Data Provenance and Destiny in Distributed Environments

Christoph Ringelstein

Vom Promotionsausschuss des Fachbereichs 4: Informatik
der Universität Koblenz-Landau
zur Verleihung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation.

Einreichung der Dissertation:	29.07.2011
Datum der wissenschaftlichen Aussprache:	22.11.2011

Vorsitzender des Promotionsausschusses:	Prof. Dr. Rüdiger Grimm
Vorsitzender der Promotionskommission:	Prof. Dr. Dieter Zöbel
Berichterstatter:	Prof. Dr. Steffen Staab
	Prof. Dr. Rüdiger Grimm
	Prof. Dr. Mathias Weske

“The biggest vulnerability is ignorance.”
- Sun Tzu in “The Art of War”

Abstract

Modern Internet and Intranet techniques, such as Web services and virtualization, facilitate the distributed processing of data providing improved flexibility. The gain in flexibility also incurs disadvantages. Integrated workflows forward and distribute data between departments and across organizations. The data may be affected by privacy laws, contracts, or intellectual property rights. Under such circumstances of flexible cooperations between organizations, accounting for the processing of data and restricting actions performed on the data may be legally and contractually required. In the Internet and Intranet, monitoring mechanisms provide means for observing and auditing the processing of data, while policy languages constitute a mechanism for specifying restrictions and obligations.

In this thesis, we present our contributions to these fields by providing improvements for auditing and restricting the data processing in distributed environments. We define formal qualities of auditing methods used in distributed environments. Based on these qualities, we provide a novel monitoring solution supporting a data-centric view on the distributed data processing. We present a solution for provenance-aware policies and a formal specification of obligations offering a procedure to decide whether obligatory processing steps can be met in the future.

Acknowledgments

I want to thank Steffen Staab for his extended support provided whenever needed. I would like to thank my colleges from the Institute of Web Science and Technologies at the University of Koblenz-Landau for discussions and their constructive feedback. Most of all, I indebted to Christoph Adolphs, Renata Dividino, Klaas Dellschaft, Thomas Gottron, Gerd Gröner, Jérôme Kunegis, Julia Preusse, Antje Schultz and Felix Schwagereit for giving me feedback on earlier and later states of my research.

Special thanks to Rüdiger Grimm and Daniel Pähler from the research group for IT-Risk Management at the University of Koblenz-Landau and Sebastian Meissner, Martin Rost, and Jan Schallaböck from the Independent Centre for Privacy Protection Schleswig-Holstein for discussing various privacy related issues leading to the invention of Sticky Logging and successively to DiALog. The basic idea of Papel has been motivated by discussions with Prof. Marianne Winslett and her working group. I thank her for the nice time at the University of Illinois in Urbana-Champaign.

The prototype implementation of sticky logging is based on a student project by Martin Schnorr, and the LogAnalyzer has been implemented in cooperation with my working student Roland Naglo. Both worked very hard. I want to thank Paul Schmücker who proofread the complete thesis for spelling and grammar mistakes.

Finally, I am grateful to my family, especially my wife and parents, not only for believing in my dreams and for respecting my commitment to my research, but also for their long lasting support.

Contents

1 Managing the Distributed Processing of Data

"Trust is good, control is better."
- Wladimir Iljitsch Uljanow

The Internet facilitates the distributed processing of data, e.g. the communication of data via a Web site or e-mail. In the same manner, the Internet enables organizations to offer business capabilities as independent Web applications and Web services. As standardized interfaces are used to communicate, a loose coupling is supported. Loose coupling eases the integration of external services into internal workflows as well as the services provisioning to external consumers. The flexibility thereof facilitates the combination of services from different organizations into one comprehensive, integrated workflow leading to an agile virtual organization able to adapt more quickly to new organizational requirements and business needs.

The gained flexibility also displays disadvantages. An integrated workflow forwards and distributes data between departments and across organizational boundaries. Technologies, e.g. Software as a Service or Hardware as a Service, support the further distribution of data. These data may be affected by privacy laws, contracts, or intellectual property rights. Thus, the distribution raises privacy and data protection issues. Under such circumstances of flexible cooperations, controlling actions and accounting for actions may be legally and contractually required.

From a legal perspective, we can control the processing by means of contracts and laws, such as service level agreements and privacy laws. Technically, we can implement the legal foundations by policies and audits. The policies define rules for the current and future processing. The adherence to policies can be controlled at run-time and allows for verification of the planned processing before it is performed. The audit can be effected after the processing.

A detailed overview about the processing is required to enable the audit. The needed details of the overview depend on contracts or laws. The overview consists of a model of the workflow expressing *who* was involved in the processing of the data as well as *why* and *how* the processing has been performed. Even for internal workflows, a predefined model is most frequently lacking. In this case, the model can only be created afterwards. The creation of the model requires information about the processing history. This information is called provenance. Merriam-Webster defines provenance in [Merriam-Webster, 2004] as:

1 : origin, source
2 : the history of ownership of a valued object or work of art or literature

The distributed processing of data requires an adjusted interpretation of provenance, cf. [Sizov, 2007]:

Provenance is a formal model of the actual processing of a certain piece of data.

In a distributed environment, provenance information can only be generated if the utilized observation mechanism facilitates the generation of a detailed overview of data processing. Hence, we require a model of the data processing in distributed environments along with a distributed mechanism for logging that collects the needed information. Both model and logging mechanism have to adhere to the confidentiality of service and data.

An addition mechanism is required for run-time restriction of the processing. Policies stipulate how data is processed. Merriam-Webster defines policies in [Merriam-Webster, 2004] as:

2 a : a definite course or method of action selected from among alternatives and in light of given conditions to guide and determine present and future decisions
b : a high-level overall plan embracing the general goals and acceptable procedures especially of a governmental body

A similar, but more technical definition of policies is given in [Hinton and Lee, 1994]:

Policies are statements of the goals for the behavior of a system.

Conditions of policies can depend on information about the environment, e.g. the receiver of a data transfer. They also can depend on information about the data itself, such as the subject of a health record. To provide this information, we can use provenance. The provenance contains information about data properties and about the data flow, e.g. has a health record been de-identified[1] before it has been transferred. Those statements refer to the temporal structure of processing histories. As the temporal structure may be very complex, a simplification is required that allows to formalize provenance-aware policies in an institutive manner.

Contracts and laws may demand a certain reaction in the future, such as that each health record must be deleted after a certain time. Therefore, some policy languages foresee the possibility to specify obligations for future processing. Obligations can conflict with each other, e.g. the obligation to keep a log of activities conflicts with the obligation to delete private data after a given time interval. The current processing step may also conflict with obligations. A deleted record can not be required in the future. To prevent

[1] http://www.ucdmc.ucdavis.edu/compliance/guidance/privacy/deident.html

obligations from becoming unfulfillable, one has to verify the existence of at least one future execution that fulfills all obligations and starts with the current processing step. We call such a future execution the *destiny* of the data processing. The challenge is that in dynamic environments the future execution is specified by policy rules. Thus, a formal method is required to decide whether a destiny exists for a planned processing step based on the given policy rules and the history.

1.1 Thesis

So far, we have discussed the motivation of managing the distributed processing of data. During the discussion we have identified various problems that hamper the management. The following list summarizes these problems and depicts our hypotheses:

1. To audit the data processing, we need a global model of the processing. In distributed environments, such a model is most often lacking. Our hypothesis is:

Hypothesis 1:

The distributed processing of data can be audited even if a global model of the processing is lacking a-priori. The auditing can be achieved by a complete and sound reconstruction of the process execution.

---

2. To be able to audit the processing, we need to monitor the distributed processing in an exhaustive manner. Applied monitoring mechanisms have to be able to collect sound provenance information of the complete processing. The monitoring is hampered by the distribution, by

the autonomy of the involved organizations, and the lacking standardization. In spite of these obstacles, the provenance information must allow for a complete and sound reconstruction of the processing to be used for auditing. Our hypothesis is:

Hypothesis 2:

A distributed, data-centric logging mechanisms can collect provenance information and generate a reconstructed model that is complete as well as sound regarding the global model and can be used for auditing.

3. To restrict the processing at runtime, we need a provenance-aware policy mechanism. However, policy languages do not support the specification of provenance based policy conditions in an intuitive manner because of the temporal structure of processing histories. Our hypothesis is:

Hypothesis 3:

Policy conditions can be based on data processing histories by mapping the temporal structure of the histories to a graph structure.

4. A planned processing step will not render obligations unfulfillable if a destiny exists. To verify the existence of a destiny, we need a decision procedure based on the given set of policy rules and history. Policy languages do neither provide such a decision procedure nor a reduction of the decision problem to another well-defined decision problem. Such a procedure respectively reduction is required to decide the existence of a destiny. Our hypothesis is:

Hypothesis 4:

The decision problem of the existence of a destiny considering a given set of policy rules and the history can be reduced to the decision problem of the reachability of transitions in colored Petri nets.

1.2 General Approach

Before starting with our hypotheses, we consider different kinds of fundamentals in Chapter 2. These are basic concepts related to the distributed processing of data in the Web and means for their technical realization, such as service-oriented architectures and Web services. With colored Petri nets and Datalog we also require certain formal models and methods. As we base parts of our work on the eXtensible Access Control Markup Language and the Open Provenance Model, we give an introduction of these. After discussing these fundamentals, we specify the terminology we use throughout this thesis.

In Chapter 3, we derive requirements to tackle the hypotheses. To this end, we introduce a health care scenario. The scenario serves as a run-through example. The legal and contractual aspects of processing data in distributed environments are discussed, before we analyze organizational and technical issues that arise from auditing and restricting the distributed data processing. Based on the scenario and discussion, we derive the requirements to solve the organizational and technical issues. Given these requirements we validate our hypotheses.

We validate Hypothesis 1 by providing a methodical description to audit the data processing in distributed environments. In Chapter 4, we introduce our approach which we call **DiALog**: **Di**stributed **A**uditing **Log**s. DiALog can specify the *who*, *why*, and *how* of the distributed data processing. It

supports a data-centric view to model the processing of one specific data item with all its instances. The processing can be across organizations and independent of business processes, e.g. data which is stored and later reused. To enable data owners to manage the distributed processing of their data, we identify different models of the execution and define formal qualities. These qualities are the soundness and completeness of the reconstruction. If a reconstructed model fulfills these qualities, it can be used for an audit.

To validate Hypothesis 2, we introduce a mechanism to collect and provide the provenance information of the distributed data processing in Chapter 5. We call this designated mechanism **sticky logging**. Sticky logging is a generic middleware for distributed logging and is tailored to observe the processing of single data items and its instances. It attaches the logs directly to the processed data instances as metadata. Furthermore, sticky logging allows for reconstructing of how data are processed by whom and why. The reconstructed model is specified in DiALog. Sticky logging consists of two parts. These are structures specifying the organization of the collected provenance information as well as an architecture defining operations how to collect and share the information about the processing. We demonstrate the feasibility of sticky logging by a prototypical implementation. To prove the functionality of the prototype we implement a business case.

Hypothesis 3 is validated by providing the **P**rovenance **A**ware **P**olicy definition and **E**xecution **L**anguage, short **Papel** in Chapter 6. Defining Papel, we focus on policies containing conditions based on processing histories. We use the sticky logging mechanism to provide the needed provenance. Based on this information, policy conditions can relate to provenance information and to the temporal structures of processing histories. We achieve this by mapping the temporal structure to a graph structure. Besides Papel's syntax, we define its semantics via an interpretation function. Its feasibility is shown by implementing Papel using Datalog. We address privacy and data protection issues that newly emerge from connecting policies with provenance.

In Chapter 7, we validate Hypothesis 4 by **Care**. Care is an extension of Papel and defines syntax and semantics to specify future obligations in pol-

icy rules. To check whether all future obligations can be met, the existence of a destiny must be shown based on the given policy rules and processing history. As for policy rules a procedure to decide the existence of a destiny is not defined, we reduce the problem to the decision problem of the reachability of transitions in colored Petri nets [Jensen, 1992] instead. To reduce the problem, we provide a translation from Care policy rules to colored Petri nets. In colored Petri nets we can decide the reachability and thus decide whether there is a destiny and whether all obligations can be met in the future.

After validating our hypotheses, we conclude by discussing our research contributions and by giving an outlook on future work in Chapter 8.

