and without eventually being deleted within them. These facts should not stay in the Working Memory, because they are not actually part of the application artifacts set. The reason the concept of technical events is applied is to make manipulations on the artifacts set possible in the first place.

It was considered that the user requests can be divided in exactly four types: 1) add a new α -Card; 2) change an adornment of an α -Card; 3) change payload in general (this involves adding or updating a participant (CRA-Payload), adding a relationship (TSA-Payload) or setting payload to an existing content card); and 4) check changeability of the adornments of an α -Card. This last request is sent before an actual adornment

change request is initiated, in order to check if this change is allowed. Figure 7.6 shows the class diagrams of the four technical events designed for the system: `AddAlphaCardEvent`, `ChangeAdornmentEvent`, `ChangePayloadEvent` and `CheckChangeabilityEvent`.

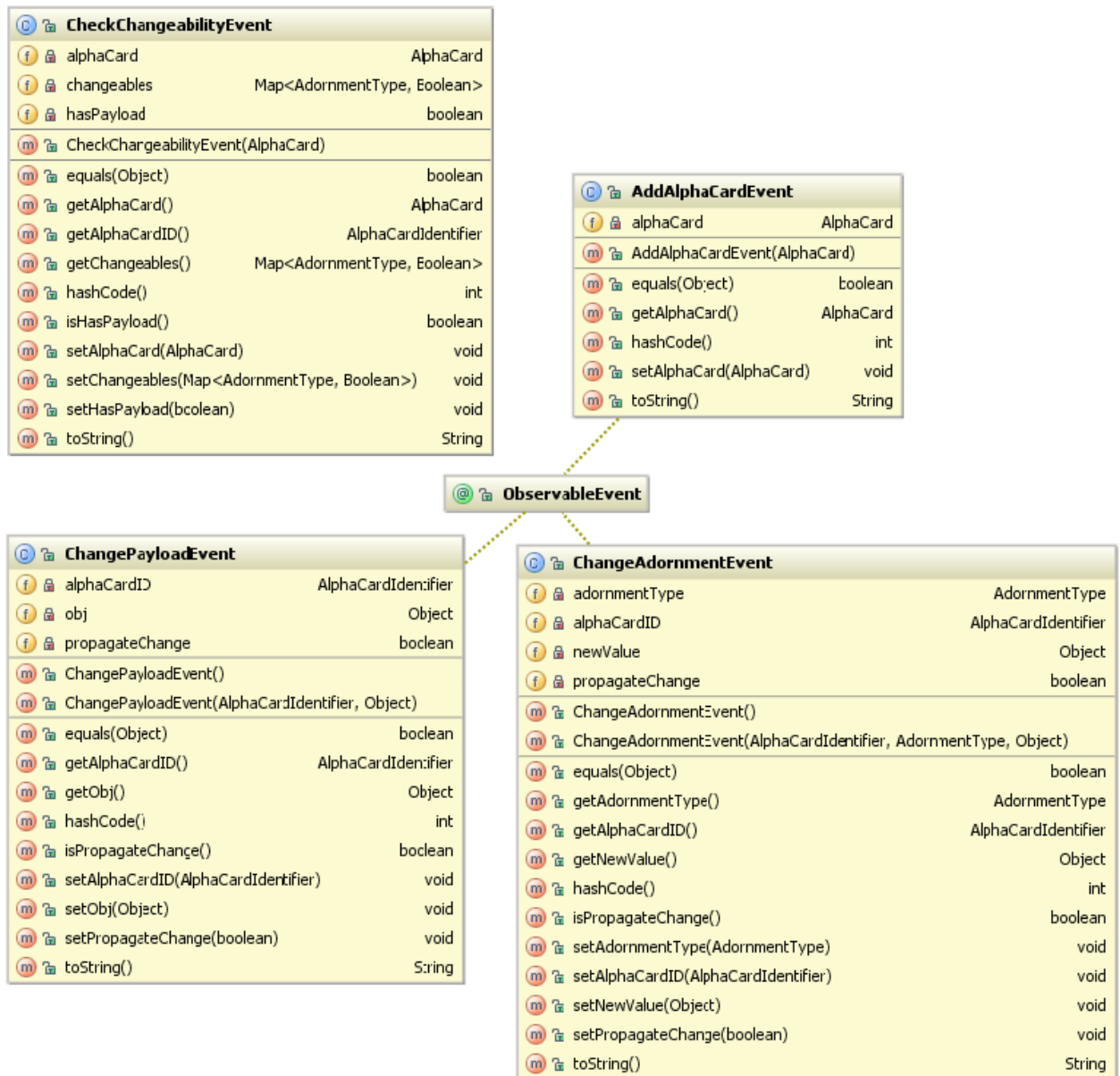


Figure 7.6: The Event Model

Table 7.1: The Technical Events

EVENT NAME	DESCRIPTION
<code>AddAlphaCardEvent</code>	add a new content α -Card
<code>CheckChangeabilityEvent</code>	check adornments changeability, depending if \exists payload
<code>ChangeAdornmentEvent</code>	change an adornment request
<code>ChangePayloadEvent</code>	chnage a payload request

The technical events generated internally correspond to the user triggered events. There are four types of events that can occur and which the subsystem expects. There is a special event, when the α -Doc is extended with a content card - the `AddAlphaCardEvent`. Before an adornment change takes place, a `CheckChangeabilityEvent` is triggered, in order to check which adornment can be changed at all. Subsequently a `ChangeAdornmentEvent` follows, regarding a certain adornment. If the payload of a coordination card should be altered or if a content card gets new payload, the `ChangePayloadEvent` handles these requests. In table 7.1 the four types of events are listed. Beyond these events only the initial insertion of artifacts conduce to adding more objects to the set of work items, involved in the active interaction.

7.4 The Groups of Rules

Twenty-four different rules were designed in order to fulfill the application requirements for the α -Properties behavior. They are distinguished semantically in four groups (see table 7.2): adding a content card, checking changeability of adornments for an α -Card, change an adornment and change a payload. The groups reflect the use-case paradigm of the application: peers should be able to create new artifacts and change their adornments and payload. The rules are grouped the same way the user requests are (cf. the technical events classification).

Table 7.2: The α -Props Groups of Rules

RULE GROUP NAME
Add an α -Card Rules
Check Changeability Rules
Change Adornment Rules
Change Payload Rules

If within the RHS¹ of a rule another rule should be triggered, then a technical event is created inside it. This technical event is conceived to cause the effect of triggering the other rule. This workaround is considered because up to the Drools version 5.0.1, nested rules are not allowed. This solution was used twice: to trigger the update of the TSA-Payload upon inserting of a new content card within the corresponding rule and to launch the propagation of the payload of a content card (if it has any) upon making it visible (*public*).

7.4.1 Adding a New Content α -Card

The initial α -Doc has implicitly only the two coordination cards. Throughout the evolve of the α -Episode many content cards are likely to be added. A new content card can be explicitly generated or be proposed for creation on dragging and dropping of a passive document on an already existing α -Doc. Without a payload, a content card exists but it is appointed as a "place-holder". Actually, the *place-holder* has just an α -Adornment-Descriptor, without payload. The purpose and meaning of a content card is to bear useful payload in the form of medical paperwork. Hence the place-holders exist pro forma, but are actually of no interest to the concerned parties until they get a payload. Table 7.3 shows what are the default values of a place-holder. If not otherwise set by the user, the place-holder is created with the default values. However, the specification of initial values is mandatory.

Figure 7.7 describes what happens if a new content α -Card is added to the α -Doc. As the graphics shows the user request to add a new content card is mapped to the technical event `AddAlphaCardEvent`. This event is inserted into the Stateful Session and it triggers the rule, which handles the actual addition of the card. This rule does the following: it inserts the new content α -Card in the Working Memory, inserts a change payload event, that has the assignment to add the new content card to the TSA-Payload and at last invokes the `sendAlphaCard()` function, which triggers the sending of the new place-holder to the other peers. After the RHS of the rule is executed, the Agenda fires a collateral rule activated by the change payload event created for the TSA-Payload. Within this rule, the version of the TSA coordination card is incremented.

¹ Right Hand Side; the action (*then-*)part of a rule

However, upon the creating of a new α -Card (regardless whether it is a place-holder or not), a copy of it is directly forwarded to all other participants. As long as it stays *private*, only its owner *or* the subject it was assigned to can change it or attach payload

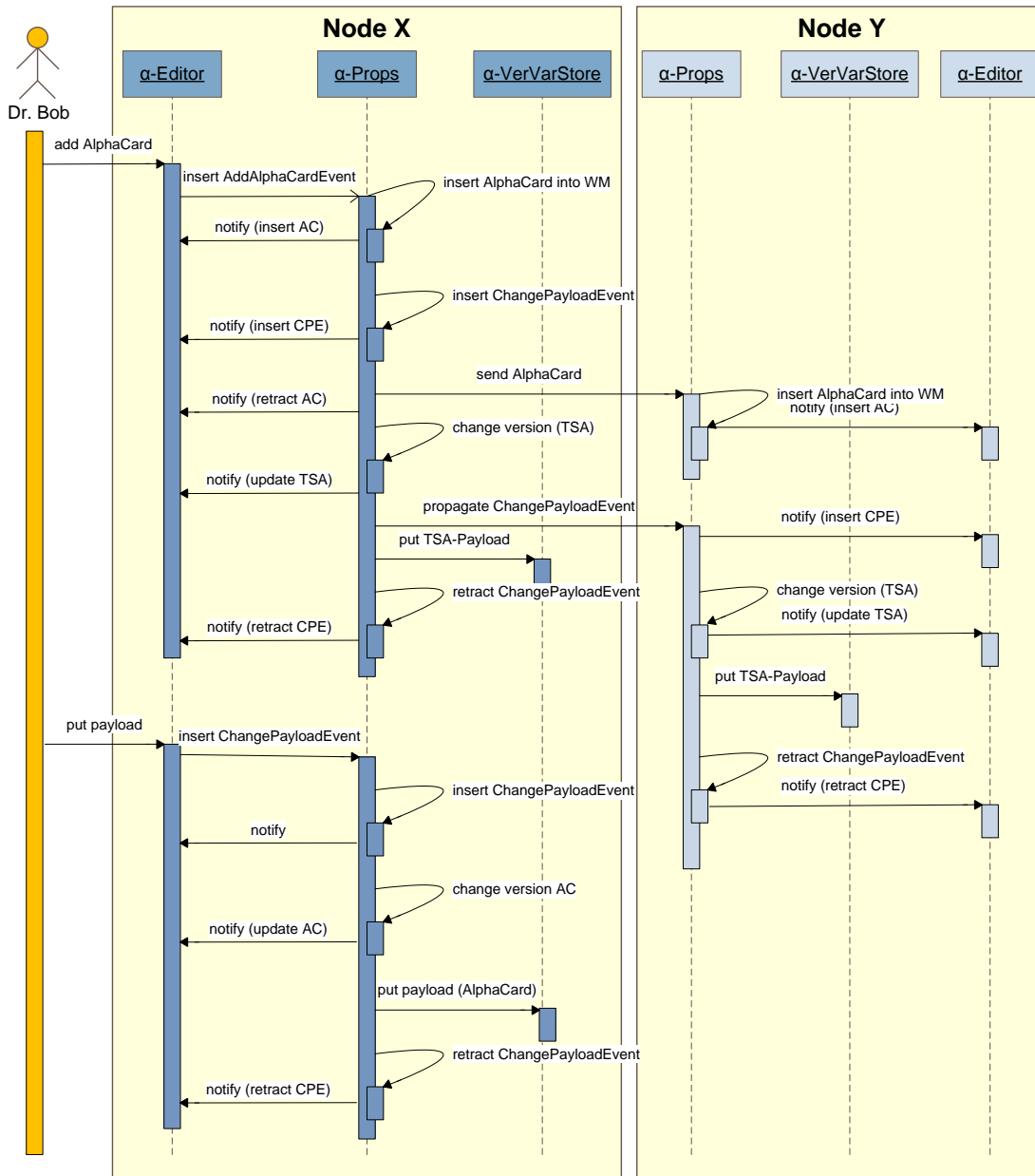


Figure 7.7: The Addition of an α -Card

Table 7.3: The Adornments: Default Values

ADORNMENT	DEFAULT VALUE
id	[<i>automatically set</i>]
alphaCardName	" "
object	[<i>the patient</i>]
subject	[<i>the owner</i>]
visibility	<i>private</i>
validity	<i>invalid</i>
version	<i>0</i>
variant	" "
fundamentalSemanticType	<i>content card</i>
semanticType(\equiv AlphaCardType)	" "
syntacticPayloadType	" "
versioning	<i>false</i>
dueDate	" "
deferred	<i>false</i>
deleted	<i>false</i>
priority	<i>normal</i>

to it. All these changes would stay local and they will not be sent to the other peers, as long as the visibility of the card is *private*. So the other participants can see up to that point only the place-holder. As soon as a content card goes *public*, a copy of the new version is broadcasted to all and it overwrites the initial place-holder. Actually, the old α -Card gets retracted from the Working Memory, as soon as its new version gets inserted. There is one extra rule for the case when the *public* content card arrives. Its action is the retraction of the place-holder.