1.3 Publications and Exploitation

This work is based on our former publications. General assumptions about the management of data processing in distributed environments have been published at the *5th International Workshop for Technical, Economic and Legal Aspects of Business Models for Virtual Goods* [Ringelstein et al., 2007b] and in the national magazine *Datenschutz und Datensicherheit* [Ringelstein et al., 2007a]. The sticky logging mechanism was published at the *Workshop on Privacy Enforcement and Accountability with Semantics* [Ringelstein and Staab, 2007] and in a second article published in *Datenschutz und Datensicherheit* [Ringelstein, 2007]. In [Ringelstein and Staab, 2009] published at the *IEEE 7th International Conference on Web Services*, DiALog is introduced. And it is further discussed in the *International Journal of Web Service Research* [Ringelstein and Staab, 2010a]. The definition of the syntax and formal semantics of Papel were published in the *8th International Conference on Business Process Management* [Ringelstein and Staab, 2010b]. Finally, a more general discussion of Papel has been published in the *IEEE Internet Computing* [Ringelstein and Staab, 2011].

Results of our work have been exploited in various research projects. We

contributed our research in the field of service-oriented architectures and Web services to the European research project "Adaptive Services Grid". In this project, our results help Web service providers (telecommunication companies) to improve the automated choreography of Web services. Our findings regarding sticky logging have been exploited in the analysis "Technique Analysis and Risk Management for Service-oriented Architectures in Virtual Organizations" founded by the Federal Ministry for Education and Research. With sticky logging we provide an improved solution for virtual organizations to meet privacy laws. Virtual organizations can use sticky logging as a means to respond to information requests by persons concerned. In the European project "Knowledge Sharing and Reuse across Media", we have exploited sticky logging and DiALog. Partners in industry use sticky logging to monitor the distributed communication of an ad-hoc created task force. They are able to improve the creation of task forces considering the monitored information. Our results are exploited in "Where eGovernment meets the eSociety". In this project, policy makers analyze comments in social networking sites to determine people's opinions about political topics. Our findings regarding DiALog help policy makers to inform people about the data analysis. The people can use Papel to restrict the usage of their data.

2 Fundamentals of the Distributed Processing of Data

We have to consider different kinds of fundamentals and related work. Understanding the basic concepts related to the distributed processing of data in the Web and means for their technical realization is essential. We discuss the related concepts of *service-oriented architectures* and *Web services* in Section 2.1. Specific models and methods are required to specify our approach and our decision procedure. These are *colored Petri nets* as well as *Datalog*, which we summarize in Section 2.2. Papel is based on existing solutions for policies and provenance. These are the *eXtensible Access Control Markup Language (XACML)* and the *Open Provenance Model (OPM)*, which we introduce in Section 2.3. The terminology used throughout this work is specified in Section 2.4. We discuss the work related to our approaches in the related work sections of the Chapters 4, 5, 6 and 7.

2.1 Distributed Environments

The Internet facilitates the distributed processing of data. Web sites, Web applications and Web services are means for implementing distributed environments in the Internet. In addition e-mail and other communication means process data in a distributed manner. Apart from the Internet, the intranets of organizations constitute distributed environments, as well.

A paradigm that specifies design principles for the distributed data processing in the Internet and intranet are service-oriented architectures. We

also introduce Web services as they are used to implement the service-oriented architecture of our scenario.

2.1.1 Service-oriented Architectures

Service-oriented architecture is a software design paradigm. In a service-oriented architecture functionality is provided as service, which can be consumed and orchestrated to more complex services. The design principle of a service-oriented architecture builds upon standards, e.g. Web services [Booth et al., 2004] and SOAP [Gudgin et al., 2007]. Adhering to these standards allows for an easy integration and loose coupling of services. Loose coupling describes the ability to use services only if required (*event driven*) and to easily exchange services. Through loose coupling organizations can easily provide business capabilities as services as well as integrate external service into internal processes leading to virtual organizations [Davidow and Malone, 1992].

Essential for service-oriented architectures is the specification of interfaces using standardized formats to find and use services. The standardized interfaces not only allow for combining services to more complex services (service orchestration), but also enable the decomposition of complex business services into flexibly interchangeable, modular components. One can easily monitor service calls through such standardized interfaces.

Service-oriented architectures are often used together with Web services as those meet the design principals of a service-oriented architecture. The standardization of the Web service interfaces by the World Wide Web Consortium (W3C) frames a technical implementation of the service-oriented paradigm and various standards exists to build a service-oriented architecture, e.g. SOAP, REST (Representational State Transfer) [Fielding, 2000], and RPC (Remote Procedure Call) [White, 1976], etc. Service-oriented architectures are supported through Web services platforms of various pro-

ducers, such as Software AG with WebMethods[1], Microsoft with .NET[2], IBM with WebSphere[3], BEA with WebLogic[4], and Red Hat Middleware with JBoss[5].

2.1.2 Web Services: An Example of SOA

Web services are software components that are accessible as Web resources in order to be reused by other Web services or software. We can use Web services to constitute a service-oriented architecture. The Web service architecture standard [Booth et al., 2004] has three major parts:

1. *A standardized communication interface:* For communication with and between Web services the standards SOAP or XML-RPC (Extensible Markup Language Remote Procedure Call) are used.

2. *A standardized service description:* The service interfaces and the services are described by means of WSDL (Web Service Description Language). Each Web service provides its WSDL description allowing for an easy integration of the service.

3. *A standardized directory service:* To automatically find and integrate Web services, one can use an UDDI (Universal Description, Discovery and Integration), a yellow page for Web services. However, even if the Web service standard has quite some spreading, UDDIs could not be established for open usage.

Web services function as middleware connecting different parties such as companies or organizations distributed over the Web.

[1] `http://www.softwareag.com/corporate/products/wm/default.asp`, retrieved: Nov. 29th, 2010
[2] `http://www.microsoft.com/net/`, retrieved: Nov. 29th, 2010
[3] `http://www-01.ibm.com/software/websphere/`, retrieved: Nov. 29th, 2010
[4] `http://webservice.bea.com/`, retrieved: Nov. 29th, 2010
[5] `http://www.jboss.com/`, retrieved: Nov. 29th, 2010

2.2 Formal Models and Methods

We require certain formal models and methods. Whenever possible, we intend to use existing means. We did so for the modeling of the distributed processing of data and for the specification and implementation of our policy language.

During its distributed processing, the data reaches different states as each processing step causes state changes. A standard means to model state changes are *state-transition nets*. Hence, the modeling of the distributed data processing can be achieved by a state-transition net. A detailed description as to how the distributed data processing is modeled by state-transition nets is given in Chapter 4. For the formal grounding of DiALog, we use colored Petri nets [Jensen, 1992] a formalism for state-transition nets that is based upon Petri nets [Petri, 1962]. Colored Petri nets are also used in Care. We reduce the problem of deciding whether a destiny exists to the decision problem of reachability in colored Petri nets.

Policy languages describe rules whether a data item may be processed or not. The interpretation of a policy language is specified by its execution semantics. We define an interpretation function to specify the execution semantics of Papel. Based on the interpretation function, we have chosen Datalog [Ceri et al., 1989] to provide a general implementation with a formal grounding.

2.2.1 Petri Nets

Colored Petri nets are an extension to Petri nets [Petri, 1962]. We introduce them, before we explain colored Petri nets in the next section. This short introduction is based upon the work of Peterson [Peterson, 1981].

Petri nets are a model to describe systems and their processes. Through the formal grounding of Petri nets, one can use Petri nets to formally analyze the modeled processes. Petri nets are bipartite, directed graphs with two kinds of nodes: places P and transitions T. Places and transitions are connected

by the input function I and the output function O of the transitions. I maps one transition to a set of input places and O maps one transition to a set of output places. These connections are modeled as arcs. An example Petri net is depicted in Figure 2.1.

Definition 2.1: *Structure of Petri Nets [Peterson, 1981]*

Let P^{∞} specify the set of all bags of places $p \in P$, a Petri net structure is defined as a quadruple (P, T, I, O), where:

- P is a finite set of places,
- T is a finite set of transitions, disjoint from the set of places: $P \bigcap T = \emptyset$.
- I is the input function: $I : T \rightarrow P^{\infty}$, and
- O is the output function: $O : T \rightarrow P^{\infty}$.

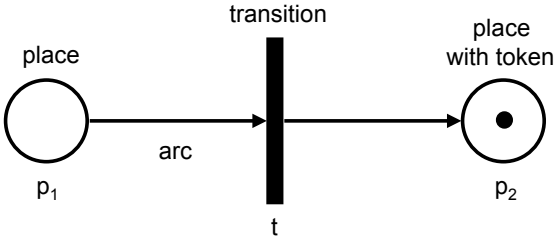

Figure 2.1: A Small Petri Net.

In addition to places and transitions, Petri nets introduce a third primitive concept called tokens. Tokens are assigned to places and modeled as dots.

The distribution of tokens to places is described by the marking μ. Any number of tokens in $\mathbb{N}_0$ may be assigned to each place.

Definition 2.2: *Marking [Peterson, 1981]*

Let (P, T, I, O) be a Petri net, and be $\mathbb{N}_0$ the nonnegative integers, a marking μ is defined as $\mu : P \to \mathbb{N}_0$.

The marking changes during the execution of a Petri net.

Definition 2.3: *Execution [Peterson, 1981]*

Let (P, T, I, O) be a Petri net with the marking μ, and let $\#(p, I(t))$ and $\#(p, O(t))$ are functions returning the number of occurrences of the place p in the input respectively output bag of the transition t, a transition $t \in T$ is *enabled* if:

$$\forall p \in P : \mu(p) \geq \#(p, I(t)).$$

A transition $t \in T$ may *fire* whenever it is enabled. Firing a transition t results in a new marking μ' defined by:

$$\mu'(p) = \mu(p) - \#(p, I(t)) + \#(p, O(t)).$$

Figure 2.1 depicts a small Petri net consisting of two places and a transition. The first place is connected to the transition by an arc and is the input place of the transition. The transition again is connected to the second place, the output place of the transition. The formal description of the depicted Petri net is given in Listing 2.1.

Listing 2.1: *Example Structure of a Petri Net [Peterson, 1981]*

$C = (P, T, I, O)$
$P = \{p_1, p_2\}$

$T = \{t\}$
$I(t) = \{p_1\}$
$O(t) = \{p_2\}$
$\mu(p_1) = 0$
$\mu(p_2) = 1$

2.2.2 Colored Petri Nets

Based on Petri nets, Jensen has defined colored Petri nets. Colored Petri nets have the same expressiveness as Petri nets but a syntax allowing a much easier specification of the nets. The following short summary of colored Petri nets is based upon the work of Jensen [Jensen, 1992].

Definition 2.4: *Structure of Colored Petri Nets [Jensen, 1992]*

A non-hierarchical colored Petri net is defined as a nonuple $CPN = (\Sigma, P, T, A, N, C, G, E, I)$, where:

- Σ is a finite set of non-empty types, called color sets,
- P is a finite set of places,
- T is a finite set of transitions, disjoint from the set of places: $P \cap T = \emptyset$,
- A is a finite set of arcs, disjoint from the set of places and transitions: $P \cap A = T \cap A = \emptyset$,
- N is a node function $N : A \rightarrow P \times T \cup T \times P$,
- C is a color function $C : P \rightarrow \Sigma$,
- G is a guard function $G : T \rightarrow expressions$ such that $\forall t \in T : [\mathit{Type}(G(t)) = \mathbb{B} \wedge \mathit{Type}(\mathit{Var}(G(t))) \subseteq \Sigma]$,

- E is an arc expression function $E : A \rightarrow expressions$ such that $\forall a \in A : [\mathit{Type}(E(a)) = C(p(a))_{MS} \wedge \mathit{Type}(\mathit{Var}(E(a))) \subseteq \Sigma]$, where $p(a)$ is the place of $N(a)$, and
- I is an initialization function $I : P \rightarrow closed\ expressions$ such that $\forall p \in P : [\mathit{Type}(I(p)) = C(p)_{MS}]$.

The Σ specifies the types, operations and functions to be used in the net inscriptions (e.g. arc expressions, guards, color sets). The input and output function of Petri nets are replaced by the node function that maps each arc to a pair of source nodes and destination nodes. In contrast to Petri nets, colored Petri nets allow multiple arcs between the same ordered pair of nodes. Each place p is mapped by the color function C to a color set $C(p)$. $\mathbb{B}$ is the boolean type consisting of the elements $\{true, false\}$. In DiALog, we make no use of guard functions. As specified by the arc expression function E that maps each arc a into an expression of the type $C(p(a))_{MS}$ each evaluation of an arc expression has to result in a multi-set (indicated by $_{MS}$) over the color sets of the corresponding place. *Type*(v) is the type of the variable v and *Type*$(expr)$ is the type of the expression. The definition of colored Petri nets does neither define the syntax nor semantics of expressions. It assumes both as given. Closed expressions are expressions without variables. *Var*$(expr)$ is the set of variables in an expression $expr$.

To specify the behavior of colored Petri nets, we are required to define bindings of transitions, tokens, markings as well as the enabling of steps and their occurrence. A binding of a transition t is a function that replaces each variable of t with a color of the correct type in a way that the guard function evaluates to true.

Definition 2.5: *Binding [Jensen, 1992]*

Let $\forall t \in T : \mathit{Var}(t) = \{v | v \in \mathit{Var}(G(t)) \vee \exists a \in A(t) : v \in Var(E(a))\}$ be, the binding function b of a transition t is defined on *Var*(t), such that:

(i) $\forall v \in \mathit{Var}(t) : b(v) \in \mathit{Type}(v)$.
(ii) $G(t) < b >$.

$A(t)$ is the function that returns the set of surrounding arcs of a transition t. $B(t)$ specifies the set of all bindings for t and a binding is indicated by $<$ and $>$. The state of a colored Petri net is described by the distribution of token elements called marking. The possible state changes are derived from the binding elements, and changes are described by steps.

Definition 2.6: *Token Element, Binding Element, Marking and Step [Jensen, 1992]*

A *token element* is defined as a pair (p, c) where $p \in P$ and $c \in C(p)$. A *binding element* is defined as a pair (t, b) where $t \in T$ and $b \in B(t)$. A *marking* is defined as a multi-set over the set of all tokens elements ***TE***. A *step* is defined as a non-empty and finite multi-set over the set of all binding elements ***BE***. The initial marking M_0 is defined as the marking obtained by evaluating the initialization expression:

$$\forall (p, c) \in \mathbf{TE} : M_0(p, c) = (I(p))(c).$$

A step in a colored Petri net may occur if it is enabled.

Definition 2.7: *Enabling and Occurring of Steps [Jensen, 1992]*

Let $E(x_1, x_2)$ be defined as: $\forall (x_1, x_2) \in (P \times T \cup T \times P) : E(x_1, x_2) = \sum_{a \in A(x_1, x_2)} E(a)$. A step Y is enabled in a marking M iff:

$$\forall p \in P : \sum_{(t,b) \in Y} E(p, t) < b > \leq M(p).$$

A step Y that is enabled in M_1 may occur, changing from the marking M_1 to another marking M_2

$$\forall p \in P : M_2(p) = (M_1(p) - \sum_{(t,b)\in Y} E(p,t) < b >) + \sum_{(t,b)\in Y} E(t,p) < b >).$$

$A(x_1, x_2)$ is the function that returns the set of arcs connecting a pair of specified nodes. The first sum $\sum_{(t,b)\in Y} E(p,t) < b >$ specifies the tokens which are removed. The second sum $\sum_{(t,b)\in Y} E(t,p) < b >$ specifies the tokens that are added.

Figure 2.2 depicts a small colored Petri net consisting of two places and a transition. The first place is connected to the transition by an arc and is the input place of the transition. The transition again is connected to the second place, the output place of the transition. Both transitions have an arc expression: "n" and "$n + 1$". The second place holds a token of the integer type with the value 2. The formal description of the depicted colored Petri net is given in Listing 2.2. The colored Petri net models the increment of a natural number by one. As a token with a value of 2 is already in the second place, the transaction has already occurred and the input token must have had the value 1.

Listing 2.2: *Example Structure of the Colored Petri Net in Figure 2.2*

$\Sigma = \{\mathbb{N}\}$.
$P = \{p_1, p_2\}$.
$T = \{t\}$.
$A = \{p_1\_to\_t, t\_to\_p_2\}$
$N(a) = \begin{cases} (p_1, t) & \text{if } a = p_1\_to\_t \\ (t, p_2) & \text{if } a = t\_to\_p_2 \end{cases}$
$C(p) = \mathbb{N}$ for all p
$G(t) = true$ for all t

$$E(a) = \begin{cases} \text{n} & \text{if } a = p_1\_to\_t \\ \text{n+1} & \text{if } a = t\_to\_p_2 \end{cases}$$

$$I(p) = \begin{cases} \{1\} & \text{if } p = p_1 \\ \{\} & \text{if } p = p_2 \end{cases}$$

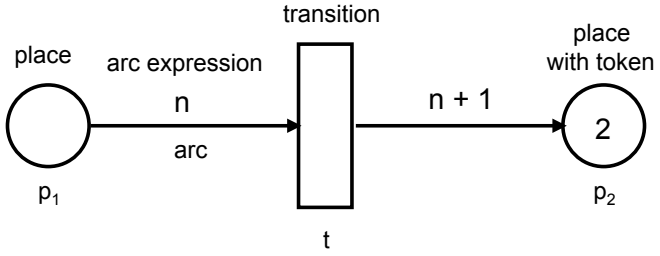

Figure 2.2: A Small Colored Petri Net.

2.2.3 Datalog

Datalog is a database query and rule language that is designed to query deductive databases. Datalog is based on the logic programming paradigm and a subset of the general purpose logical programming language Prolog [Colmerauer and Roussel, 1993]. The following short introduction summarizes the explanation of Datalog in [Ceri et al., 1989].

Datalog is a simplification of normal logic programs (for a detailed introduction into logic programming see [Lloyd, 1993]). A Datalog program represents knowledge as rules and facts. An example fact is that *Jane Doe is a patient*. By means of rules, facts can be derived from other facts, e.g. *if* X *is a patient and* X *is pregnant,* X *is a female*; where *X* is a variable.

Definition 2.8: *Syntax of Datalog [Ceri et al., 1989]*

In Datalog, rules and facts are defined as Horn clauses:

$$L_0 : -L_1, .., L_n$$

where each L_i is a literal and each literal has the form $p_i(t_1, .., t_n)$, where p_i is a predicate symbol and each t_j is a term specifying either a variable or a constant.

The clauses can be divided into the left hand side and right hand side. The left hand side is called the *head* and the right hand side is called the *body* of the clause. Facts are clauses with a empty body and rules have at least one literal in their body. Listing 2.3 depicts the facts '*Jane Doe is a patient*' (first line) and '*Jane Doe is pregnant*' (second line) as well as the rule '*if* `X` *is a patient and* `X` *is pregnant,* `X` *is a female*' (third line) as a Datalog program.

Listing 2.3: *Datalog Rules and Facts*

```
patient(JaneDoe).
pregnant(JaneDoe).
female(X) :- patient(X), pregnant(X).
```

Datalog programs have certain qualities. These guarantee that the derivable set of facts is finite. To this end, all facts of a Datalog program must be *ground* (see Definition 2.9), and each variable appearing in the head of a rule must also appear in the body of the rule.

Definition 2.9: *Ground Facts [Ceri et al., 1989]*

A fact that does not contain any variable is *ground*.

For instance, the facts in Listing 2.3 are ground.

We can consider two sets of clauses the set of ground facts called the extensional database (DB_E) and the Datalog program called the intensional database (DB_I). The semantics of Datalog are defined by a mapping (see Definition 2.10) that is based on the following basics. Each Datalog fact F represents an atomic formula $F*$ of first-order logic [Smullyan, 1968] and each rule R a formula $R*$ of the form $\forall X_1, .., \forall X_m(L_1 \wedge .. \wedge L_n \Rightarrow L_0)$, where X_1 to X_m represent the variables occurring in R. If S is a finite set of Datalog clauses, be $cons(S*)$ the set of all facts that are logical consequences of S.

Definition 2.10: *Semantics of Datalog [Ceri et al., 1989]*

Be H the Herbrand base of the Datalog language consisting of the set of all facts one can express with Datalog, be H_E the extensional part of the Herbrand base that contains all literals of the Herbrand base H whose predicate is a DB_E-predicate, and be H_I the set of all literals in H whose predicate is a DB_I-predicate, the semantics of a Datalog program are defined by a mapping $\mathbb{M}$ from H_E to H_I. The mapping $\mathbb{M}$ assigns to each possible extensional database $E \subseteq H_E$ the set $\mathbb{M}(E)$ of intensional result facts defined by $\mathbb{M}(E) = cons(P \cup E) \cap H_I$.

2.3 Fundamental Approaches

We based our work on different existing solutions regarding the general modeling of provenance information as well as the underlying policy language. With respect to model provenance information, we build on the Open Provenance Model (OPM) [Moreau et al., 2008]. In Section 2.3.1, we summarize the basic idea of OPM. Regarding policy language, we based Papel on the eXtensible Access Control Markup Language (XACML) [Moses et al., 2005] standard, which we shortly introduce in Section 2.3.2.

2.3.1 Open Provenance Model

The *open provenance model* aims to provide a standard for modeling provenance [Moreau et al., 2010]. The main constitutions of the open provenance model are nodes, dependencies and roles. Nodes are used to model artifacts, processes, and agents. Artifacts model subjects of state changes. The open provenance model defines artifacts as *'immutable piece of state, which may have a physical embodiment in a physical object, or a digital representation in a computer system'* [Moreau et al., 2010]. The state changes are produced by involving artifacts in a process, which is an *'action or series of actions performed on or caused by artifacts, and resulting in new artifacts'* [Moreau et al., 2010]. A process is performed by an agent, which is an *'contextual entity acting as a catalyst of a process, enabling, facilitating, controlling, or affecting its execution'* [Moreau et al., 2010].

To model the provenance, the open provenance model uses a directed graph. Nodes represent artifacts, processes and agents. Edges model the causal dependencies between the effect (source node) and its cause (destination node). The open provenance model defines all possible dependencies that are *used(R)*, *wasGeneratedBy(R)*, *wasControlledBy(R)*, *wasTriggeredBy* and *wasDerivedFrom*. The *R* in the first two dependencies specifies the role of the artifact during the processing, and the *R* of the third dependency donates to the role of the agent during the processing. Figure 2.3 depicts an example of an open provenance model specifying the updating (`P1`) of a health record (`A1` updated to `A2` in the role `R1` as the updated document) by a hospital (`Ag1` in the role `R2` as actor).

2.3.2 eXtensible Access Control Markup Language

The motivation behind the eXtensible Access Control Markup Language (XACML) is to combine many established ideas for access control policies using an extension language of XML [Moses et al., 2005]. XACML combines single rules and policies to sets, deals with multiple subjects and

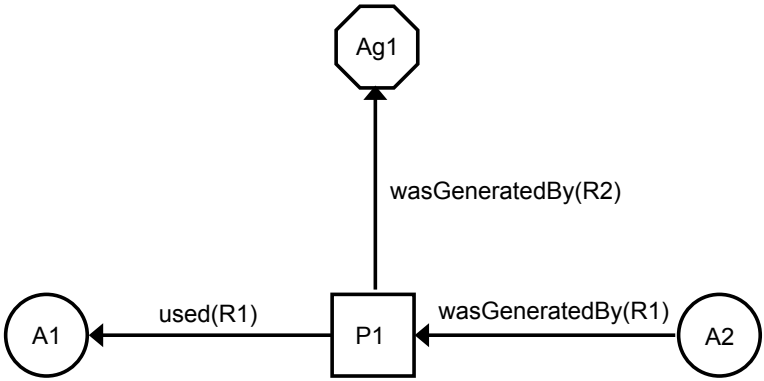

Figure 2.3: A Small Open Provenance Model Graph.

multi-valued attributes, supports policy enforcement, and much more.

The fundamental concept of XACML are policies that are defined as *'a set of rules, an identifier for the rule-combining algorithm and (optionally) a set of obligations'* [Moses et al., 2005], where a rule consists of *'a target, an effect and a condition'* [Moses et al., 2005]. In XACML, one can combine policies to policy sets. The policies can express permissions and prohibitions. They can also contain obligations which are defined as *'an operation specified in a policy or policy set that should be performed by the Policy enforcement point in conjunction with the enforcement of an authorization decision'* [Moses et al., 2005]. Apart from the policy language, XACML defines a model for policy enforcement that specifies an accurate procedure for accessing resources and fulfilling obligations.

2.4 Terminology

Throughout the presentation of our work, we are required to relate to various concepts. Some of these concepts have no clear definition in related work. We outline our terminology for a clear understanding:

Entity An *entity* is an *'independent, separate, or self-contained existence'* [Merriam-Webster, 2004] that could be involved in the processing of data (e.g. a Web server) or is related to data (e.g. the person concerned).

Action An *action* is *'the manner or method of performing'* [Merriam-Webster, 2004]. For instance, reading or updating a data instance are actions.

Actor An *actor* is an entity actively performing actions on the data. Depending on the granularity, an organization or an application running on a server or workstation can be examples of an actor.

Processing Step A *processing step* is a single step in the processing of data and consists of at least one performed action or a set of performed actions. For instance, the sharing of a data item between two entities is a processing step that may consists of read, transfer, receive and write actions.