In case (such as the described scenario) the card was created with payload, some additional steps take place in order to store this payload. The Editor requests the adding of payload subsequently to the request for the addition of the card. The insertion of another payload change event (this time for the content card) is therefore triggered, which fires another rule. Within this rule the payload of the α -Card is stored. More detailed descriptions of the attendant rules are given in the following subsections. In figure 7.7, it is assumed that the α -Card is still private, that's why its payload is only stored locally and is not propagated to the other participants. It is further assumed, that the card is still not under version control. If it were, than the incrementing of its

version within the rule, would have triggered the sending of a notification about this adornment change back to the Editor.

7.4.2 Checking the Changeability of Adornments

Before any adornment of an α -Card can be changed, the Editor checks which adornments at all can be altered in respect to the current status of the α -Card. There are three rules that serve this purpose. Depending on the fact whether the content card has already payload or not or if the card has been marked as *deleted*, different adornments are allowed to be altered. These rules are triggered by a `CheckChangeabilityEvent`. An overview of the possible values in each of the cases is given in table 7.4. The adornments that cannot be changed are referred to as *false*. These adornments are therefore inaccessible for the user and blocked from being altered.

Table 7.4: The Changeability of the Adornments

ADORNMENT	NO PAYLOAD	\exists PAYLOAD	DELETED
<code>id</code>	<i>false</i>	<i>false</i>	<i>false</i>
<code>alphaCardName</code>	<i>true</i>	<i>true</i>	<i>false</i>
<code>object</code>	<i>false</i>	<i>false</i>	<i>false</i>
<code>subject</code>	<i>true</i>	<i>false</i>	<i>false</i>
<code>visibility</code>	<i>false</i>	<i>true/false</i>	<i>false</i>
<code>validity</code>	<i>false</i>	<i>true/false</i>	<i>false</i>
<code>version</code>	<i>false</i>	<i>true</i>	<i>false</i>
<code>variant</code>	<i>false</i>	<i>true</i>	<i>false</i>
<code>fundamentalSemanticType</code>	<i>false</i>	<i>false</i>	<i>false</i>
<code>semanticType(\equivAlphaCardType)</code>	<i>true</i>	<i>false</i>	<i>false</i>
<code>syntacticPayloadType</code>	<i>true</i>	<i>true</i>	<i>false</i>
<code>versioning</code>	<i>false</i>	<i>true/false</i>	<i>false</i>
<code>dueDate</code>	<i>true</i>	<i>true</i>	<i>false</i>
<code>deferred</code>	<i>true</i>	<i>true</i>	<i>false</i>
<code>deleted</code>	<i>true</i>	<i>true</i>	<i>true</i>
<code>priority</code>	<i>true</i>	<i>true</i>	<i>false</i>

The adornments, which have both *true* and *false* put in the possible values, can be only altered if the corresponding adornment has not already been set, so their initial value would be *true*. If it has been set once, then it cannot be changed any more afterwards and therefore their value is *false* henceforth.

7.4.3 Changing an Adornment

The adornment model describes the properties of an α -Card. There are many reasons why these properties are not static. For one thing, it is anticipated that events, occurred in the workflow, are going to affect its artifacts, and respectively their adornments. There are certain actions, important for the application, that are especially designed to be triggered through these changes. Besides, it is a functional requirement that it should be possible for the user to change the adornments of an α -Card and thus be able to manipulate the workflow.

When the set of adornments, that can be changed, is settled (after checking the changeability of the adornments), the Editor shows it to the user. Some adornments can only be set once and never be changed again. Others can be altered only after the content card they belong to has payload. There are adornments, that can be set many times, regardless of the content card being payloadless or not. Change adornment requests get into the Working Memory via the α -Properties interface.

If a change request is invoked, a `ChangeAdornmentEvent` is inserted into the Working Memory, which triggers according to the adornment type the change was requested for, the corresponding rule. Its action is on the one hand the altering of the adornment of the α -Card and on the other hand, in case the card is already *public*, the propagating of the same `ChangeAdornmentEvent` object to the other peers. If the request is to set the *visibility* of the card to *public*, then this request is broadcasted directly. There is one special issue though about the rules for *visibility*, *validity* and *versioning*. It shall be deemed that once a card is made *public* or *valid*, it stays *public* or *valid* henceforth - the value of this adornments cannot be reversed from then on. And if version control is started (set *versioning* to true), it is also irreversible.

Within the rules concerned with adornment change requests, versioning of the cards is executed. Version control is activated in two ways. One option is, that the user triggers the versioning explicitly. This can be done at any time, whilst it is still not the case. The second option takes place implicitly. If versioning is not already started by the user, it is enforced at the latest, when the card becomes *valid* in addition to being *public* or vice versa. Either way, once version control is activated, it is irreversible.

If the user wants to trigger versioning for an α -Card, they can do one of the following. The first option is by giving the α -Card a specific initial version value, the α -Properties

ignores the forwarded value, sets the version of the card to "1.0" and activates versioning. The second possibility is by setting *versioning* to true, whereby versioning gets activated but no version number gets incremented. Once versioning is enabled, it is regulated internally. Each time the *version* adornment is altered, this request is interpreted as simple incrementation of its former value.

New versions are generated only if the card is under version control *and* one of the following events takes place: a new payload is added or current payload is altered. The coordination cards are from the beginning on under version control, because they are valid and public per default due to the fact that they should be fully accessible to all peers involved in the the α -Episode.

7.4.4 Changing Payload

There is a group of events that goal to alter the payload of an α -Card. Such an event implies for a coordination card a payload change request, because the payload of the coordination is designed to be extendable. Adding a new content card can be such a payload change request, that will affect the payload of the TSA coordination card. Two content cards may be in a relationship to one another. Defining their relationship is made explicitly with a special payload change request, which also concerns the payload of the TSA coordination card. Expanding the CRA-Payload on demand is also possible. If a new participant is added to the set of actors involved in the workflow or the fields of an existing one are altered, the payload of the CRA coordination card must be respectively changed.

In addition, the payload of a content card must be considered. This payload cannot be actually altered, it can only be replaced with a new variant or version of it. Nevertheless, adding the new variant or version of the payload of a content card is considered also a payload change request and therefore counted to this group of events. Once sent these events match rules of the corresponding group.

The group of rules which handles change payload requests consists of four rules. One rule handles the event of attaching payload to a content card, one manage the addition of a relationship, another one - the addition of a new card, and the fourth concerns the insertion of new participant or the altering of an existing one, if a new port is assigned to this peer.

Adding a Relationship or a Participant or Updating a Participant

By adding a relationship or a participant a `ChangePayloadEvent` is inserted into the Working Memory. There is exactly one rule for each one of the requests that can be fired, depending on the type of payload the `ChangePayloadEvent` was created for. In either case the payload of a coordination card gets modified and the same `ChangePayloadEvent` object is propagated to all the other peers (if the card is *public*). At last, as a part of the action in the RHS of these rules, the `VerVarStore` data is also updated to the new version of the payloads.

Adding a Payload to a Content α -Card

When attaching payload to a content card, the user can put some adornment changes to this request as well. They are chopped up in separate `ChangeAdornmentEvents` and consecutively inserted into the Working Memory, inducing the corresponding rules. Adjoining, the `ChangePayloadEvent` is inserted. As a result the rule in the Knowledge Base for setting the payload of a content card is triggered. There are a few things that take place within the action part of this rule. Namely, the correct version of the α -Card is incremented (if the *versioning* adornment of the card is set to true) and the `VerVarStore` is made aware of the new payload. If *versioning* is still not activated at that time, the new payload is stored in one and the same initial folder and as long as this is the case it gets every time overwritten by the next incoming payload. Again, if the card is already public, the `ChangePayloadEvent` object is forwarded to the other participants.

7.5 Propagating Updates

Summarized the propagation of updates is realized as follows. When an object should be sent out, in the triggered rule a function is invoked, that builds a set of the current peers and sends to each one of them an update-event by invoking the `sendUpdate()` method from the `UpdateServiceSender`. On the receiving side a Drools Pipeline is set up, which accepts these events and inserts them directly into the Working Memory. The pipeline is described in more detail in section 8.2. Figure 7.8 outlines the basic steps of the data transfer.

The `UpdateServiceSender` and the `UpdateServiceReceiver` interfaces are shown in figure 7.9. The Sender interface provides so far the facility to send only objects of

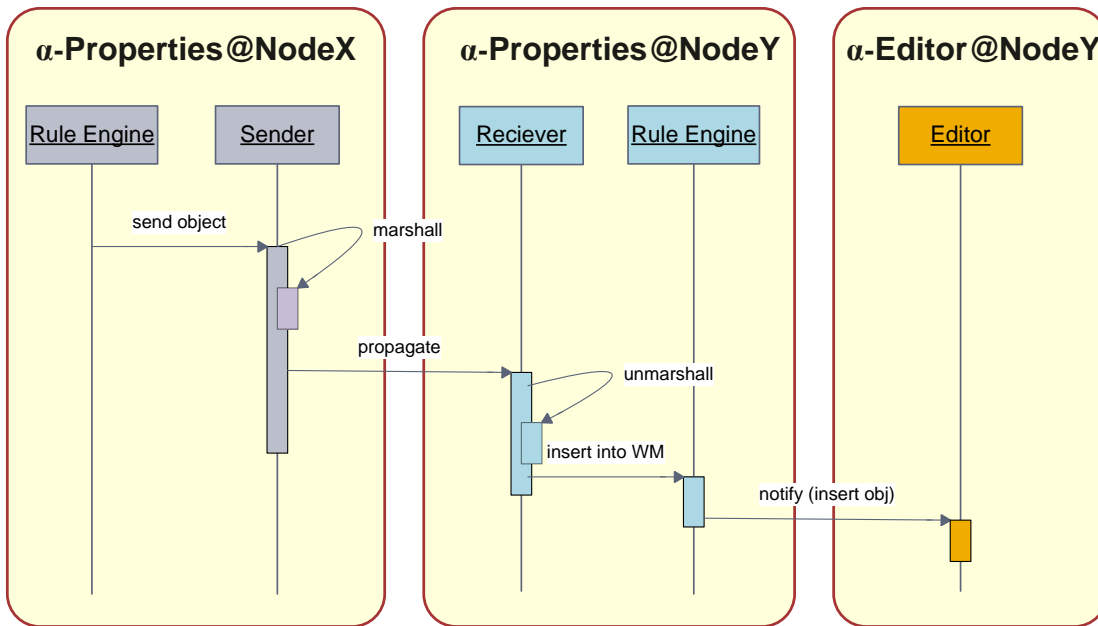


Figure 7.8: The Update Propagation Interaction

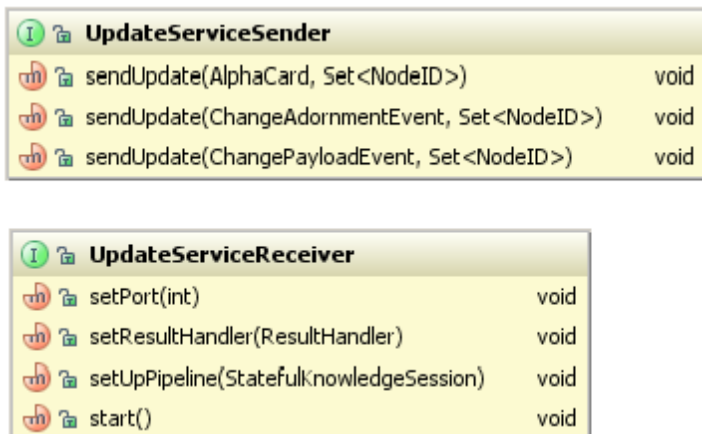


Figure 7.9: The Update Service Interfaces

the type `AlphaCard`, `ChangeAdornmentEvent` or `ChangePayloadEvent` as other types of objects are not conceived to be propagated. The instance of the `Sender` is imported in the `Stateful Session` as a global in order to provide this service within the action part of rules. The `Receiver` on the other hand is instantiated in the `initializeConfig()` method of the `AlphaPropsFacade` and its instance is designed to be running as long as the application is running, thus providing in background the facility to accept continual incoming events.

7.6 Notification

In order to inform the Editor of occurred changes in the state of the objects in the Working Memory, the concept of `WorkingMemoryEventListeners` was applied.

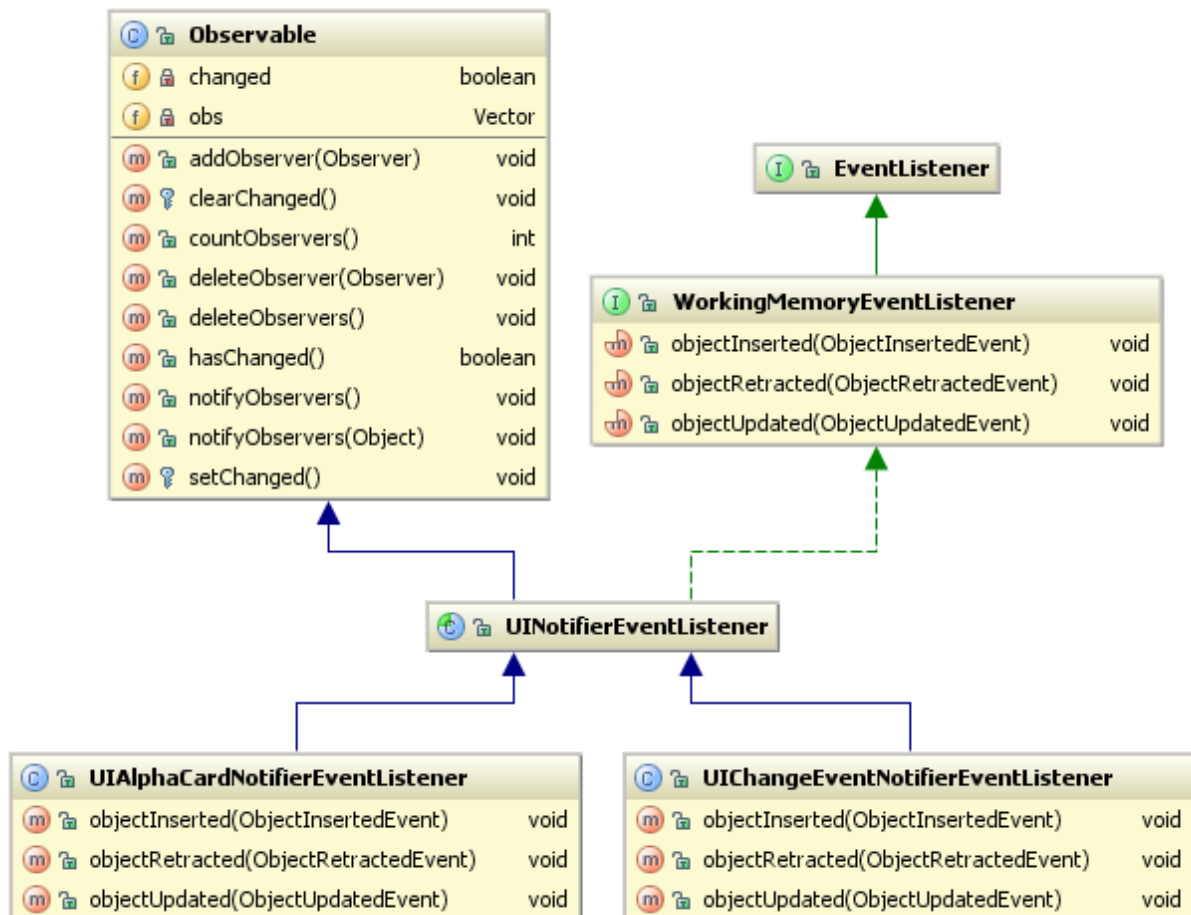


Figure 7.10: The Event Listeners

There are two kind of events that interest the Editor. For one thing, changes in the adornments and the payloads, and for another when in particular an α -Card is altered or created. Therefore two Listeners were developed. The Working Memory event listeners monitor the Working Memory and trace what happens in it. All facts that get inserted, updated or retracted are observed, thus satisfying the need of **Observables** (also known as *Subjects* from the Observer Pattern [GHJV95, HK02]) that communicate

with the Editor and inform it about new states of the objects in the Working Memory. The Observer pattern is retrieved in the concept of the Editor being registered by the `WorkingMemoryEventListeners` as the *Observer*.

There are two implementations of the `WorkingMemoryEventListener` interface: `UIAlphaCardNotifierEventListener` and `UIChangeEventNotifierEventListener` (figure 7.10). The two of them send notifications to the Editor. The implementations extend the `Observable` class. The Editor implements two Observers as well: one that gets notified when an α -Card is added or altered; and one that gets notifications of events that go with adornment change or payload change requests.

7.7 Conclusion

In this chapter the system design of the prototype, implemented for the proposed solution, was explained in detail. A general view of the interaction between the active-properties and the α -Editor was given. The tasks of the α -VerVarStore component and its place in the α -Flow were briefly discussed as well. Furthermore, the system design of the α -Properties module was outlined. Moreover, the system-specific realization of the requirements with the help of rules, Drools pipeline and the use of the Observer Pattern design was demonstrated as well. In the following chapter some particular implementation issues will be revealed in more details.

8 Implementation Issues

In the following some implementation issues of the rules, used in the prototype, and some further technical issues from the α -Properties module will be described. The system design and its implementation present the first prototype for the proposed solution and as such there are some delimitations and open issues. They are amplified in chapter 9.

8.1 The Rule Package

A rule package contains rule definitions, function declarations, queries just to name a few. They are written in DRL and compacted into one *.drl file. The rule package conceived for the prototype makes use of this encapsulation of logic. In this package all rules were defined as well as some useful functions for the propagation and some specific queries. The whole rule package has been listed in the appendix A. This package can be taken into account as the basis for future extensions.

8.1.1 Queries

Drools allows to create queries, which inquire the Working Memory state. Three general queries were defined in the `alpha.props.rules` rule package. They are especially useful for the α -Editor. Every time it refreshes its view it uses two of them: *alphaDoc* and *alphaCards* (see table 8.1). With the returned hits, the Editor acquires the current snapshot of the artifacts and shows them to the user. The third query was created in order to make it possible to retrieve an α -Card by its identifier.

By applying queries it is ensured that only the current state of the objects will be returned. They should be considered as standalone LHS¹ (as of rules), where the engine

¹ Left Hand Side; the condition (*if*-)part of a rule

Table 8.1: The α -Queries

QUERY NAME	DESCRIPTION
alphaDoc	gets the current state of the α -Doc
alphaCards	gets all currently available factHandles from type α -Card
alphaCardByID	gets the current α -Card object that corresponds to this ID

only tries to find a match to the specific conditions and then returns the hits without doing anything else. Queries however provide just one way to implement the facility of gaining information about the artifacts in the Working Memory. One can just as well provide a customer solution.

8.1.2 Globals

In Drools there is the possibility to pass an object to the Working Memory without extra inserting it. For example in order to make a service from the application available to the rule engine and hence be used in the RHS of a rule. As it is conceived that the synchronization of the artifacts and their distribution take place in the action part of the rules, the `UpdateServiceSender` is set in the rule package as a global. This way making it on-hand available to be used in order to send out the artifacts.

8.1.3 Functions

The propagating of adornment or payload changes is an action, which takes place for every adornment and for the payload of both coordination cards and the content cards, as long as the card is *public*. This particular action is therefore invoked over and over again only with different parameters for each rule. For such cases using functions, which keep the logic at one place is preferable. Moreover, if this logic is likely to be changed, this must be done only once - in the function declaration.

Table 8.2: The Functions

FUNCTION NAME	DESCRIPTION
getNodeIDs	gets the network information for all participants
sendAlphaCard	sends out the new α -Card object
propagateAdornmentChange	sends out a change adornment request for an α -Card
propagatePayloadChange	sends out a change payload request for an α -Card

On overview of the functions in the rule package can be taken from table 8.2. The first function helps obtaining the network information of the participants, to whom the events should be broadcasted. The second listed function helps to send out a content card - that is a place-holder or a content card, that has just been made visible (*public*). The last two are responsible for the propagation of adornment change requests and the propagation of altered or new payload respectively.

The functions are all placed within the RHS of the rules, where the actions take place. So, in case the card is public and a change request occurs the corresponding functions will be invoked. The same takes effect, if an α -Card is supposed to be distributed.

8.1.4 Rules

All twenty-four rules, named as they are defined in the rule package, are listed in table 8.3. They are distinguished semantically in four groups (see table 7.2 and in table 8.3 outlined in **bold**). But technically (that is with regard to Drools-Groups) they are divided into only two groups with different scope. Most of the rules are in the scope of the default Agenda Group: **main**. Only three of the rules belong to an Activation Group called "**check changeability**". These rules form an Activation Group, because only one of them can be activated at a time.

The application is designed in a way that not more than one rule can be scheduled to fire per incoming fact. And because there is technically no possibility that two or more rules can be triggered based on the same facts, there is no need of priorities for the rules (there is no salience defined), meaning that the activated rules get fired on the LIFO (last in, first out) principle. Without salience the Agenda is not able to warrant a fix order of execution. If the application expects that and it is not granted, unforeseeable effects may occur.

There is one special rule though, that has high priority - "*once valid and public, start versioning*". If the conditions for this rule get fulfilled, the rule is activated and scheduled for immediate execution. This rule is also in the **main** group. Thus ensuring regardless of which group the focus was last set on, it would always be activated and scheduled on the Agenda for execution.