Workflow A *workflow* is a sequence of actions or processing steps.

Process A *process* is a sequence of processing step executions to achieve a business goal.

Processing History The *processing history* is a trace of a process that constitutes a directed graph of executed processing steps.

In the following, we use the terms *category*, *item* and *instance* to refer to different abstraction levels of data. See Table 2.1 for an overview about the three abstraction levels.

Data Category We use *data category* to refer to the class of data (e.g. health records).

Data Item We use the term *data item* to refer to a specific piece of information (e.g. Jane Doe's health record).

Data Instance We use the term *data instance* as specific realization of a data item (e.g. the instance of Jane Doe's health record stored in a specific data base).

Table 2.1: Abstraction Levels of Data.

Term	Example
data category	health record
data item	Jane Doe‘s address ("Example Street 13, Some City")
data instance	"Example Street 13, Some City" @www.uni-koblenz.de/database

We use certain terms to relate persons and legal entities to data items.

Data Subject or Person-concerned We use the term *data subject* as it is defined by the EU Directive as *'[..] an identified or identifiable natural person [..]'* [The European Parliament and the Council of the European Union, 1995] the data is about. For instance, a patient is the data subject of her health record. We extend this definition to include subjects that are not natural persons.

Data Collector The *data collector* is the entity that has collected the data. For instance, the hospital or a staff member that collected the personal data of a patient.

Data Owner The *data owner* is the entity holding the legal ownership rights of a data item (e.g. copyrights). For instance, the hospital is the data owner of all health records, even if a specific staff member creates it.

Data Holder We use the term *data holder* for the entity in possession of a data item. For instance, if the hospital forwards a health record to a research institute, the institute will be the data holder while the hospital is still the data owner.

Finally, we use certain terms to address certain states of obligations.

Instantiated During the execution of a process, an obligation is *instantiated* when its trigger condition is fulfilled. An instantiated obligation can have one of the following states: *active*, *fulfilled* and *violated*.

Active An obligation instance will be *active* if the obligatory processing steps have not been executed yet.

Fulfilled An obligation instance will be *fulfilled* if the obligatory processing steps have been executed.

Violated The obligation instance will be *violated* if the obligatory processing steps have not been executed and can not be executed anymore.

3 Scenario and Requirements for Managing the Distributed Processing of Data

In Chapter 1, we identified the need of data subjects and owners to manage the distributed processing of their data. Managing the distributed processing of data requires the consideration of various legal, contractual, organizational and technical aspects. For instance, privacy laws affect the processing of privacy-related data in any kind of environment, including distributed environments. In the member states of the European Union, these are privacy laws implementing the EU Directive 95/46/EC [The European Parliament and the Council of the European Union, 1995]. For processes distributed among organizations located in different countries, special privacy agreements define requirements, such as the Safe Harbor Frameworks[1], which is implemented between the European Union, the United States and Switzerland. Further rules for the usage and protection of data are defined in contractual agreements concluded between the involved parties. In the Internet, the rules are often specified by general service agreements for end customers. And between organizations service level agreements are closed. Organizational and technical issues affect the implementation of the legal and contractual requirements. For instance, the distribution of processes among independent, autonomous organizations hampers the monitoring and the auditing of the processing.

As the aim of this work is the management of the distributed processing of

[1] `http://www.export.gov/safeharbor/`, retrieved Nov. 23th, 2010.

the own data, all these aspects must be considered before we can derive requirements. We introduce a scenario in Section 3.1 and discuss the involved stakeholders, the details of the process with the policies specifying rules of the processing, and the technical architecture implemented in the scenario. The legal and contractual aspects of processing data in distributed environments are discussed in Section 3.2 and in Section 3.3. In section 3.4, we analyze organizational and technical issues that arise from auditing and from using policies in distributed environments. Based on (a) the scenario, (b) the legal and contractual aspects, and (c) the organizational and technical issues, we derive requirements for formal models and methods that aim at managing the distributed processing of data.

3.1 Scenario - Distributed Processes in Health Care

Managing the processing of data is required in distributed environments as depicted by the following every-day scenario. The scenario presents the *Middle Rhine Hospital*, a local hospital, and its research cooperation with the *University of Koblenz*. Health records are shared between both organizations as part of the cooperation. The scenario depicts the need to audit data processing in environments distributed across multiple organizations. The scenario shows that it is necessary to restrict the processing of the data by means of policy rules expressing permissions, denials and obligations.

3.1.1 Stakeholders

In the scenario, different stakeholders are involved who have different relations to the data and motivations for processing the data. The following list depicts these stakeholders:

Middle Rhine Hospital The Middle Rhine Hospital is a local hospital. The core business of the Middle Rhine Hospital are health services provided

for patients. To improve their health services, the hospital is doing research in various fields of medicine and health care. As its core business is not research, it cooperates with external research institutions. Different fields of research require the analysis of real data (e.g. cancer recognition in X-rays). The health records of patients are a source for such data and are shared in this scenario. To share the records, the hospital provides access for the research institutions on a server via Internet. Since health records are highly sensitive data, the hospital wants to be able to audit their complete processing and to restrict their use. Beside this, the patients are asked to specify additional policies restricting the processing of their records. If requested by the corresponding patient, the hospital will delete a health record or provide it to the patient. Regarding the health record, the hospital is the data collector and the data owner.

University of Koblenz An institute of the University of Koblenz performs research on image processing of X-rays. The University requires real world data about patients, to analyze the correctness of research results and the applicability of the approach for health service providers. A cooperation with the Middle Rhine Hospital allows for using the required data and make the necessary analysis. The University and the hospital placed a contract about the cooperation. The contract allows the University to access health records of patients of the hospital. However, the hospital restricts the access to health records of patients that gave their explicit consent. The contract restricts the use of health records for research purposes only. In addition, the University is not allowed to forward the data to other organizations. The hospital requires that the University monitors the processing of health records. The data captured by the monitoring mechanism must enable the hospital to audit the processing performed by the University. As long as health records are handled or stored at the University, the University is the data holder of the regarding records.

Jane Doe Jane Doe is a patient of the Middle Rhine Hospital. During her admission, Jane Doe demands that the hospital has to hand out her health record after her treatment ends. She gives the hospital the permission to use her record for research purposes and to share her record for this purpose with the University of Koblenz. However, Jane Doe specifies the restriction that only her de-identified health record can be used for research purposes. Because of other serious health issues that are not related to her current hospital stay, but occur in her record, she demands that the hospital restricts the access to the related parts of her health record. Regarding the health record, Jane Doe is the Data Subject, the person concerned.

3.1.2 The Process

For our considerations we need a detailed overview of the processing of a health record by the Middle Rhine Hospital and by the University of Koblenz. In addition, we require details about the policies issued by the hospital and Jane Doe for restricting the processing of her health record. Figure 3.1 depicts an action-by-action run-through of the processing of her health record while she is treated as a patient in the hospital and through the research performed on her data. The steps depicted in the figure are:

1. *Admission:* With the admission of Jane Doe as a patient of the Middle Rhine Hospital a new health record is created by the hospital. At the beginning the health record is empty. The create action is logged by the hospital. As part of the admission the personal information of Jane Doe is collected and stored in the record. The record is stored on a file server of the hospital and is only accessible by the hospital. Because of organizational and legal restrictions, the hospital is initially only allowed to use the health record for the purpose it has been created for. The purpose is treating Jane Doe's cerebral tumor. Even the hospital is not allowed to perform research on the record without the consent of Jane Doe. The regarding restrictions are expressed by `Policy 1`,

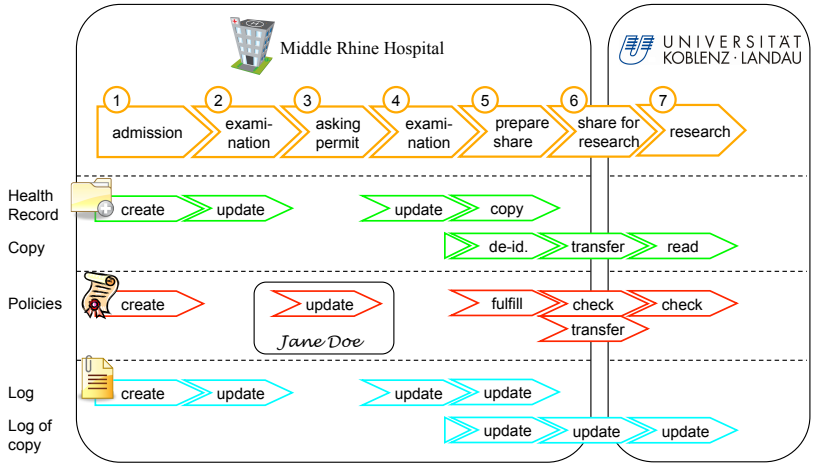

Figure 3.1: Process Overview of Privacy Scenario.

which is associated with Jane Doe's health record:

- `Policy 1.1`: *The Middle Rhine Hospital is allowed to use Jane Doe's health record for treating Jane Doe as patient.*
- `Policy 1.2`: *The Middle Rhine Hospital is not allowed to use Jane Doe's health record for any other purposes without explicit permission.*
- `Policy 1.3`: *The Middle Rhine Hospital is not allowed to share Jane Doe's health record with other organizations for any purposes without explicit permission.*

A more general policy specifies that everyone is permitted to hand out health records to patients. Based on this policy, Jane Doe asks that the Middle Rhine Hospital has to transfer her record to her after her treatment ends. The additional permission and obligation are specified

by `Policy 2`:

- `Policy 2.1`: *Everyone is permitted to transfer Jane Doe's health record to Jane Doe.*
- `Policy 2.2`: *Jane Doe demands to receive her health record after her discharge.*

2. *Examination:* When the examination of Jane Doe starts, the results are stored in her health record. The actions performed to update the health record are logged by the Middle Rhine Hospital. The examinations are carried out during the hospital stay of Jane Doe and possibly afterwards.

3. *Asking consent:* Patients may choose if their data can be used for research or not. Thus, at some point during the examination Jane Doe, is asked whether she permits the sharing of her data. The hospital provides the questions in natural language, e.g.:

```
(1) Do you allow the Middle Rhine Hospital to
use your health record for research purposes?
Yes / No.
```

```
(2) Do you allow the Middle Rhine Hospital to
forward your health record to the University
of Koblenz for research purposes?  Yes / No.
```

As Jane Doe gives her consent that the hospital can use her record for its research, `Policy 3` is created to express this permission:

- `Policy 3`: *The Middle Rhine Hospital is allowed to use Jane Doe's health record for research purposes.*

Additionally, she allows for sharing her health record with the University for research purposes. However, the contract between the hospital and the University does not allow the University to forward the data to any other organization as specified by `Policy 4`.

- `Policy 4.1`: *The University of Koblenz is allowed to process Jane Doe's health records for research purposes.*
- `Policy 4.2`: *The University of Koblenz is not allowed to transfer health records to any other organization.*

An internal policy of the hospital requires that the patient's permission has to be approved by a doctor. The hospital creates `Policy 5` to express these restrictions:

- `Policy 5`: *The sharing of health records has to be approved by Jane Doe and the approval must be confirmed by a doctor before the record is accessed by any other organization.*

In addition to giving her consent, Jane Doe can define additional rules for sharing her health record. Jane Doe additionally demands that her record is only used after it has been de-identified by exchanging her personal information by a pseudonym. `Policy 6` specifies this restriction:

- `Policy 6`: *Jane Doe's personal information that is contained in her health record has to be de-identified before the record is shared.*

Neither the health record nor the log are updated in this step.

4. *Examination:* As stated before Jane Doe has another health condition that requires special confidentiality. The examination results related to this condition are also stored in the health record and the associated processing steps are logged. Jane Doe demands that the examinations and examination results relating to the condition require to be hidden

before the record is used for research. This restriction is specified by means of `Policy 7`:

- `Policy 7` *Information contained in Jane Doe's health record about other conditions than her cerebral tumor have to be hidden before the record is used for research purposes.*

5. *Prepare Sharing:* As Jane Doe requires the hiding of confidential information and the de-identification of her health record, the hospital has to prepare the health record before sharing it with the University. The hospital copies Jane Doe's health record. The copy is then shared with the University after two preparatory steps. The first step is that the copy is de-identified[2] by replacing all data identifying the patient, such as her name, her address, and other information that allows for connecting the health record with Jane Doe, by a pseudonym. As second step, the hospital removes the confidential information about Jane Doe's other health conditions. The hospital logs the creation of the copy and the changes performed on it.