The following example is based on the prototype implementation which was realized with JBoss Drools, therefore the example is illustrated in the DRL/dialect *java*. It

Table 8.3: The α -Props Rules

RULE NAME	SALIENCE	GROUP
Add an α-Card Rules:	-	
"Add a place-holder content α -Card"	-	main
"Add a content α -Card (PUBLIC)"	-	main
Check Changeability Rules:	-	
"Check Changeability: card does not have payload"	-	check changeability
"Check Changeability: card has payload"	-	check changeability
"Check Changeability: card is marked as deleted"	-	check changeability
Change Adornment Rules:	-	
"Set visibility"	-	main
"Set validity"	-	main
"Once valid and public, start versioning"	50	main
"Set version"	-	main
"Set variant"	-	main
"Set versioning"	-	main
"Set alphaCard type"	-	main
"Set alphaCard name"	-	main
"Set dueDate"	-	main
"Set priority"	-	main
"Set deleted"	-	main
"Set deferred"	-	main
"Set subject (owner of the alphaCard)"	-	main
"Set object (patient)"	-	main
"Set syntactic payload type"	-	main
Change Payload Rules:	-	
"Add a new content alphaCard to the ToDoItems"	-	main
"Add a relationship"	-	main
"Add or update a participant"	-	main
"Add payload to a content alphaCard"	-	main

illustrates a rule that is activated if a request to set a new subject (owner of the card) is inserted in the Working Memory. It triggers therefore this change, and propagates the request to the other participants, if the parameter *propagateChange* is true. This parameter is set internally to true, if the *visibility* of the card is evaluated to *public*. The rest of the rules are listed in the appendix A.


```

1 rule "Set subject (owner of the alphaCard)"
2   no-loop true
3   when
4     craPayload : CRAPayload()
5     cae : ChangeAdornmentEvent(
6       acid : alphaCardID,
7       at : adornmentType,
8       nv : newValue,
9       pc : propagateChange)
10    ac : AlphaCard(id == acid)
11    eval(at.equals(AdornmentType.SUBJECT))
12  then
13    modify(ac) { setSubject((SubjectID)nv) };
14
15    if(pc == true) {
16      propagateAdornmentChange(cae, craPayload,
17        updateServiceSender);
18    }
19    retract(cae);
20  end

```

Listing 8.1: Rule that changes an adornment: sets the subject of an α -Card

8.2 Distribution

The `UpdateServiceSender` interface applies a very primitive use of sockets. For every `sendUpdate()` method call triggered within a rule action, a new socket connection is set (one per recipient). The event is then transformed into XML-bound object and thus prepared for serialization. Eventually, this object is sent out onto the socket. Afterwards the connection is closed. In the implementation of this prototype, the events are broadcasted to all peers involved in the α -Episode. The XML binding is accomplished with a customized *JAXB Marshaller*, which recognizes in its context the object models of the artifacts.

On the receiving side, *UpdateServiceReceiver* interface eavesdrops perpetually for incoming connections. In case there is a request, a new socket is set, the connection

is accepted and forwarded to it by the `ServerSocket`. The incoming payload is then directly inserted into an *incoming pipeline*, that was set up upon the initializing of the α -Properties. There is always one incoming pipeline per stateful session, ready to receive XML-bound events from the listening socket. Once these objects are inserted into the pipeline, they undergo several stages, get transformed into `FactHandles` and are eventually fed directly into the Working Memory. Once in the Working Memory of a new Drools instance, these events trigger the firing of the same rules, so the changes take place as they have in the *ego-peer*. The effects of the updates can be seen on the Editor, as soon as it refreshes its view upon notification from the observed Subjects (see section 7.6).

The term *ego-peer* should be defined briefly here - an *ego-peer* is the participant from whom changes come. Ego-peer is every peer, which represents a participant who initializes changes that affect the flow. These changes get eventually propagated to all the other peers. This is more a role, assigned to those peers which happen to be active at a certain moment, rather than a feature.

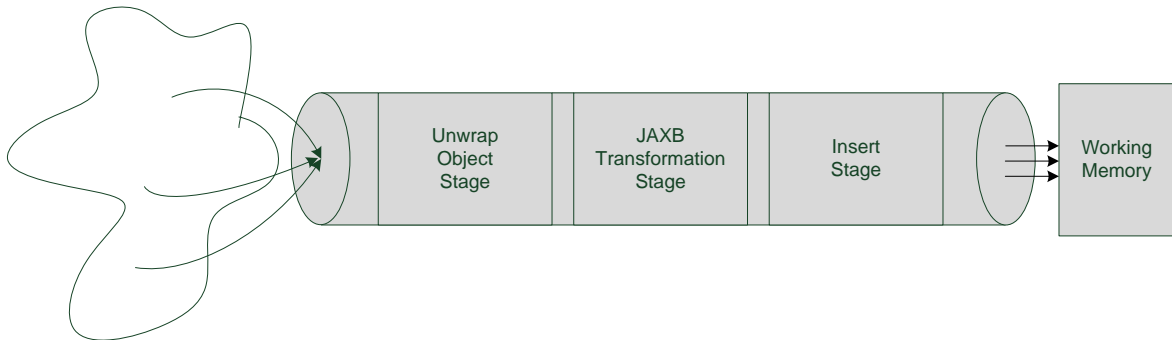


Figure 8.1: The *Incoming Pipeline*

A *pipeline* undergoes many stages, the types of which depend on the application needs (figure 8.1). A pipeline is built from the bottom up. On the top of the pipeline an `ExecuteResultHandler` Action is set, which converts the incoming objects into `FactHandles` and hence makes them available to the rule engine. It serves as the Receiver for the former stage - the *Insert Stage*, where objects are inserted into the Working Memory (or here the `StatefulKnowledgeSession`). The Insert Stage is the Receiver of the next stage - the Transformer Stage. Here the transformer instance and

the transformer stage is created, where XML-bound objects are transformed into POJOs. The transformer is realized with use of an *JAXB Unmarshaller*, implemented from Drools and offered as one of the few predefined Transformers designed for embedding in the Pipeline. And lastly, the Transformer stage sets the Receiver for the entrance stage where the start adapter Pipeline for the `StatefulKnowledgeSession` is created. In this stage the socket payload is inserted.

8.3 Monitoring

In order to distinguish for what changes a notification should be sent, the objects of interest are accordingly annotated. Listing 8.2 shows the simple annotation declared for this purpose. The `@Retention(RetentionPolicy.RUNTIME)` indicates that annotations with this type are to be retained by the Virtual Machine so they can be read reflectively at run-time.

```

1 import java.lang.annotation.Retention;
2 import java.lang.annotation.RetentionPolicy;
3 /**
4  * Denotes if a class will be sent to any Observer if changed within
5  *   an Observable.
6  */
7 @Retention(RetentionPolicy.RUNTIME)
8 public @interface ObservableEvent {
9 }

```

Listing 8.2: Declaration of the `ObservableEvent` Annotation

The `UIAlphaCardNotifierEventListener` sends a notification to the Editor when an α -Card is inserted, retracted or somehow altered. Whereas the `UIChangeEventNotifierEventListener` notifies if an object annotated with the tag `@ObservableEvent` is added, retracted or updated.

8.4 Conclusion

In this chapter some implementation details of the prototype were provided. An initial rule package was conceived for the prototype. By and large changes of the adornments of content cards, a flexible way of extending the payload of coordination cards and propagation of events in a distributed environment are realized with the designed rules. Furthermore, version control of the artifacts is internally considered, as is access control in regard to *private* content cards, which are not supposed to be accessible or be seen by non-owner participants unless they are *public*. Some further issues in regard to the implementation of the distribution and the monitoring were also discussed.

9 Discussion

The following chapter offers a lookout of the future work. Some of the discussed issues have already been conceived but have not yet been fully implemented. Others are not conceptually elaborated but are considered open and important aspects for the architecture. Whereas many of them refer to the α -Flow as a whole, there are some meeting points with the α -Properties module as well.

9.1 Delimitations in the Model

As already mentioned in chapter 6, there are some generalized assumptions made for the prototype implementation. They are distinguished mainly in respect of the communication and in respect of the adornments model.

Concerning the communication, there are three aspects that are considered. First of all, the always-on semantic of the nodes is elevated. It is assumed that none of the peers goes offline and they are always reachable and ready to accept incoming connections. Secondly, so far all participants get informed in case changes occur on an ego-peer. Therefore the changes are broadcasted to all participants involved in the flow. It should be possible though that only a subset of selected parties can be targeted according to whom these changes actually concern. And thirdly, the recipient list extracted from the CRA-Payload is static for now and it can be only extended, if a participant is added during the run-time. The objective is that such recipient lists are created dynamically on demand.

In the prototype the most adornments are designed in a simple way. For example by a `boolean` or through an `enum`. But it is just for the illustrative purposes a prototype should serve. More complex model presentation should be conceived. So far only a generalized model is considered, which is immutable. What is needed is the conversion of the adornments into a dynamically extendable model. Furthermore, with respect to

the adornments, some rules were designed, such as the irreversibility of the *visibility*, *validity* and *versioning* adornments: once they are changed they cannot be reset to their initial values. In addition, there is the firm rule controlling the activation of the version control - namely, it is assured that as soon as a content card becomes both valid and visible (and it already has payload, of course) versioning is turned on. The set of rules that is initially proposed in this prototype merely provides a minimum of exemplary rules, but it sets up a basis for expansion.

9.2 Prototype Optimization Options

9.2.1 Networking

The receiving port is set upon initialization of the α -Properties. It should be possible to change this port dynamically and repeatedly. So far, in order for any changes to take place, a restart of the application is needed.

9.2.2 Clearing up the Working Memory

The permanent `FactHandles` in the Working Memory are not supposed to be technical events, but objects of the domain model; i.e. the content cards, the coordination cards, their payload, the α -Doc and the `VerVarStore` instance. The technical events, described in section 7.3, are items of short existence and they are not supposed to stay in the Working Memory for long. Nevertheless, in case a technical event does not fulfill its task of triggering some rule, it will incorrectly remain in the Working Memory. Measures to avoid such occurrences must be taken. A work-around could be to query the Working Memory whether such technical objects reside within it and force the rule engine to file all rules again. Nonetheless, an explicit exception management should rather be deliberated, which should classify the use-cases that could cause such situations in the first place.

9.2.3 Flow Features Elaboration

Flow provides so called *pluggable work items*, which form the building blocks of a process. They can be manifold combined and extended by custom, domain-specific ones, designed

by domain experts without any technical knowing. Work items are useful for integrating external services as well. Drools offers default implementations for the following tasks already ([Dro10]):

- sending email
- finding files
- FTP
- google calendar
- instant messaging
- REST services
- RSS feeds
- creating archives
- executing system commands
- transforming data

Sometimes the collaboration of humans is needed in processes. As already mentioned Flow makes use of human tasks. A default implementation human task management based on the WS¹-HumanTask is provided ([Dro10]), but must not necessarily be used. As human tasks are just another pluggable work item, any human task management solution can be integrated. As these features could be applied to the concept of α -Flow, they should be elaborated in more detail.

9.3 Rules Management, Dynamic Load of Rules and Rules Propagation

There are should be a way to provide the user with a tool for composing rules themselves and deploying them at run-time. For people with no technical background, there should be the possibility to self-manage the rules within the own node. This includes writing the rules as well as deciding for which parties these new rules should be of interest. Different policies could be determined, appointing the classification of rules as universal or customized.

Rules are based on the domain model, because they use it in the left (*condition*) and sometimes in the right (*action*) side of a rule. The goal of attempting to design flexible and extendable adornment models implies the need of adjustable rules and most of all the need of being able to load them on demand by adding them to the Knowledge

1 Web Services

Base at run-time. The loading of the new rules (or packages of rules) and additionally their regulated propagation to concerned participants should ensue dynamically and the synchronization between the peers, which are involved in the rules propagation, should be granted as well.

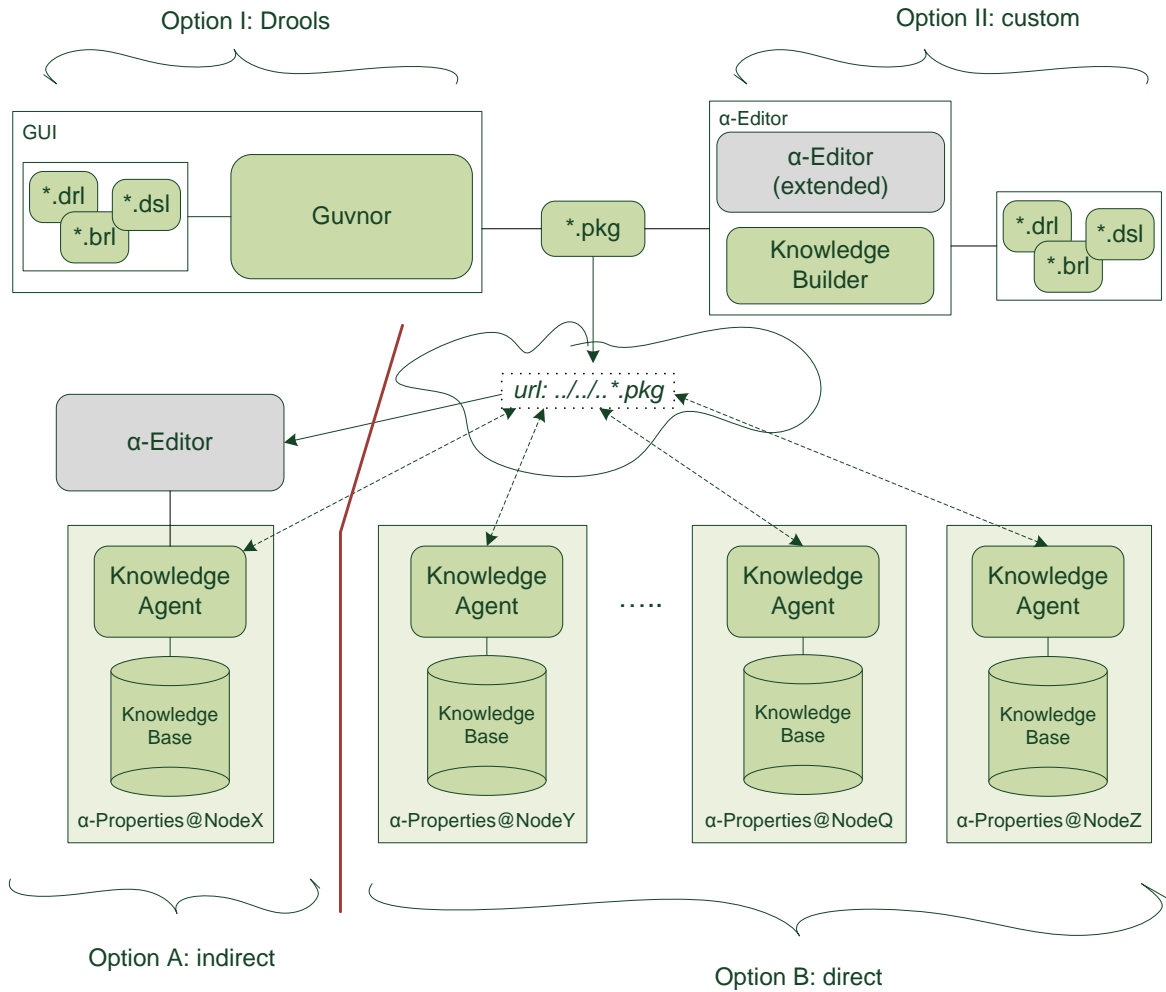


Figure 9.1: Dynamic Rule Loading Proposal

Some proposed scenarios for the solution of these issues are described in figure 9.1. Combinations of options A and B with option I or II are in all variations possible. Options I and II depict two possible solutions for the creation of rules. Drools Guvnor offers a fully implemented platform for rule management. As described in section 5.3.5, Guvnor provides a guided Editor, can manage a great amount of rules and can provide access

control for the users. Guvnor builds the new rules directly in packages and makes them available via an URL, for example. Using Guvnor, however, requires a central installation and an own repository. All in all, a Guvnor distribution does not fit into the α -Flow scenario. The second option on the other hand exposes a custom solution. For example, the existing α -Editor could be extended in order to support an own rule editorial guide. The Knowledge Builder from Drools can be used to build the rule packages. Once rule packages are compiled, they must only be made accessible somewhere.

In chapter 6 was explained that a Knowledge Agent is installed in the α -Properties, which is configured by its initialization to observe resources of knowledge definitions. Any time a registered resource changes, the Knowledge Agent pulls the new version and updates the Knowledge Base at run-time. An optimization of the α -Properties can be made in respect of the memory footprint of the application by providing ready rule packages (*.pkg) in the resources file. Thus, a Knowledge Builder is not needed any more and neither is the corresponding `drools-compiler` library.

Table 9.1: Dynamic Rule Loading Proposal: Options Comparison

CRITERIA	OPTION I	OPTION II
Heaviness	high	depends on the customized solution
Libraries	embedded Guvnor	extension of the α -Editor needed
GUI	provided	extension of the α -Editor needed
CRITERIA	OPTION A	OPTION B
Sovereignty	manually (user)	Drools (not user, not α -Properties)
Control	regulated	no control, no policies
Extendibility	addition of new packages	only at start registered resources

Options A and B expose two different ways of loading the new rules. The direct way (Option B) is by configuring the Knowledge Agent to observe some registered knowledge packages and let it autonomously rebuild the Knowledge Base, when they are changed. This option offers an automatically performed update, which is conducted from Drools in background. Nevertheless, this approach has some disadvantages as well. The most important of them is that if a new resource path is added to the property file of the Knowledge Agent, it is not taken under consideration until a new restart of the application takes place. A possible solution (Option A) would be if the user interface interacts with the Knowledge Base and provides a way for the Editor to feed it

with new Knowledge Packages. Drools offers this function of adding rule packages to an existing Knowledge Base, it must only be assured that they are precompiled. The `addKnowledgePackages()` method can be called iteratively to add additional packages.

The propagation of the new rules could either take place as the action of a triggered rule inside the α -Properties module or be invoked autonomously by the α -Editor itself.

Still another open issue is rights management. There must be some regulations for the rules, as for example who can alter rules and to what extent. Besides, there should be policies about the customized rules that would constitute for whom they should be valid: for the participants of the whole workflow, or just for those involved in the α -Episode, or just for the radiologists, or the like.

9.4 Versioning and Variants Management

All α -Cards have the adornments *version* and *variant*. They have a special purpose in the status of the α -Card and must be treated separately. It is required that an integrated version control module is provided, which organizes the different versions and variants of the documents. It should be possible to rebuild a history of changes of the artifacts backwards any time an older version or a different variant of it is requested.

9.5 Participant Management

As already discussed at the beginning of this chapter, a dynamic recipient list is required and a concept for dynamic node registration is missing. Real case scenarios consider nodes that are mostly offline. As a consequence, a facility for informing the peers of changes when they go online again is needed. The participant management and the node registration is to be settled in another afoot thesis.

9.5.1 Access Control

An open security issue is the assignment of writing permissions. So far only the owner of an α -Card or the participant it was assigned to are allowed to change the card or attach payload to it. In addition, all *private* content cards can be altered only by their creator. That would mean that place-holders cannot be operated on. Up until now, the Editor

is in charge of the access control. Nevertheless, the rights management is likely to be outsourced in the hands of the rule engine in the form of rules. This special rule package should secure the access to cards, if changes of artifacts are requested.

9.5.2 Assignment of Tokens

There are two special roles that can be assigned to participants in the distributed system regarding the global workflow. There should be a token for the peer, who starts the process in the first place: the *process initiator* and another one for the peer, who dictates the further steps in the flow: the *spokesman*. The latter is transferable. It is always in the hands of that peer, who is temporarily in charge of the workflow, and only as long as it makes sense in the current episode phase, that he/she is appointed this role. Facilities are required to create both tokens, and to allow for passing the spokesman token on to the another peers.

9.6 Data Synchronization

Data synchronization regards the attempt to keep multiple copies of a data set coherent with one another. It refers to the idea of maintaining data integrity. One possible solution for the management of distributed incoming requests offer *Lamport timestamps* [Lam78]. Data synchronization, realized with the help of technical versions, lock protocols and distributed timestamps, should be considered. There are distributed version control systems, which detect differentiations between copies and perform merging of their versions. There is a need of an application-specific solution for the α -Flow artifacts, which implements similar features and hence realizes synchronization of data, manipulated by distributed change events.

Beside the technical aspects, there is another issue referring policy assignments: the order of the events. It should be considered, for instance, whether always the youngest will be taken and the older ones ignored, or the FIFS (first in, first served) principle should be applied. Furthermore, it should be determined what kind of changes are allowed in general and to whom. The status quo is that *one* participant is treated as a representative of *one* node. In reality *many* nodes are assigned to *one* participant, which could lead to the conflict that more than one node try to write at a certain time. There

should be a mechanism that appoints which one of the participant's nodes has the right to execute changes. Further, the synchronization among the nodes is required. Thereby locks offer one possible solution.

9.7 Thread Synchronization and Race Conditions

In order to accept all incoming requests from many simultaneously sending peers, a devoted thread was set *per* request at the receiving side of the peer. It was detected that by doing so, the threads get executed in an unpredictable order. Such race conditions occur, for example, when a `ChangeAdornmentEvent`, which is much smaller than the payload of a content card, is sent after the payload, but reaches its destination before it. A possible solution for an enforced sequential execution of the threads at the receiving end in the same order as they are created, is using a queue or some kind of buffer to store the threads and then start them one after the other. As a work-around for this, the `UpdateServiceReceiver` does not create for every accepted socket connection a new thread any more, but confides in sequential arrival of the requests. Unfortunately, in a network of multiple connected peers this cannot always be the case. So, an improvement is hereto needed perhaps, as suggested, with the help of a buffer.

Sending multiple payloads out of the sockets into the network and expecting them to arrive on the other side in the same order they left the source is not possible. It often comes to race conditions on the way and the network payloads arrive in arbitrary order. But sometimes the order is of great importance. It can happen that by insertion of the transported units different rules can be matched, depending on what the former arrived payloads were.

9.8 Persistence

Along with its function as a version manager, the `VerVarStore` module is responsible for storing the artifacts and their payloads. A different approach for the persistence is proposed by shifting this responsibility to the `α -Properties` module. So far it is document-based and time-consuming. But the storage management of the artifacts could be in fact conducted already in the Working Memory. Hence the persistence

becomes under the control of Drools. In order to persist objects from the Working Memory one can use the Java Persistence API (JPA) and a Java Transaction API (JTA) implementation, organizing the data eventually in an embedded database. Along with it come all advantages databases provide, such as full support for the ACID¹ guarantees.