6. *Share for Research:* After the copy has been created and prepared by the Middle Rhine Hospital, it will be shared with the University of Koblenz under the condition that all policies are fulfilled. To this end, the hospital stores the copy on a Server that can be accessed by the University via the Internet. The sharing is carried out as soon as the University accesses the copy of Jane Doe's health record on the server of the hospital. Through the access the copy of the record is transferred form the hospital to the University. In addition to the copy of the record, the associated policies are also transferred. The data transfer has to be logged by both parties. The hospital logs the sending of the copy and the University logs its reception. Neither the health record nor the copy are modified in this step.

[2] `http://www.ucdmc.ucdavis.edu/compliance/guidance/privacy/deident.html`, retrieved April 18th, 2011.

7. *Research:* After receiving the records, the University performs research with the data contained in the copy of Jane Doe's health record. During the processing of Jane Doe's data the University controls the compliance with the associated policies. Additionally, the University logs all actions performed on the data, like accessing or analyzing.

8. *Archiving:* After Jane Doe's treatment ends, the health record is archived by the Middle Rhine Hospital. To this end, nurses have to transfer all health records to the archive when a patient is discharged. After a health record is archived, the archive is not allowed to hand it out to patients. However, members of the hospital staff are allowed to share health records with other staff members and with the corresponding patients. The additional `Policy 8` expresses these circumstances:

   - `Policy 8.1`: *Nurses must transfer health records to the archive after the patient's discharge.*
   - `Policy 8.2`: *The archive is denied to transfer health records to patients.*
   - `Policy 8.3`: *The staff is permitted to transfer health records to other staff.*

   The hospital may delete health records when they are not needed anymore. As Bob is responsible for data protection in the hospital and he is the only person allowed to delete health records, it is his duty to delete records. `Policy 9` specifies the corresponding permit and obligation rule:

   - `Policy 9.1`: *Bob is permitted to delete health records.*
   - `Policy 9.2`: *Bob must delete all health records he receives after the patient has been discharged.*

3.1.3 Technical Architecture

Beside the single processing steps performed on the health record, the underlying technical architecture is of relevance for our considerations. The technical architecture describes the combination of different systems, the distribution of the systems and the communication as well as the data transfer between the different systems. Figure 3.2 depicts the technical architecture of the scenario.

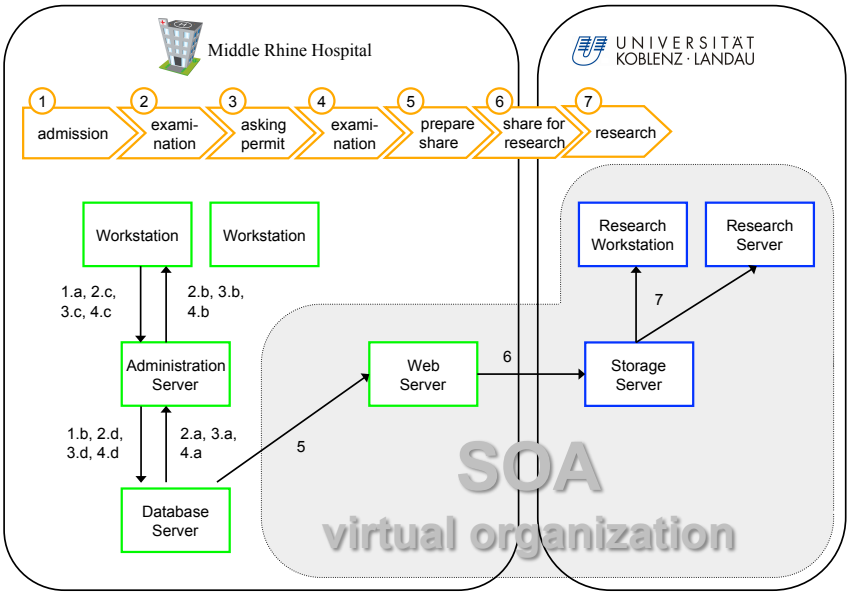

Figure 3.2: The Technical Architecture of the Scenario.

- **Database Server:** The Middle Rhine Hospital operates a dedicated server running a database with the health records of its patients. The server and its database can not be accessed directly by the hospital's

employees. To enable applications to access the data, the database management system running on the server provides an API. The interface allows for integrating the database into other applications. Authorization mechanisms assure that only certain applications running on other servers have access to the data. The database server logs all actions performed on the data stored in its database.

The database server is also used as an archive. The archive is part of the same database. Health records stored in the archive are not accessible without special clearance.

- **Administration Server:** The Middle Rhine Hospital uses a management software specialized for the administration of patients and health records. The software is running on a dedicated server operated by the hospital. The administration software is a client-server solution with a proprietary communication protocol. To access health records via a workstation, the workstation must be authorized. The server logs all actions performed on the patients' records. The server itself stores the records on the database server using the API that is provided by the database management system.

- **Workstations:** The medical staff of the Middle Rhine Hospital uses client applications of the management software to access and manage health records of patients. The client application is running on workstations. Using the workstations requires authentication by the user and only authorized users may access the data. The workstations are operated by the different departments of the hospital and are connected with the server via the Intranet. The client application running on the workstation logs all actions the user performs.

- **Web Server:** The Middle Rhine Hospital operates a Web server for sharing the records with external research institutes. The Web server provides a Web service as interface to export health records. Only authorized clients can access health records by means of the Web service.

Before sharing a health record, the Web service creates a copy of the health record stored on the database server and prepares the copy to be shared (e.g. de-identification). The Web server logs all data transferred by the Web service and the Web service itself logs all actions it performs on the records.

- **Storage Server:** The University of Koblenz uses a specific server to access the health records through the Web service provided by the Middle Rhine Hospital. The access occurs via the Internet. Specific security mechanisms are used (e.g. SSL) to protect the data transfer. The copies of the health records held by the University are stored in a database running on this server. The server and the database log all actions performed on the health records. Through the connection of the hospital's Web server and the University's storage server, a virtual organization emerges [Davidow and Malone, 1992].

- **Research Server and Workstations:** The researchers at the University of Koblenz use different workstations and servers to perform their research with the health records. Authorization and authentication mechanisms are used to protect the records. Research servers and workstations access the health records on the storage server via the University's Intranet. All research servers and workstations are required to log all actions performed on health records.

As the Web server, storage server as well as research server and workstations provide and consume services, their part of the technical architecture constitutes a service-oriented architecture (Figure 3.2).

3.2 Legal Aspects

Different laws regulate the processing of data, e.g. copyright laws or privacy laws. These laws are also applicable for the data processing in distributed

environments. The analysis of these laws and their application is an extensive task and requires expert knowledge as from a lawyer.

We analyze the legal foundation of handling personal information in Europe, such as the processing of Jane Doe's health record. As for any kind of data, certain laws apply for handling Jane Doe's health record. The laws specify rules for handling data in general and privacy-related information in particular. In the European Union, directives define objectives for the legislation of the member states of the European Union (EU). An EU Directive *"[..] is binding on the Member States as to the result to be achieved but leaves them the choice of the form and method they adopt to realise the Community objectives within the framework of their internal legal order."*[3] All EU Member States need to implant the EU Directive into national legislation.

A Directive relating to the processing of data is the EU Directive 95/46/EC [The European Parliament and the Council of the European Union, 1995] that defines objectives for privacy laws of private persons. We have chosen the EU Directive 95/46/EC as starting point to derive legal requirements for the distributed processing of data which is called: *"Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data"*

The Directive uses the terms *'controller'*, *'data subject'*, and *'processing'*. Article 2 defines the terms in the context of the Directive:

- A 'controller' *"[..] shall mean the natural or legal person, public authority, agency of any other body which alone or jointly with others determines the purposes and means of the processing of personal data; ..."*,
- a 'data subject' is *"[..] an identified or identifiable natural person [..]"*, and

[3] `http://eur-lex.europa.eu/en/droit_communautaire/droit_communautaire.htm#1.3.3`, retrieved April 18th, 2011.

- ‘processing’ *“[..] shall mean any operation or set of operations [..] whether or not by automatic means, such as collection, [..] storage, adaption or alteration, [..] use, disclosure by transmission, [..]”.*

We start with Article 10 *‘Information in cases of collection of data from the data subject’* and Article 11 *‘Information where the data have not been obtained from the data subject’*, which specify the information that must be provided to the person concerned by notification. In detail, Article 10 regulates:

“[..] the controller or his representative must provide a data subject [..]

- ***(a)** the identity of the controller and of his representative, if any;*
- ***(b)** the purpose of the processing for which the data are intended;*
- ***(c)** any further information such as the recipients or categories of recipients of the data [..]”*

Article 11 also compels the service provider to notify the persons concerned, even if the data is not obtained from them. Article 10 and Article 11 describe the need for an information service attending to the information rights of private persons. Such an information service can support the process of notifying the person concerned by providing the needed information.

Article 12 *‘Right of access’* describes details that have to be given to the person concerned by the service provider if requested:

“Member states shall guarantee every data subject the right to obtain from the controller:

- ***(a)** without constraint at reasonable intervals and without excessive delay or expense:*

- *confirmation as to whether or not data relating to him are being processed and information at least as to the purposes of the processing, the categories of data concerned, and the recipients or categories of recipients to whom the data are disclosed,*
- *communication to him in an intelligible form of the data undergoing processing and of any available information as to their source,*

.. *"*

Outside the European Union, there are similar laws. In the United States, the Health Insurance Portability and Accountability Act (HIPAA) of 1996 (P.L. 104-191) [104th United States Congress, 1996] specifies privacy and security rules for information regarding heals.

3.3 Contractual Aspects

The distribution of data does not only occur between departments of one organization but also beyond organizational boundaries. Contracts are a standard means to define rules for inter-organizational data processing. The contracts are concluded between the involved organizations and entities. Like laws, contracts define rules for data processing. They differ depending on various aspects, e.g. the involved organizations, the kind of service, and the binding laws. Contracts between two companies are not often publicly accessible. Only the general terms and conditions of open available services are accessible, such as the general privacy terms and conditions of facebook[4]. These also vary from service provider to service provider, but through binding laws there is a common foundation. One sort of binding laws are the privacy laws we have already discusses in section 3.2. Thus, we apply the legal aspects discussed before to the distributed processing of data in general and not only the processing of private data.

[4] `http://www.facebook.com/policy.php`, retrieved April 18, 2011.

Contracts define rules for processing data. Entities processing the associated data must comply with these rules. The rules can be expressed by policies. Our scenario contains a contract between the Middle Rhine Hospital and the University of Koblenz. `Policy 4.2` specifies parts of the contract as policy. The policies issued by Jane Doe are also part of the contract between Jane Doe and the hospital. To account for policies, a method for handling them is required.

3.4 State of the Art and Requirements

As discussed in Chapter 1, we need the capability to manage the distributed processing of data by means of policies in combination with audits. The scenario also depicts this by requiring mechanisms which allows Jane Doe to audit the past processing of her health record and to specify policies to restrict the future processing of her data. Analyzing the weaknesses of existing means, we can identify various organizational and technical issues that hamper the audit and compliance checking with policies. These organizational and technical issues need to be considered before deriving requirements for managing the processing of data in distributed environments.

3.4.1 Organizational and Technical Issues

We identify multiple organizational and technical issues while audits should take place in distributed environments. In addition, we identify various issues with the evaluation of policies that are not directly related to the distributed environment but grow more acute in such environments. In the following, we provide an overview of the identified issues. We start with the auditing of distributed processes.

For several reasons existing logging mechanisms (e.g. [Hallam-Baker and Behlendorf, 1996], c.f. Chapter 5) are not sufficient to gain a full overview of a workflow that is distributed among multiple organizations. The loose

coupling of services as provided by service-oriented architectures also hampers the generation of such an overview. The main reason is that existing logging mechanism are tailored to perform logging within one execution environment (e.g. [Lonvick, 2001]). Because of the diversity of execution environments and of a lack of standardized interfaces for exchanging logs, distributed logs cannot automatically be combined into one log. Without a continuous log, a complete history of the processing cannot be reconstructed. Existing means for modeling distributed processes (c.f. Chapter 4) are not sufficient to model the processing of one data instance or data item. Existing models specify the actions performed on all data items processed in one business process (e.g. [van der Aalst et al., 2004]) or the actions performed by one entity on all processed data items, independent of the process (e.g. [Barth et al., 2007]). All these weaknesses are reinforced by the different process ownerships in distributed workflows and the heterogeneous environments thereof. We derive the following issues regarding the audits in distributed environments:

Issue 'Loosely-coupled Architectures': At the level of implementation, the distributed processing of data in the Web hampers the generation of an overview. Paradigms, such as service-oriented architectures (SOA) and software as a service (SaaS), and associated standards, such as SOAP, allow for a loose coupling of services. The loose coupling enables the incorporation of services which are defined and implemented independently of each other running on different middleware and execution environments. In such environments, cross cutting concerns, such as logging, are hard to realize if they are not standardized at the interface level. Workflows can be configured in an agile manner making it difficult *a posteriori* to assert which organizations had accessed the data during the execution of the workflow.