9.9 Network Security

Artifacts are sent from one participant to the other through intricate networks. In distributed systems, where sensible data exchange takes place, network security is a great issue. Furthermore, the transport in the networks, despite potential firewalls, must be possible. Among others, it should be assured that the data is transferred and accepted unforged. Cryptographic algorithms could be used to encrypt the sent network payload. The non-repudiation of the received data must be granted, e. g. by signing the data could ensure its accountability. Additionally, the confidentiality of the transferred artifacts in the context of the sensible nature of the patient's private data and the proper access control over this data must be warranted as well. The integrity of the data as a whole and its availability by all means should be assured as well. As none of these security issues was covered in the prototype design, they are still subject for future work on the α -Overlay-Network component.

9.10 Import and Export von "Process Templates"

Common procedures for treatment episodes which take place always by the book can be used for creating "*process templates*". After defining a treatment workflow of that kind, it could be useful to export it as a template. α -Flows (or distinct α -Episodes) can be reproduced based on the coordination cards. A possible form for such templates could be the TSA-Payload document without the actual cards, but only with references of the needed ones. When such templates are available, healthcare workers can optimize the treatment process by just importing them and starting up the workflow. The same could be applied for the CRA-Payload, if a certain group of participants are used to

1 ACID - Atomicity, Consistency, Isolation, Durability

be working together in the scope of particular treatment episodes. Such "nice to have" features can be applicable in case of pre-structured workflows.

10 Conclusion

Exchange of patient's personal data and treatment-associated documents in healthcare requires an infrastructure, that suffices the impediments distributed heterogeneous systems bring along. A document-oriented workflow approach was considered, which focuses on the relationship between content and coordination aspects of collaborative and inter-institutional environments. The α -Flow model assigns collaboration issues to the documents - the artifacts that constitute the primary means of information exchange. A part of this infrastructure is a rule-based system, called α -Properties. The α -Properties reside within a document. They contribute to its autonomy and liveliness and assemble it to an active document. Such documents are the α -Docs. The models of content and coordination α -Cards were also explained. The cards represent the mapping from the classical paper-based medical records with some meta process-relevant data in the form of adornments.

The feasibility of the concept is proven by a reference implementation of the α -Properties. The module follows the adopted functional requirements, detailed in chapter 4. This module orchestrates logic in the form of predefined knowledge (models and rules) and deals with the execution of actions, occurred as a result of triggered rules. Its main contributions are dynamically changing of the α -Cards metadata, the management of versions of the artifacts and the propagation of changes and other units to all concerned parties in the workflow. The α -Properties were designed to understand requests for workflow extension by adding new artifacts dynamically to the list of existing content cards or requests for updating the metadata and payload of the α -Card. Apart from the events, caused by the user, there are several internal events that are triggered by the α -Properties itself, as a unit of autonomy. These internal events, activated within the module, are result of the reasoning over the current state of the facts. Therefore, requests for changes by the user are interpreted individually. As a result, the changes that take place are sometimes only local, but sometimes they get propagated also to the

other participants. In order to keep the users informed about what actions their requests have triggered, notifications are sent to them.

The design of the α -Flow results in a comprehensive impact on the way treatment episodes are organized in medicine. A dynamic evolution of workflow steps in combination of fine-grained control units (α -Docs) satisfy the need for retained autonomy of the peers in distributed systems. The α -Properties offer a solution, which can be considered for any infrastructure that presumes its artifacts to be active documents that maintain built-in functionality and support distributed systems in their heterogeneous and complex nature.

A The Rule Package

```
1 package alpha.props.rules
2
3 #import classes
4 import org.drools.WorkingMemory
5 import java.util.LinkedHashSet
6 import java.util.HashSet
7 import java.util.Set
8
9 import alpha.adornment.AdornmentType
10 import alpha.adornment.AlphaCardType
11 import alpha.adornment.FundamentalSemanticType
12 import alpha.adornment.Priority
13 import alpha.adornment.Visibility
14 import alpha.adornment.Validity
15 import alpha.eventfact.AddAlphaCardEvent
16 import alpha.eventfact.ChangeAdornmentEvent
17 import alpha.eventfact.ChangePayloadEvent
18 import alpha.eventfact.CheckChangeabilityEvent
19 import alpha.model.AlphaCard
20 import alpha.model.AlphaDoc
21 import alpha.model.AlphaCardRelationship
22 import alpha.model.NodeID
23 import alpha.model.ObjectID
24 import alpha.model.Participant
25 import alpha.model.SubjectID
26 import alpha.model.identification.AlphaCardIdentifier
27 import alpha.payload.Payload
28 import alpha.payload.tsa.TSAPayload
29 import alpha.payload.cra.CRAPayload
30 import alpha.service.impl.AlphaPropsFacadeImpl
31 import alpha.service.UpdateServiceSender
```

```

32 import alpha.services.VerVarStore
33 import alpha.vvs.VerVarStoreImpl
34 import alpha.utility.StringWrapper
35
36 #queries
37 query "alphaDoc"
38     ad : AlphaDoc()
39 end
40
41 query "alphaCards"
42     ac : AlphaCard()
43 end
44
45 query "alphaCardByID" (AlphaCardIdentifier acid)
46     ac : AlphaCard(id == acid)
47 end
48
49 #global variables
50 global
51     UpdateServiceSender updateServiceSender
52
53 #functions
54 function Set getNodeIDs(CRAPayload craPayload) {
55     Set<NodeID> nodeIDs = new HashSet<NodeID>();
56     for (Participant participant : craPayload.getLoParticipants())
57     {
58         nodeIDs.add(participant.getNode());
59     }
60     return nodeIDs;
61 }
62
63 function void sendAlphaCard(AlphaCard ac, CRAPayload craPayload,
64     UpdateServiceSender updateServiceSender) {
65     updateServiceSender.sendUpdate(ac, getNodeIDs(craPayload));
66 }
67
68 function void propagateAdornmentChange(ChangeAdornmentEvent cae,
69     CRAPayload craPayload, UpdateServiceSender updateServiceSender) {
70     cae.setPropagateChange(false);

```

```

68     updateServiceSender.sendUpdate(cae, getNodeIDs(craPayload));
69 }
70
71 function void propagatePayloadChange(ChangePayloadEvent cpe,
72     CRAPayload craPayload, UpdateServiceSender updateServiceSender) {
73     cpe.setPropagateChange(false);
74     updateServiceSender.sendUpdate(cpe, getNodeIDs(craPayload));
75 }
76 #####
77 ### Add an AlphaCard Rules #####
78 #####
79 rule "Add_a_place-holder_content_alphaCard"
80     no-loop true
81     when
82         craPayload : CRAPayload()
83         aace : AddAlphaCardEvent( ac : alphaCard )
84         tsa : AlphaCard(tsaAcid : id)
85         eval(tsaAcid.getCardID().equals("$tsa"))
86     then
87         insert(ac);
88
89         ChangePayloadEvent cpe = new ChangePayloadEvent(tsaAcid, ac.
90             getId());
91         cpe.setPropagateChange(true);
92         insert(cpe);
93
94         sendAlphaCard(ac, craPayload, updateServiceSender);
95         retract(aace);
96
97 end
98
99 rule "Add_a_content_alphaCard_(PUBLIC)"
100     no-loop true
101     when
102         vvs : VerVarStoreImpl()
103         ac : AlphaCard(acid : id, vis : visibility)
104         eval(vis == Visibility.PUBLIC)
105         _ac : AlphaCard( id == acid && visibility == Visibility.
106             PRIVATE)

```

```

104     then
105         retract(_ac);
106 end
107
108 #####
109 ### Check Changeability Rules #####
110 #####
111 rule "Check Changeability: card does not have payload"
112     no-loop true
113     activation-group "check changeability"
114     when
115         chre : CheckChangeabilityEvent(acid : alphaCardID, hp :
116             hasPayload)
117         ac : AlphaCard( id == acid )
118         eval(hp == true)
119     then
120         chre.getChangeables().
121             put(AdornmentType.SUBJECT, Boolean.TRUE);
122         chre.getChangeables().
123             put(AdornmentType.TITLE, Boolean.TRUE);
124         chre.getChangeables().
125             put(AdornmentType.ALPHACARDTYPE, Boolean.TRUE);
126         chre.getChangeables().
127             put(AdornmentType.FUNDAMENTALSEMANTICTYPE, Boolean.FALSE);
128         chre.getChangeables().
129             put(AdornmentType.VALIDITY, Boolean.FALSE);
130         chre.getChangeables().
131             put(AdornmentType.VISIBILITY, Boolean.FALSE);
132         chre.getChangeables().
133             put(AdornmentType.VARIANT, Boolean.FALSE);
134         chre.getChangeables().
135             put(AdornmentType.VERSION, Boolean.FALSE);
136         chre.getChangeables().
137             put(AdornmentType.SPTYPE, Boolean.TRUE);
138         chre.getChangeables().
139             put(AdornmentType.OBJECT, Boolean.FALSE);
140         chre.getChangeables().
141             put(AdornmentType.VERSIONING, Boolean.FALSE);
142         chre.getChangeables().

```

```

142         put(AdornmentType.DUEDATE, Boolean.TRUE);
143     chre.getChangeables().
144         put(AdornmentType.DEFERRED, Boolean.TRUE);
145     chre.getChangeables().
146         put(AdornmentType.DELETED, Boolean.TRUE);
147     chre.getChangeables().
148         put(AdornmentType.PRIORITY, Boolean.TRUE);
149     update(chre);
150 end
151
152 rule "Check_Changeability:_card_has_payload"
153     no-loop true
154     activation-group "check_changeability"
155     when
156         chre : CheckChangeabilityEvent(acid : alphaCardID, hp :
157             hasPayload)
158         ac : AlphaCard( id == acid )
159         eval(hp == false)
160     then
161         chre.getChangeables().
162             put(AdornmentType.SUBJECT, Boolean.FALSE);
163         chre.getChangeables().
164             put(AdornmentType.TITLE, Boolean.TRUE);
165         chre.getChangeables().
166             put(AdornmentType.ALPHACARDTYPE, Boolean.FALSE);
167         chre.getChangeables().
168             put(AdornmentType.FUNDAMENTALSEMANTICTYPE, Boolean.FALSE);
169         chre.getChangeables().
170             put(AdornmentType.OBJECT, Boolean.FALSE);
171
172         if(ac.getVisibility() == Visibility.PUBLIC) {
173             chre.getChangeables().
174                 put(AdornmentType.VISIBILITY, Boolean.FALSE);
175         } else {
176             chre.getChangeables().
177                 put(AdornmentType.VISIBILITY, Boolean.TRUE);
178         }
179
180         if(ac.getValidity() == Validity.VALID) {

```

```

180         chre.getChangeables().
181             put(AdornmentType.VALIDITY, Boolean.FALSE);
182     } else {
183         chre.getChangeables().
184             put(AdornmentType.VALIDITY, Boolean.TRUE);
185     }
186
187     chre.getChangeables().
188         put(AdornmentType.VARIANT, Boolean.TRUE);
189     chre.getChangeables().
190         put(AdornmentType.VERSION, Boolean.TRUE);
191     chre.getChangeables().
192         put(AdornmentType.SPTYPE, Boolean.TRUE);
193
194     if(ac.isVersioning()) {
195         chre.getChangeables().
196             put(AdornmentType.VERSIONING, Boolean.FALSE);
197     } else {
198         chre.getChangeables().
199             put(AdornmentType.VERSIONING, Boolean.TRUE);
200     }
201
202     chre.getChangeables().
203         put(AdornmentType.DUEDATE, Boolean.TRUE);
204     chre.getChangeables().
205         put(AdornmentType.DEFERRED, Boolean.TRUE);
206     chre.getChangeables().
207         put(AdornmentType.DELETED, Boolean.TRUE);
208     chre.getChangeables().
209         put(AdornmentType.PRIORITY, Boolean.TRUE);
210     update(chre);
211 end
212
213 rule "Check Changeability: card is marked as deleted"
214     no-loop true
215     activation-group "check changeability"
216     when
217         chre : CheckChangeabilityEvent(acid : alphaCardID, hp :
                hasPayload)