Issue 'Lack of Process Awareness': In order to report on the previous handling of data, an organization must be aware of and account for their

internal data flows at a specific level of granularity. Depending on the law or contract, the level of granularity may be very fine-grained or more abstract (e.g. has Jane Doe be treated in a certain department or by a specific doctor). Such awareness on a fine-grained level is rarely available explicitly for cross-organizational workflows. Single organizations may already lack a global model of the processing through independent departments as well as through lacking or incomplete process specifications.

Issue 'Recurring Processing': The creation of the first instance of a data item is part of one specific business process. Instances of the data item may be involved in several other processes and data instances may outlive the process in which they have been created. Jane Doe's record may be stored in a data base and reused at future hospital visits of Jane Doe. For an exhaustive answer, an overview about the processes, where all instances of one data item are involved in, is required. Such an overview is often lacking as the connections between business processes are not modeled.

Issue 'Distributed Process Ownership': As autonomous organizations do not want to be managed or audited by third parties, transparency decreases with increasing distributed process ownerships. Hence, on the level of cross-organizational workflows the lack of process awareness is further increased.

Besides the auditing of distributed processes, the structure of the processing history is a source of issues for the evaluation of policy conditions based on the history. Some existing policy languages allow for building policies containing conditions based on provenance information (e.g. [Kagal et al., 2003], c.f. Chapter 6). They lack a formal specification of how to specify or access provenance information as well as its temporal structure and are too weak to express policies based on provenance information. They also do not provide means to connect data with policies in distributed environments

and to protect sensitive information in those environments. We identify the following organizational and technical issues:

Issue 'Lack of Linkage': Certain policies often relate to a specific data item, e.g. Jane Doe's policies regarding the processing of her health record. If a certain entity wants to process a specific data item, the entity will need to know which processing steps are permitted or restricted. Permissions and prohibitions defined by policies often depend on conditions that depend on contextual information, e.g. the University of Koblenz will be allowed to access the health record, if it has been de-identified. Within organizations, data and its meta data as well as associated policies are often stored independently and only partially linked, e.g. directed links from policies to data but not vice versa. In distributed environments the lack of links even increases by not providing the necessary meta data connected with transferred data.

Issue 'Temporal Structures' Permission and restrictions may depend on specific states of the processing, e.g. the transfer is allowed *after* Jane Doe gave her consent. Conditions of policies may relate to the history of the data processing and the temporal structures of such histories. Temporal structures are of high complexity and demand for complex methods for handling. In cross-organizational workflows, where already process overviews are missing, keeping track of the processing history and its temporal structures is even worse.

Issue 'Sensitive Information' The processing history may contain sensitive information about the processed data (e.g. performed examinations) or confidential information about organization-internal processes. Protection mechanisms (e.g. encryption) are used to protect this information. At the same time, parts of this information are required to interpret policy conditions.

Sometimes the current processing does not need to be restricted, but contracts or laws demand a certain reaction in the future (e.g. the health record

must be transferred to Jane Doe after her discharge). Therefore, some policy languages foresee the possibility to specify obligations, e.g. [Bradshaw et al., 2003,Hilty et al., 2005,Lupu and Sloman, 1999,Moses et al., 2005] (cf. Chapter 7). However, they do not provide methods to decide the existence of a future execution that meets all obligations. We derive the following issues:

Issue 'Invalid Future Executions' The execution of processing steps may conflict with active obligations in multiple ways. Fulfilling obligations can violate policies, e.g. the need to transfer a recored, which must not be transferred. Newly instantiated obligations can conflict with active obligations, e.g. a record must be kept and deleted. Obligations can hamper the execution by generating a loop, e.g. X must send the record to Y and Y has to send it to X. The missing of a semantic specification of the dependencies between obligations and the execution of processing steps impairs the recognition of invalid future executions.

Issue 'Valid Processing Steps' To prevent conflicts, one has to decide whether all future obligations can be meet before continuing the processing. For each current processing step the actor must decide whether it is valid or invalid. A valid processing step does not prevent the fulfillment of future obligations and does not violate any policy. Only an according decision procedure is able to decide whether a processing step is valid or not.

3.4.2 Legal and Contractual Requirements

To manage the distributed data processing, we derive requirements for a solution that overcomes the identified organizational and technical issues with respect to the discussed legal and contractual aspects. For the legal aspects, we can derive the Requirement Identifiability of an information service from the Articles 10 and 11 of the EU Directive:

Legal Requirement 'Identifiability': *The provided information must be sufficient to identify the entities processing the data (e.g. service provider) as well as the recipients and sources of personal data.*

The difference between non-distributed and distributed environments is that not only one entity processes the data but many. The information service must be able to provide a complete list of all involved entities and their clear identification. The *Issue of Lack of Process Awareness* and the *Issue of Distributed Process Ownership* hamper the identification of the entities. In the scenario, the Middle Rhine Hospital and the University of Koblenz are legal entities processing the data. On the organizational level different departments are entities and on a technical level the different servers are single entities. Depending on the required granularity all these entities must be identifiable. As for the entities processing the data, the sources of the data and its recipients must also be clearly identified. In our scenario, the sources are the medical staff collecting information about Jane Doe during interviews and examinations. The University is a recipient of Jane Doe's health record. We can state that all involved entities must be clearly identified.

From Article 12, but also from Article 10 and Article 11, we can derive the Requirements Accessibility and Exhaustiveness:

Legal Requirement 'Accessibility': *The information service must enable the person concerned to access information about the complete processing of her personal data at any time.*

The information about the data processing must be accessible by the data subject and data owner. The information service has to provide an interface for these entities to access the data about the processing. The interface has to not only be accessible but also usable for the entity. While for organizations as the Middle Rhine Hospital a Web service is reasonable, Jane Doe must be able to access the data by submitting her

request by mail or fax. The accessibility also implies that a complete overview of the executed parts of the process is required at any time. This is hampered by the *Issue of Lack of Process Awareness*, *Issue of Loosely-coupled Architectures* and *Issue of Distributed Process Ownership*. In our scenario, the hospital must provide Jane Doe with all information about the processing of her health record. It has to provide access to all parts of the information about the processing. Thus, the hospital is not allowed to hide the processing of Jane Doe's health record performed by certain internal entities or the transfer of the data to the University of Koblenz.

Legal Requirement 'Exhaustiveness': *The information service must inform about which personal data item is processed, how it is processed, and why it is processed.*

The given answer must be exhaustive and in an intelligible form. It must cover all processing steps performed on the data independent of the business process (cf. *Issue of Recurring Processing*). The answer must contain details of each processing step, such as its category (e.g. de-identification) and purpose (e.g. research). The different purposes and categories must be defined in domain ontologies. In the scenario, the Middle Rhine Hospital has defined a domain ontology that is used by the hospital itself and the University of Koblenz to describe the processing steps performed on patient's health records. The first step in the scenario is the creation of Jane Doe's health record by the Middle Rhine Hospital. The category of this processing step is the *creation of a health record* and the purpose is the *treatment of a patient*.

If this information can be obtained from an information service, security mechanisms must assure that the confidentiality of the data will be preserved. The information that personal data is processed may also be considered as personal data. We assume that a log contains personal data as long as it is not proved otherwise. Analogously, we have to assume that logs contain

information about the internal processes of the entities processing the data (e.g. the order of processing steps). This data must be protected, as well.

Legal Requirement 'Confidentiality': *The information service must ensure that only the entities performing a processing step, the person concerned and data owner have access to information about the step.*

We assume that information about the internal processes of organizations is confidential. Thus processing histories are confidential (cf. *Issue of Sensitive Information*). At the same time, the processing history must be provided to the person concerned or data owner. In the scenario, the University of Koblenz must inform Jane Doe about the processing. Because the University does not know Jane Doe, the University has to pass the information on to the hospital that provides the information to Jane Doe. To keep the confidentiality of internal processes, a protection mechanism is required.

Based on the above mentioned contractual aspects we can point out the following requirements for a policy mechanism:

Contractual Requirement 'Accessibility': *The policy mechanism must provide the required policies to anybody accessing the associated data.*

To adhere to policies, an entity must check whether an performed action complies to effective policies. To this end, the entity requires access to the policies and must be able to identify the policies associated with the data. The *Issue of Lack of Linkage* hampers the access. In our scenario, the University of Koblenz needs to know the policies regarding Jane Doe's health record issued by the Middle Rhine Hospital and Jane Doe. Furthermore the University must not have access to the policies of other patient's health records (cf. *Legal Requirement Confidentiality*). Thus, we require a mechanism providing the needed policies.

Contractual Requirement 'Availability': *The information required by policy conditions must be available even if confidential parts of the data and of the provenance information stay hidden.*

Policy conditions must allow for relating to contextual information or the history of the processing. For instance, the University of Koblenz needs to know whether Jane Doe's health record has been de-identified. This information must be available to all entities evaluating policies. Data protection mechanisms may hamper the availability of the required information (cf. *Issue of Sensitive Information*). Hence, we require a mechanism to provide the needed information to entities evaluating policy conditions. In the scenario, the Middle Rhine Hospital will have to provide the de-identification status of Jane Doe's record to the University of Koblenz, even if it encrypts the provenance information.

Policy conditions may depend on the previous data processing and the temporal aspects of the processing (e.g. order of steps in the history). We define the following requirement:

Contractual Requirement 'Expressiveness': *Policies must allow for conditions based on temporal aspects of processing histories.*

One must be able to depend permissions and restrictions on the history of the processing of data. The language used to specify conditions has to allow for expressing this dependencies. This is hampered by the *Issue of Temporal Structures*. For instance, `Policy 5` demands for getting Jane Doe's consent before approving her free-will by a doctor before the record may be shared. To control whether this policy is violated, one has to be able to express exactly this temporal dependency between these two actions.

Contracts may define obligations which must be fulfilled in the future.

Contractual Requirement 'To Fulfill Obligations': *Policies must allow for specifying obligations and to decide whether all active obligations can be meet in the future.*

To comply to contracts, entities must be able to fulfill future obligations. In our scenario, `Policy 2.2` demands that Jane Doe's health record must be transferred to Jane Doe after she is discharged. The current processing may conflict with the fulfillment of obligations (cf. *Issue of Invalid Future Executions*). For instance, the hospital may delete Jane Doe's health record before it is transferred to her. To verify that a processing step does not render an obligation unfulfillable, the policy language must provide a procedure to decide whether all future obligations can be met (cf. *Issue of Valid Processing Steps*).

To define formal models and develop methods for auditing and restricting the distributed processing of data, we must not only adhere to the legal and contractual requirements but also have to overcome the organizational and technical issues. To this end, we point out the following additional requirements for our formal models and methods.

Requirement 'Well-defined Semantics': *To avoid ambiguities and to reach standardization, the provenance information must be formalized using a language with a well-defined semantics.*

Privacy laws ask for an exhaustive answer that provides detailed information about the processing of the data (cf. *Legal Requirement Exhaustiveness*). The data must be provided in a understandable manner, to reach interoperability between the involved entities. In our Scenario, the Middle Rhine Hospital must be able to understand the executed processing steps performed by the University. If the hospital does not understand the provided processing history, it will not be able to audit the processing. A domain-specific language that is based on well-defined semantics provides a mean to align taxonomies and

associate concepts to each other. Such a language supports the organizations to overcome the *Issue of Distributed Process Ownership* and the *Issue of Lack of Process Awareness* by enabling a common understanding of the provenance information.

Requirement 'Standardized Interfaces': *Standardized interfaces are required to access and share provenance information between all involved parties.*

The person concerned and the data owner have the right to access the information about the processing of their data, as demanded by the *Legal Requirement Accessibility*. They have to be provided with the provenance of their data. Using a provenance-aware policy mechanism requires entities to access the provenance information to check whether processing steps are permitted or not, see *Contractual Requirement Accessibility*. The *Issue of Distributed Process Ownership* hampers the provision of the required information and the *Issue of Loosely-coupled Architectures* increases the problem by allowing for connecting different platforms. To address these issues, we require standardized communication means.

Requirement 'Level of Granularity': *The used language must be able to express details about the performed actions, their actors, their purposes and their order. These details must have the required level of granularity.*

The *Legal Requirement Identifiability* in combination with the *Legal Requirement Exhaustiveness* demands a sufficient level of detail about the processing. An insufficient level of information hampers the common understanding of distributed processes and results from the *Issue of Distributed Process Ownership*. Detailed information is required to create an overview that helps to overcome this issue and the *Issue of Lack of Process Awareness*.

4 A Model for the Distributed Processing of Data

In Chapter 1 and Chapter 3 we identified the need to audit the distributed processing of data as accounting for actions performed on data may be legally and contractually required. We also discussed that organizational and technical issues, such as the *Issue of Loosely-coupled Architectures*, the *Issue of Lack of Process Awareness*, the *Issue of Recurring Processing*, and the *Issue of Distributed Process Ownership*, hamper the auditing.

To control the compliance with laws, contracts, or policies, a data provider may request information about the processing and whereabouts of their data. The answer has to contain details defined by the contracts or laws. This can be information about who processed the data as well as why and how the data has been processed, cf. *Legal Requirement Identifiability* and *Legal Requirement Exhaustiveness*. If there is a model of the process which facilitates a detailed overview, the information can be derived from this model. Most frequently such an overview is lacking, even for internal workflows. Our hypothesis is that the distributed processing of data can be audited, even if such global model of the processing is lacking a-priori the process execution (cf. Section 1.1). To achieve the auditing, we require a formalism to model the data processing in distributed environments. And we need a methodology that allows for auditing and fulfills the requirements we identified in Section 3.4.

To this end, we first discuss different execution models and their relations to each other in Section 4.1. Based on the execution models, we specify a formal model by means of colored Petri nets in Section 4.2. We call this

model *DiALog*: Distributed Auditing Logs. DiALog can be used to model the logical execution, to achieve a global model as well as to model the reconstructed execution. Deriving a reconstructed execution from the physical execution has to fulfill certain qualities to constitute a global model. We define the qualities soundness and completeness between the reconstructed execution and the global model in Section 4.3. The qualities are required to assure the correctness of the audit performed by means of the reconstructed execution. We use these qualities in Chapter 5 to develop a logging mechanism that accounts for the requirements we derived in Chapter 3.

4.1 Execution Models

The execution of a process can be modeled from various perspectives. The main model is the global model that defines the processing from a global view. As discussed in section 3.4.1, the *Issue of Lack of Process Awareness* and the *Issue of Recurring Processing* hamper the creation of a global model. Various other models exist that result from the execution of the process. The *physical execution* of the process (see Figure 4.6) is the actual execution of the process in a physical and technical environment (e.g. calling and executing a Web service). The specification of the physical execution is given by the *logical execution*. The logical execution is a formal model of the execution, which is not necessarily defined before the physical execution is performed. In this case, it is defined by performing the physical execution. The logical execution and the physical execution constitute both a global model of the execution.

Observing the physical execution leads to logs that constitute the *monitored execution*. We can create a *reconstructed execution* from the monitored execution. If the reconstructed execution fulfills specific qualities with respect to the logical execution, the reconstructed execution will constitute a global model of the processing. The methodology we are looking for must allow for generating reconstructed executions fulfilling these qualities.

4.1.1 Global Models

Global models describe how data is processed and by whom as asked for by the *Legal Requirement Exhaustiveness* that we derived in Chapter 3. To this end, a global model specifies the actions performed on the data item during its lifespan. The lifespan of a data item ranges from the creation of its first instance to the deletion of its last instance. During its lifespan a data item may be involved in the execution of various business processes. The global model specifies the parts of these processes where actions are performed on the data. Each of the business processes may span multiple organizations and so the global model. The global model implements contractual agreements.

4.1.2 Logical Execution

The execution of the global model can be specified by means of a labeled transition system (cf. Definition 4.10). We call such a labeled transition system, specifying the execution of the global model, the *logical execution*. To formalize the logical execution, we have chosen colored Petri nets (see Definition 4.1). A snippet depicting a logical execution modeled in DiALog is shown in Figure 4.1.

Example 4.1: *Example of a Logical Execution*

Figure 4.1 shows a snippet of the logical execution performed on Jane Doe's health record as described in the scenario in Section 3.1. The snippet depicts the sharing of the health record between the Middle Rhine Hospital and the University of Koblenz (step 5 and 6). The record (represented by the token matching m) is stored in the database ($database_{DB\ server}$) of the database server of the Middle Rhine Hospital at the begin of step 5. The first action performed on the health record is the creation of a copy ($copy_{sharing}$) and has the purpose of sharing the record. The copy (represented by the token matching n) is created at the database management system ($DBMS_{DB\ server}$) also running on the database server. From there,

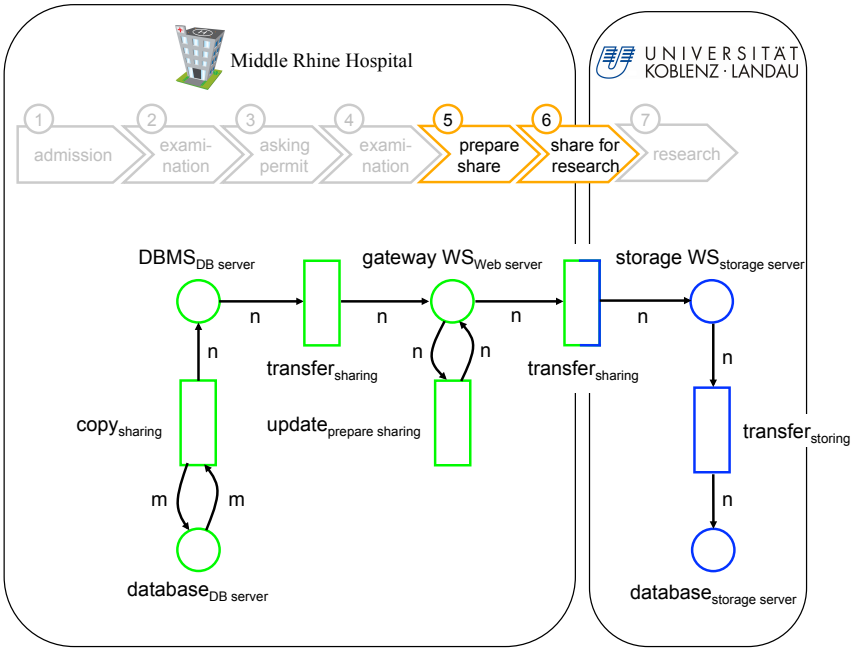

Figure 4.1: Logical Execution.

the data instance is transfered ($transfer_{sharing}$) to the gateway Web service ($gateway\ WS_{Web\ server}$) running on the Web server of the Middle Rhine Hospital. The Web service de-identifies Jane Doe's health record ($update_{prepare\ sharing}$) with the purpose of preparing the sharing of the record with the University of Koblenz. In step 6, the record is transfered ($transfer_{sharing}$) from the gateway Web service of the Middle Rhine Hospital to the storage Web service ($storage\ WS_{storage\ server}$) running on the storage server of the Univeristy of Koblenz. To store the health record it is transfered ($transfer_{storing}$) to the database ($database_{storage\ server}$) pro-

vided by the storage server.

4.1.3 Physical Execution

Global models may exist in controlled environments (e.g. within one organization). If the global model is given, the actions performed on the data during the execution can be monitored and audited by means of this model (cf. [van der Aalst et al., 2008]). The global model supports the monitoring of the data processing by predicting the single steps of the processing and their order. The auditing is supported by the global model by providing a formal model that can be used to verify the execution of a process.

The lack of process awareness of organizations as well as the recurring processing of data items hamper the creation of a global model (cf. regarding Issues in Section 3.4.1). The creation of a global model becomes even harder considering the distribution of processes between various organizations, including the resulting *Issue of Distributed Process Ownership*.

A lot of information about the processing of the data may be lacking *a priori* when the data is created and the processing starts. One reason is the dynamic combination of workflows, as described by the *Issue of Loosely-coupled Architectures*. Later, the processing of data items may continue after the execution of all current and planned processes have ended (see *Issue of Loosely-coupled Architectures*). Hence, in a dynamic and distributed environment, it may be impossible to specify a global model.

In a Web environment, data is processed by Web services as well as other Web applications or Web pages (e.g. Web forms). As part of the workflow execution, entities (services, applications, etc.) call operations of other entities. We call this the *physical execution* of the workflow (see Figure 4.2 for an example). Observing the processing of one specific data item during the physical execution leads to a call history of all operations and methods that have processed the data. The observation delivers traces of the data identifying the organizations involved in the processing.

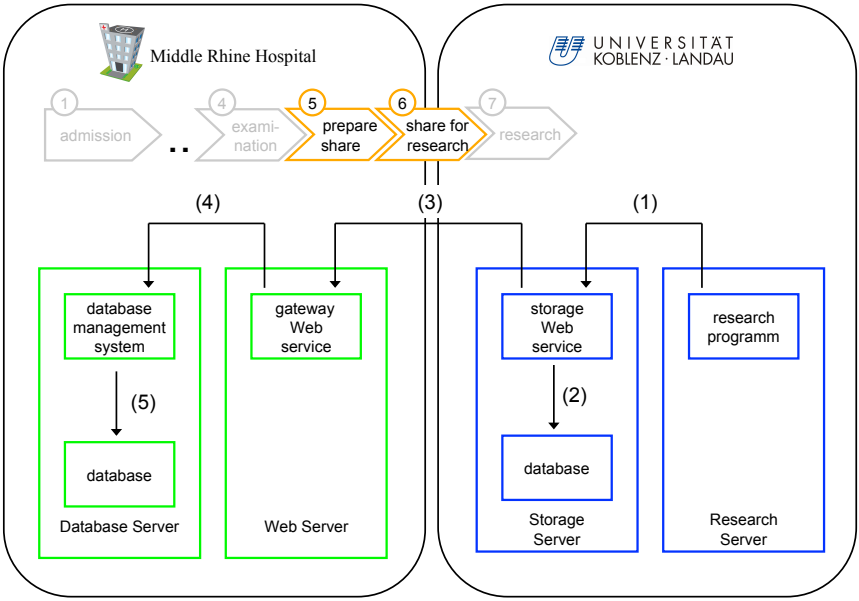

Figure 4.2: Physical Execution.

Example 4.2: *Example of a Physical Execution*

Figure 4.2 depicts the physical execution of the fifth and sixth processing step of our health care scenario. The figure shows the technical details of the snippet of the processing that is also depicted as logical execution in Figure 4.1.

The sharing of the health records starts at the University with the research server requesting new health records from the storage server via the Web service method `getNewHealthRecords` (call **(1)**). The Web service queries the data base running on the same server by means of the database API (call **(2)**). If no new record is found, the Web service calls the

`requestSharingOfHealthRecords` (call **(3)**) operator of the gateway Web service provided by the Web server of the Middle Rhine Hospital. After confirming the validity of the request, the gateway Web service forwards the request to the database server of the Middle Rhine Hospital using a proprietary API provided by the database management system (call **(4)**). The API reads the new health records from the database (call **(5)**) and returns them to the gateway Web service as part of the answer of the API call (call **(4)**). The gateway Web service prepares the health record for the transfer and uses the SOAP response message of the request of the storage server of the University of Koblenz (call **(3)**) to transfer the health records from the hospital to the University. Finally, the health records are stored in the database and transfered to the research server as part of the response of the SOAP call of the `getNewHealthRecords` operator (call **(1)**).

4.1.4 Executed Subsystem

The physical execution implements the logical execution, if given. Without logical execution, the physical execution can be carried out as long as each actor has a local model of the execution. The local models constitute snippets of the logical execution. Executing all snippets leads to the physical execution. Hence, the physical execution will define the logical execution, if the logical execution is not given. Concluding we can state that trough the organizational and technical issues, aggravated by dynamic processes, it may happen that none of the participating organizations and individuals will neither know the complete logical nor the complete physical execution of the process.

Processes may contain alternative execution paths and the actual path may be selected at runtime. Thus, when the process execution is enacted, only a subsystem of the logical execution may be involved in the actual execution or in the processing of an observed piece of data. We call this subsystem the *executed subsystem* of the logical execution (see Figure 4.3).