```

```

218     ac : AlphaCard( id == acid )
219     eval(ac.isDeleted() == true)
220     then
221     alpha.adornment.AdornmentType[] types = alpha.adornment.
        AdornmentType.class.getEnumConstants();
222     for(int i = 0; i < types.length; i++) {
223         if(types[i].equals(AdornmentType.DELETED)) {
224             chre.getChangeables().put(types[i], Boolean.TRUE);
225         } else {
226             chre.getChangeables().put(types[i], Boolean.FALSE);
227         }
228     }
229 end
230
231 #####
232 ### Change Adornments Rules #####
233 #####
234 rule "Set_visibility"
235     no-loop true
236     when
237         craPayload : CRAPayload()
238         vvs : VerVarStoreImpl()
239         cae : ChangeAdornmentEvent(acid : alphaCardID, at :
            adornmentType, nv : newValue, pc : propagateChange)
240         ac : AlphaCard(id == acid, vis : visibility)
241         eval(at.equals(AdornmentType.VISIBILITY) && (vis == Visibility
            .PRIVATE || vis == null))
242     then
243         modify(ac) { setVisibility((Visibility)nv) };
244         if(pc == true && ((Visibility)nv).equals(Visibility.PUBLIC)) {
245             sendAlphaCard(ac, craPayload, updateServiceSender);
246
247             if(vvs.getPayload(acid) != null) {
248                 ChangePayloadEvent cpe = new ChangePayloadEvent(acid,
                    vvs.getPayload(acid));
249                 propagatePayloadChange(cpe, craPayload,
                    updateServiceSender);
250             }
251     }

```

```

252         retract(cae);
253     end
254
255
256 rule "Once valid and public, start versioning"
257     no-loop true
258     salience 50
259     when
260         ac : AlphaCard()
261         eval(ac.getVisibility() == Visibility.PUBLIC && ac.getValidity
262             () == Validity.VALID && !ac.isVersioning())
263     then
264         #start versioning from now on
265         modify(ac) { setVersioning(true) };
266     end
267
268 rule "Set validity"
269     no-loop true
270     when
271         craPayload : CRAPayload()
272         cae : ChangeAdornmentEvent(acid : alphaCardID, at :
273             adornmentType, nv : newValue, pc : propagateChange)
274         ac : AlphaCard(id == acid, val : validity)
275         eval(at.equals(AdornmentType.VALIDITY) && (val == Validity.
276             INVALID || val == null))
277     then
278         modify(ac) { setValidity((Validity)nv) };
279
280         if(pc == true) {
281             propagateAdornmentChange(cae, craPayload,
282                 updateServiceSender);
283         }
284
285         retract(cae);
286     end
287
288 #new values sent from user will be ignored!
289 rule "Set version"
290     no-loop true

```



```

287   when
288       craPayload : CRAPayload()
289       vvs : VerVarStoreImpl()
290       cae : ChangeAdornmentEvent(acid : alphaCardID, at :
          adornmentType, pc : propagateChange)
291       ac : AlphaCard(id == acid)
292       eval(at.equals(AdornmentType.VERSION))
293   then
294       StringWrapper _nv = new StringWrapper(String.valueOf((Double.
          valueOf(ac.getVersion()).doubleValue() + 1)));
295
296       if (!ac.isVersioning()) {
297           modify(ac) { setVersioning(true) };
298       }
299       #new version
300       modify(ac) { setVersion(_nv.getNewValue()) };
301
302       if(pc == true) {
303           cae.setNewValue(_nv);
304           propagateAdornmentChange(cae, craPayload,
          updateServiceSender);
305       }
306       retract(cae);
307
308       if(vvs.getPayload(acid) != null) {
309           vvs.setSptype(ac.getSPType());
310           vvs.setVersion(String.valueOf((Double.valueOf(ac.
          getVersion()).doubleValue())));
311           vvs.putPayload(acid, vvs.getPayload(acid));
312       }
313   end
314
315   rule "Set_variant"
316       no-loop true
317       when
318           craPayload : CRAPayload()
319           cae : ChangeAdornmentEvent(acid : alphaCardID, at :
          adornmentType, nv : newValue, pc : propagateChange)
320           ac : AlphaCard(id == acid)

```

```

321     eval(at.equals(AdornmentType.VARIANT))
322     then
323     if(ac.getVariant() == null || ac.getVariant().equals("-")) {
324         ac.setVariant("0");
325     }
326     if(nv instanceof StringWrapper && (Integer.valueOf(((
327         StringWrapper)nv).getNewValue())) > Integer.valueOf(ac.
328         getVariant())) ) {
329         modify(ac) { setVariant(((StringWrapper)nv).getNewValue())
330             };
331
332         if(pc == true) {
333             propagateAdornmentChange(cae, craPayload,
334                 updateServiceSender);
335         }
336     } else if (nv instanceof String && (Integer.valueOf((String)nv
337         ) > Integer.valueOf(ac.getVariant())) ) {
338         StringWrapper _nv = new StringWrapper((String)nv);
339         modify(ac) { setVariant(_nv.getNewValue()) };
340
341         if(pc == true) {
342             cae.setNewValue(_nv);
343             propagateAdornmentChange(cae, craPayload,
344                 updateServiceSender);
345         }
346     }
347     retract(cae);
348 end
349
350 rule "Set_dueDate"
351     no-loop true
352     when
353         craPayload : CRAPayload()
354         cae : ChangeAdornmentEvent(acid : alphaCardID, at :
355             adornmentType, nv : newValue, pc : propagateChange)
356         ac : AlphaCard(id == acid)
357     eval(at.equals(AdornmentType.DUEDATE))
358     then
359     if(nv instanceof StringWrapper) {

```

```

353         modify(ac) { setDueDate (((StringWrapper)nv).getNewValue()
354             ) };
355
356         if(pc == true) {
357             propagateAdornmentChange(cae, craPayload,
358                 updateServiceSender);
359         }
360     } else {
361         StringWrapper _nv = new StringWrapper((String)nv);
362         modify(ac) { setDueDate(_nv.getNewValue()) };
363
364         if(pc == true) {
365             cae.setNewValue(_nv);
366             propagateAdornmentChange(cae, craPayload,
367                 updateServiceSender);
368         }
369     }
370     retract(cae);
371 end
372
373 rule "Set_□priority"
374     no-loop true
375     when
376         craPayload : CRAPayload()
377         cae : ChangeAdornmentEvent(acid : alphaCardID, at :
378             adornmentType, nv : newValue, pc : propagateChange)
379         ac : AlphaCard(id == acid)
380         eval(at.equals(AdornmentType.PRIORITY))
381     then
382         modify(ac) { setPriority((Priority)nv) };
383
384         if(pc == true) {
385             propagateAdornmentChange(cae, craPayload,
386                 updateServiceSender);
387         }
388     }
389     retract(cae);
390 end
391
392 rule "Set_□deleted"

```

```

387     no-loop true
388     when
389         craPayload : CRAPayload()
390         cae : ChangeAdornmentEvent(acid : alphaCardID, at :
            adornmentType, nv : newValue, pc : propagateChange)
391         ac : AlphaCard(id == acid)
392         eval(at.equals(AdornmentType.DELETED))
393     then
394         if(nv instanceof StringWrapper) {
395             modify(ac) { setDeleted(Boolean.valueOf(((StringWrapper)nv
                ).getNewValue())) };
396
397             if(pc == true) {
398                 propagateAdornmentChange(cae, craPayload,
                updateServiceSender);
399             }
400         } else {
401             StringWrapper _nv = new StringWrapper(String.valueOf(nv));
402             modify(ac) { setDeleted(Boolean.valueOf(_nv.getNewValue()
                ) ) };
403
404             if(pc == true) {
405                 cae.setNewValue(_nv);
406                 propagateAdornmentChange(cae, craPayload,
                updateServiceSender);
407             }
408         }
409         retract(cae);
410     end
411
412     rule "Set_deferred"
413     no-loop true
414     when
415         craPayload : CRAPayload()
416         cae : ChangeAdornmentEvent(acid : alphaCardID, at :
            adornmentType, nv : newValue, pc : propagateChange)
417         ac : AlphaCard(id == acid)
418         eval(at.equals(AdornmentType.DEFERRED))
419     then

```

```

420     if(nv instanceof StringWrapper) {
421         modify(ac) { setDeferred(Boolean.valueOf(((StringWrapper)
422             nv).getNewValue())) };
423
424         if(pc == true) {
425             propagateAdornmentChange(cae, craPayload,
426                 updateServiceSender);
427         }
428     } else {
429         StringWrapper _nv = new StringWrapper(String.valueOf(nv));
430         modify(ac) { setDeferred(Boolean.valueOf(_nv.getNewValue()
431             )) };
432
433         if(pc == true) {
434             cae.setNewValue(_nv);
435             propagateAdornmentChange(cae, craPayload,
436                 updateServiceSender);
437         }
438     }
439     retract(cae);
440 end
441
442 rule "Set␣object␣(patient)"
443     no-loop true
444     when
445         craPayload : CRAPayload()
446         cae : ChangeAdornmentEvent(acid : alphaCardID, at :
447             adornmentType, nv : newValue, pc : propagateChange)
448         ac : AlphaCard(id == acid)
449         eval(at.equals(AdornmentType.OBJECT))
450     then
451         modify(ac) { setObject((ObjectID)nv) };
452
453         if(pc == true) {
454             propagateAdornmentChange(cae, craPayload,
455                 updateServiceSender);
456         }
457     }
458     retract(cae);
459 end

```

```

453
454 rule "Set subject of the alphaCard"
455     no-loop true
456     when
457         craPayload : CRAPayload()
458         cae : ChangeAdornmentEvent(acid : alphaCardID, at :
459             adornmentType, nv : newValue, pc : propagateChange)
460         ac : AlphaCard(id == acid)
461         eval(at.equals(AdornmentType.SUBJECT))
462     then
463         modify(ac) { setSubject((SubjectID)nv) };
464
465         if(pc == true) {
466             propagateAdornmentChange(cae, craPayload,
467                 updateServiceSender);
468         }
469         retract(cae);
470     end
471
472 rule "Set alphaCard name"
473     no-loop true
474     when
475         craPayload : CRAPayload()
476         cae : ChangeAdornmentEvent(acid : alphaCardID, at :
477             adornmentType, nv : newValue, pc : propagateChange)
478         ac : AlphaCard(id == acid)
479         eval(at.equals(AdornmentType.TITLE))
480     then
481         if(nv instanceof StringWrapper) {
482             modify(ac) { setAlphaCardName(((StringWrapper)nv).
483                 getNewValue()) };
484
485             if(pc == true) {
486                 propagateAdornmentChange(cae, craPayload,
487                     updateServiceSender);
488             }
489         } else {
490             StringWrapper _nv = new StringWrapper((String)nv);
491             modify(ac) { setAlphaCardName(_nv.getNewValue()) };
492         }
493     end

```

```

487
488         if(pc == true) {
489             cae.setNewValue(_nv);
490             propagateAdornmentChange(cae, craPayload,
491                                     updateServiceSender);
492         }
493     }
494     retract(cae);
495 end
496 rule "Set syntactic payload type"
497     no-loop true
498     when
499         craPayload : CRAPayload()
500         cae : ChangeAdornmentEvent(acid : alphaCardID, at :
501                                   adornmentType, nv : newValue, pc : propagateChange)
502         ac : AlphaCard(id == acid)
503         eval(at.equals(AdornmentType.SPTYPE))
504     then
505         if(nv instanceof StringWrapper) {
506             modify(ac) { setSPTYPE(((StringWrapper)nv).getNewValue())
507                             };
508
509             if(pc == true) {
510                 propagateAdornmentChange(cae, craPayload,
511                                         updateServiceSender);
512             }
513         } else {
514             StringWrapper _nv = new StringWrapper((String)nv);
515             modify(ac) { setSPTYPE(_nv.getNewValue()) };
516
517             if(pc == true) {
518                 cae.setNewValue(_nv);
519                 propagateAdornmentChange(cae, craPayload,
520                                         updateServiceSender);
521             }
522         }
523     }
524     retract(cae);
525 end

```

```

521
522 rule "Set_alphaCard_type"
523     no-loop true
524     when
525         craPayload : CRAPayload()
526         cae : ChangeAdornmentEvent(acid : alphaCardID, at :
527             adornmentType, nv : newValue, pc : propagateChange)
528         ac : AlphaCard(id == acid)
529         eval(at.equals(AdornmentType.ALPHACARDTYPE))
530     then
531         modify(ac) { setAlphaCardType((AlphaCardType)nv) };
532
533         if(pc == true) {
534             propagateAdornmentChange(cae, craPayload,
535                 updateServiceSender);
536         }
537     retract(cae);
538 end
539
540 rule "Set_versioning"
541     no-loop true
542     when
543         craPayload : CRAPayload()
544         cae : ChangeAdornmentEvent(acid : alphaCardID, at :
545             adornmentType, nv : newValue, pc : propagateChange)
546         ac : AlphaCard(id == acid)
547         eval(at.equals(AdornmentType.VERSIONING))
548     then
549         if(nv instanceof StringWrapper) {
550             modify(ac) { setVersioning(Boolean.valueOf(((StringWrapper)
551                 nv).getNewValue())) };
552
553             if(pc == true) {
554                 propagateAdornmentChange(cae, craPayload,
555                     updateServiceSender);
556             }
557         } else {
558             StringWrapper _nv = new StringWrapper(String.valueOf(nv));

```



```

554         modify(ac) { setVersioning(Boolean.valueOf(_nv.getNewValue
555             ())) };
556
557         if(pc == true) {
558             cae.setNewValue(_nv);
559             propagateAdornmentChange(cae, craPayload,
560                 updateServiceSender);
561         }
562     }
563     retract(cae);
564 end
565
566 #####
567 ### Change Payload Rules #####
568 #####
569 rule "Add a new content alphaCard to the ToDoItems"
570     when
571         vvs : VerVarStoreImpl()
572         tsaPayload : TSAPayload()
573         craPayload : CRAPayload()
574         cpe : ChangePayloadEvent( acid : alphaCardID, o : obj, pc :
575             propagateChange )
576         ac : AlphaCard( id == acid )
577         eval(acid.getCardID().equals("$tsa") && o instanceof
578             AlphaCardIdentifier)
579     then
580         tsaPayload.getLoToDoItems().add((AlphaCardIdentifier)o);
581
582         #new version
583         modify(ac) { setVersion(String.valueOf((Double.valueOf(ac.
584             getVersion()).doubleValue() + 1)))}
585
586         if(pc == true) {
587             propagatePayloadChange(cpe, craPayload,
588                 updateServiceSender);
589         }
590
591         vvs.setSptype("xml");
592         vvs.setVersion(ac.getVersion());

```

```

587     vvs.putPayload(acid, tsaPayload);
588
589     retract(cpe);
590 end
591
592 rule "Add_a_relationship"
593     when
594         vvs : VerVarStoreImpl()
595         tsaPayload : TSAPayload()
596         craPayload : CRAPayload()
597         cpe : ChangePayloadEvent( acid : alphaCardID, o : obj, pc :
                    propagateChange )
598         ac : AlphaCard( id == acid )
599         eval(acid.getCardID().equals("$tsa") && o instanceof
                    AlphaCardRelationship)
600     then
601         tsaPayload.getLoTodoRelationships().add((AlphaCardRelationship
                    )o);
602         #new version
603         modify(ac) { setVersion(String.valueOf((Double.valueOf(ac.
                    getVersion()).doubleValue() + 1)))}
604
605         if(pc == true) {
606             propagatePayloadChange(cpe, craPayload,
                    updateServiceSender);
607         }
608
609         vvs.setSptype("xml");
610         vvs.setVersion(ac.getVersion());
611         vvs.putPayload(acid, tsaPayload);
612
613         retract(cpe);
614 end
615
616 rule "Add_or_update_a_participant"
617     when
618         vvs : VerVarStoreImpl()
619         craPayload : CRAPayload()

```

```

620     cpe : ChangePayloadEvent( acid : alphaCardID, o : obj, pc :
        propagateChange )
621     ac : AlphaCard( id == acid )
622     eval(o instanceof Participant)
623     eval(acid.getCardID().equals("$cra") && o instanceof
        Participant)
624     then
625     #update participant
626     for (Participant participant : craPayload.getLoParticipants())
        {
627         if(participant.getSubject().equals(((Participant)o).
            getSubject())) {
628             System.out.println( "RULE_NAME:\\"CRAPayload:\Add_or_
                update_a_participant-UPDATE\\"");
629             craPayload.getLoParticipants().remove(participant);
630             craPayload.getLoParticipants().add((Participant)o);
631
632             #new version
633             modify(ac) { setVersion(String.valueOf((Double.valueOf
                (ac.getVersion()).doubleValue() + 1)))}
634
635             if(pc == true) {
636                 propagatePayloadChange(cpe, craPayload,
                    updateServiceSender);
637             }
638
639             vvs.setSptype("xml");
640             vvs.setVersion(ac.getVersion());
641             vvs.putPayload(acid, craPayload);
642         }
643     }
644
645     # add another participant
646     if(!craPayload.getLoParticipants().contains((Participant)o)) {
647         System.out.println( "RULE_NAME:\\"CRAPayload:\Add_or_
            update_a_participant-ADD\\"");
648         craPayload.getLoParticipants().add((Participant)o);
649
650         #new version

```

```

651         modify(ac) { setVersion(String.valueOf((Double.valueOf(ac.
        getVersion()).doubleValue() + 1)))}
652
653         if(pc == true) {
654             propagatePayloadChange(cpe, craPayload,
        updateServiceSender);
655         }
656
657         vvs.setSptype("xml");
658         vvs.setVersion(ac.getVersion());
659         vvs.putPayload(acid, craPayload);
660     }
661
662     retract(cpe);
663 end
664
665 rule "Add_payload_to_a_content_alphaCard"
666     when
667         vvs : VerVarStoreImpl()
668         craPayload : CRAPayload()
669         cpe : ChangePayloadEvent( acid : alphaCardID, o : obj, pc :
        propagateChange )
670         ac : AlphaCard( id == acid )
671         eval(o instanceof Payload)
672     then
673         vvs.setSptype(ac.getSPType());
674
675         if(ac.isVersioning()) {
676             #new version
677             modify(ac) { setVersion(String.valueOf((Double.valueOf(ac.
        getVersion()).doubleValue() + 1)))}
678             vvs.setVersion(ac.getVersion());
679         } else {
680             vvs.setVersion("0");
681         }
682
683         vvs.putPayload(acid, (Payload) o);
684
685         if(pc == true) {

```

```
686         propagatePayloadChange(cpe, craPayload,
687                                 updateServiceSender);
688     }
689     retract(cpe);
690 end
```

Listing A.1: The Rule Package

Bibliography

- [ACJZ08] J. Adamczyk, R. Chojnacki, M. Jarzab, and K. Zieliński. Rule Engine Based Lightweight Framework for Adaptive and Autonomic Computing. *Computational Science–ICCS 2008*, pages 355–364, 2008.
- [CWW⁺06] B. Chaudhry, J. Wang, S. Wu, M. Maglione, W. Mojica, E. Roth, S.C. Morton, and P.G. Shekelle. Systematic review: impact of health information technology on quality, efficiency, and costs of medical care. *Annals of internal medicine*, 144(10):742, 2006.
- [DBM88] U. Dayal, A. Buchmann, and D. McCarthy. Rules are objects too: a knowledge model for an active, object-oriented database system. *Advances in Object-Oriented Database Systems*, pages 129–143, 1988.
- [DEH⁺00] P. Dourish, W. K. Edwards, J. Howell, A. LaMarca, J. Lamping, K. Petersen, M. Salisbury, D. Terry, and J. Thornton. A programming model for active documents. In *Proc of the 13th annual ACM symposium on User interface software and technology*, pages 41–50. ACM New York, NY, USA, 2000.
- [DKM86] Klaus R. Dittrich, Angelika M. Kotz, and Jutta A. Mülle. An event/trigger mechanism to enforce complex consistency constraints in design databases. *SIGMOD Rec.*, 15(3):22–36, 1986.
- [Dro10] JBoss Drools Documentation. <http://www.jboss.org/drools/documentation.html>, 2010.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading, MA, 1995.
- [HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. *SIGPLAN Not.*, 37(11):161–173, 2002.

- [HM00] E. Heinrich and H.A. Maurer. Active documents: Concept, implementation and applications. *Journal of Universal Computer Science*, 6(12):1197–1202, 2000.
- [HS09] T. Heimrich and G. Specht. Enhancing ECA Rules for Distributed Active Database Systems. *Web, Web-Services, and Database Systems*, pages 199–205, 2009.
- [Jac98] Peter Jackson. *Introduction to Expert Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.
- [JSR02] Java Community Process. JSR 94 Java Rule Engine API. <http://www.jcp.org/en/jsr/detail?id=94>, 2002.
- [KL98] A. Koschel and P.C. Lockemann. Distributed events in active database systems: Letting the genie out of the bottle. *Data & Knowledge Engineering*, 25(1-2):11–28, 1998.
- [Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
- [LED⁺99] A. LaMarca, W. K. Edwards, P. Dourish, J. Lamping, I. Smith, and J. Thornton. Taking the work out of workflow: mechanisms for document-centered collaboration. In *Proc of the 6th conference on European Conference on Computer Supported Cooperative Work*, pages 1–20. Kluwer Academic Publishers Norwell, USA, 1999.
- [LR07] R. Lenz and M. Reichert. IT support for healthcare processes-premises, challenges, perspectives. *Data & Knowledge Engineering*, 61(1):39–58, 2007.
- [MD89] Dennis McCarthy and Umeshwar Dayal. The architecture of an active database management system. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 215–224, New York, NY, USA, 1989. ACM.
- [NL09] Christoph P. Neumann and Richard Lenz. alpha-Flow: A Document-based Approach to Inter-Institutional Process Support in Healthcare. In *Proc of the*

3rd Int'l Workshop on Process-oriented Information Systems in Healthcare (ProHealth '09) in conjunction with the 7th Int'l Conf on Business Process Management (BPM'09), Ulm, Germany, September 2009.

- [NL10] Christoph P. Neumann and Richard Lenz. The alpha-Flow Use-Case of Breast Cancer Treatment – Modeling Inter-Institutional Healthcare Workflows by Active Documents. In *Proc of the 8th Int'l Workshop on Agent-based Computing for Enterprise Collaboration (ACEC) at the 19th Int'l Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE 2010)*, Larissa, Greece, June 2010.
- [WKJ⁺01] P. Werle, F. Kilander, M. Jonsson, P. Lönnqvist, and C. Jansson. A ubiquitous service environment with active documents for teamwork support. In *UbiComp 2001: Ubiquitous Computing*, pages 139–155. Springer, 2001.