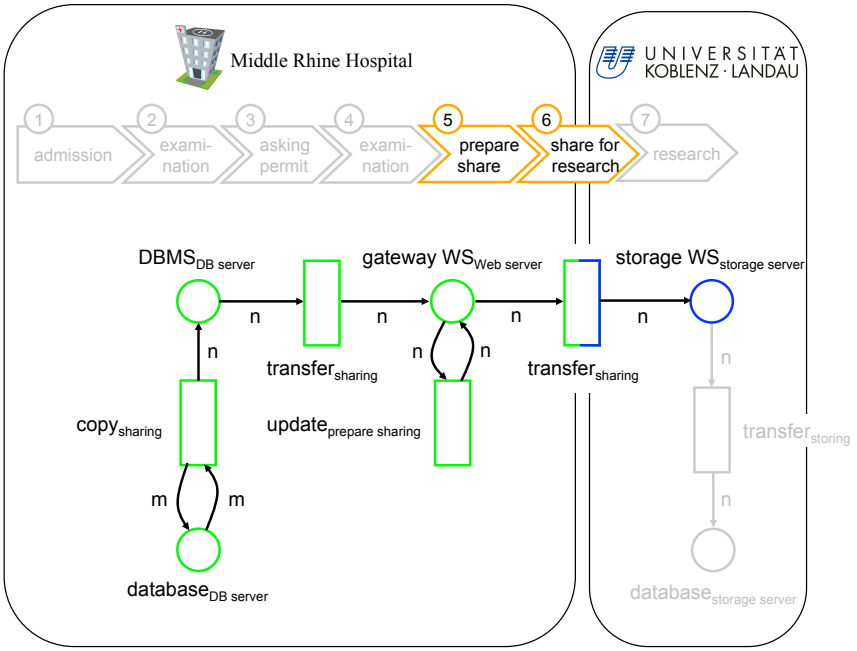

Figure 4.3: Executed Subsystem of a Logical Execution.

Example 4.3: *Example of an Executed Subsystem*

Figure 4.3 depicts the example of Figure 4.1. In contrast to the example of Figure 4.1 now only a subsystem of the process is executed.

A patient demands that her record must not be stored in a database of the University of Koblenz. Thus, the storage server of the University of Koblenz does not store the health record in its database.

4.1.5 Monitored Execution

If the logical execution is unknown, its executed subsystem may still be reconstructed by observing the physical execution of the process. Therefore the physical execution have to be monitored including the actions the actors perform on the data The monitoring leads to a log that specifies a *trace* (see Definiton 4.9) of the execution. We call this log the *monitored execution* of the workflow (see Figure 4.4). In Chapter 5, we introduce a logging mechanism capable to log the processing of pieces of data in distributed environments. If such a mechanism is not used and no continuous log generated, the multiple distributed logs must be integrated to gain a combined monitored execution. However, different runtime environments and logging standards hamper the integration of different logs into one combined log.

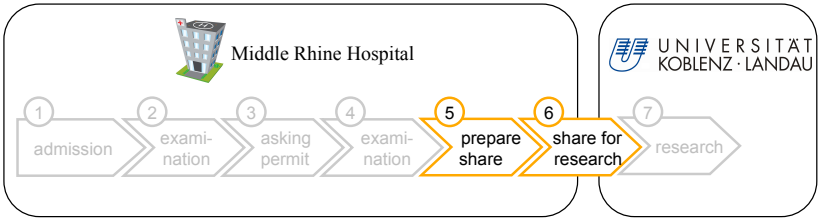

Figure 4.4: Monitored Execution.

Example 4.4: *Example of a Monitored Execution*

Figure 4.4 shows a monitored execution of the same snippet of our scenario depicted by Example 4.1. The shown monitored execution makes use of the algebraic syntax (see Definition 5.2) of the sticky logging mechanism, which we introduce in Chapter 5. Each entry consists of a tuple $(data, logentry)$, where $logentry$ is a sextuple consisting of the category of the performed action, the actor, the entity receiving a transfered data instance, the purpose of the action, the identifier of the action and the identifier of the preceding action. The entries are numbered ($i + 0$ *to* $i + 4$). As a copy is created in $i + 0$, the example depicts the monitored execution of the processing of two instances of the data item ($record\_JD_{original}$ and $record\_JD_{copy\_1}$).
In the example of Figure 4.4, the first entry depicts that the database management system ($databasemanagementsystem_{MRH}$) running on the database server hosted by the Middle Rhine Hospital creates a copy ($record\_JD_{copy\_1}$) of Jane Doe's health record ($record\_JD_{original}$) from the database ($database_{MRH}$). The copy is created at the database server itself. The purpose for copying the data instance is the sharing of the instance for research purposes ($sharing\ for\ research$). The identifier of this action is t_0 and the identifier of the preceding actions is t_{-1}.

4.1.6 Reconstructed Execution

A transition system modeling the executed path of the logical execution can be derived from a monitored execution created by a monitoring mechanism, which is capable to observe all required information (see Chapter 5). We call this transition system the *reconstructed execution* (see Figure 4.5).

Example 4.5: *Example of a Reconstructed Execution*

In Figure 4.5 we depict a reconstructed execution of the executed subsystem (see Figure 4.3) of the logical execution shown in Figure 4.1. We use a labeled state-transition system to represent the reconstructed execution.

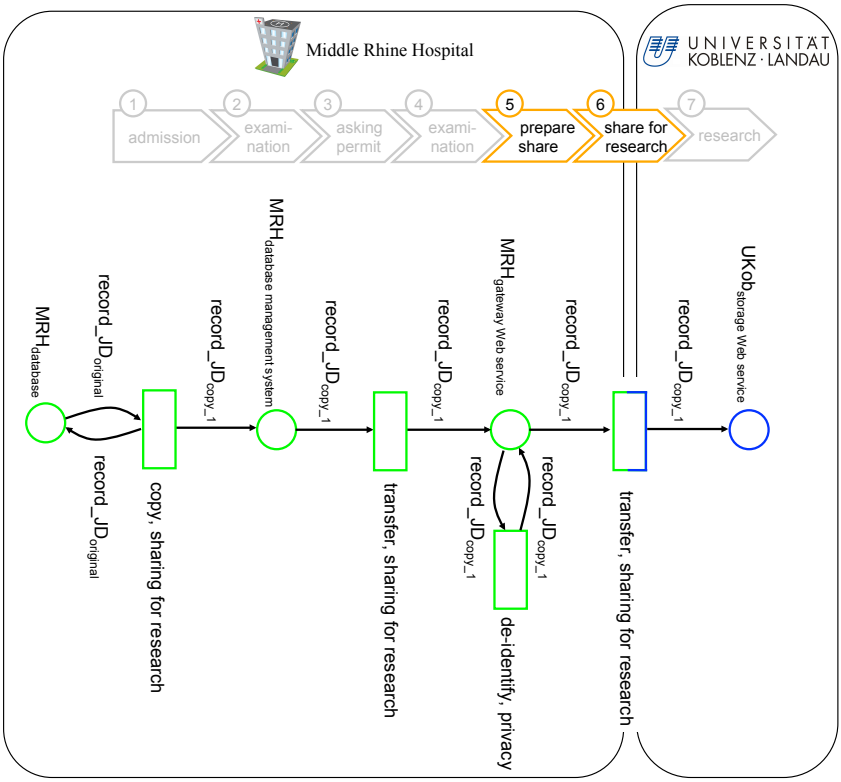

Figure 4.5: Reconstructed Execution.

In difference to the logical execution (see Figure 4.1) the variables m and n have been replaced by the specific data instance $record\_JD_{original}$ and $record\_JD_{copy\_1}$. The labels of the transitions have been derived from the category and purpose specified by the monitored execution. The labels of the places have been taken from the entities specified by the monitored exe-

cution.

4.1.7 Relations Between Executions

Summarizing, the logical execution is a global model of the execution. It defines the physical execution. If no logical execution is given, performing the physical execution defines the logical execution at runtime. Logging the physical execution leads to the monitored execution. Based on the monitored execution, a model of the execution can be reconstructed resulting in the reconstructed execution. The relationships between the different notions are depicted in Figure 4.6.

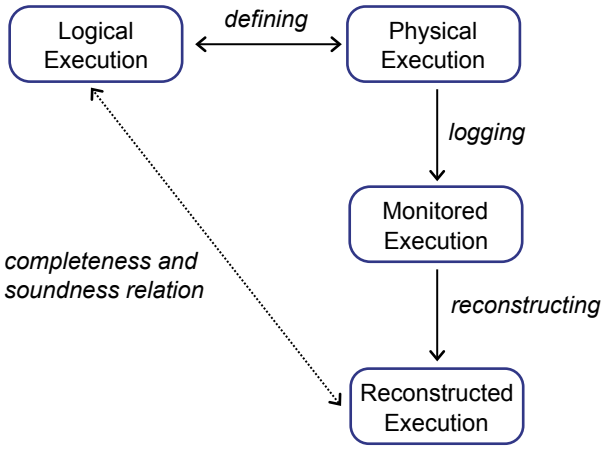

Figure 4.6: Relation Between Executions.

The reconstructed execution could be used for auditing purposes if the behavior specified by the reconstructed execution matches the behavior of the logical execution and if the reconstructed execution models the complete

behavior of the executed subsystem of the logical execution. We name the quality of matching behaviors *soundness* and the quality of complete modeling *completeness*. We require the soundness and completeness of reconstructed executions to assure that statements derived from the reconstructed execution also hold for the logical execution. Before we formalize soundness and completeness by means of a graph morphism in Section 4.3, we introduce the formal definition of DiALog in the following section.

4.2 A Formal Model for Distributed Auditing Logs

In this section, we introduce DiALog, a model of the distributed processing of data in the Web. Defining DiAlog, we consider the *Legal Requirement Exhaustiveness* we discussed in Chapter 3. In the following, we define the structure of DiALog at the atomic level. The structure of DiAlog is designed to formalize the structure of the process executed on the data. We discuss the dynamics of DiALog, which define the sequence of the processing. We define the soundness and completeness of reconstructed executions with respect to logical executions in the following section.

4.2.1 Modeling Data Processing in Distributed Workflows

As a first step of defining DiALog, we have to choose a formalism suitable to model the data processing in distributed workflows. One formalism we can use to model the distributed processing of data items are state-transition system. In state-transition systems, entities can be represented by states, while actions performed on the data are modeled by transitions. Before we define DiALog, we analyze a set of formalisms for state-transition systems such as finite automaton, Petri nets [Peterson, 1981], and colored Petri nets [Jensen, 1992]. We have choose these, as using Petri nets and colored Petri nets to model processes is an existing approach (e.g. [Narayanan and McIlraith, 2002]).

The first formalism we analyze are finite automatons. Using finite automaton we can model entities as states (*Legal Requirement Identifiability*) and processing steps as state changes (*Legal Requirement Exhaustiveness*). However, finite automatons are not adequate for modeling parallel processing of multiple data instances in one automaton. To model the parallel processing we have to use multiple automatons, for each data instance one automaton. As we require to make statements about the distribution of the data instances at a certain point in time (*Legal Requirement Identifiability*), we are required to link the state changes of the multiple automatons to each other. As such a link is not defined for finite automatons we have chosen not to use them.

In Petri nets, we can model entities by means of places, processing steps by means of transitions and data as tokens. By using multiple tokens, we can model multiple data instances in one Petri net, and the distribution of the tokens represents the distribution of data instances in a distributed process. The *Legal Requirement Exhaustiveness* demands an exhaustive answer that clearly identifies what has happened and when. Such an answer will be feasible only if we use unique identifiers for data instances. The unique identification can be modeled by multiple Petri nets, one for each data instance. The multiple Petri nets can be folded into one net. However, the folding decreases the readability.

Instead of folding Petri nets, we can make use of colored Petri nets. Colored Petri nets provide additional syntactical elements (cf. Section 2.2.2), such as different colors (identifiers) for each token (data instance). So, we can use colored Petri nets to model the distributed processing of multiple data instances of one data item in one single net.

4.2.2 DiALog

In DiALog, we model the distributed processing of data items by colored Petri nets. Even if a single colored Petri net can model the processing of all data items of one data category with all data instances, we assume that only the processing of one specific data item is modeled by one colored

Petri. This restriction allows us for using one-dimensional indexes to address data instances. It also improves the readability by a clear identification of processed data instances. The modeling of all data items can be reached by folding the colored Petri nets of these data items into one net and using multi-dimensional indexes.

The atomic level of the defined structure of DiALog is able to model actions at the lower level implementations. The level of granularity of DiALog can be freely chosen and thus allows for modeling on the required abstraction levels. From the legal requirements (cf. Section 3.4) we derive the following properties of distributed systems that require to be modeled:

- **Actions:** To meet the *Legal Requirement Accessibility* the processing of data must be monitored. To monitor the processing, we log all actions performed on the data. Different types of actions can be performed and require different modeling (see below). All together we distinguish the following six types of actions:

$$ActionTypes = \{create, read, update, copy, transfer, delete\}$$

- **Actors:** The *Legal Requirement Identifiability* postulates to log information about the service provider as well as about recipients and sources of data. Thus, we model every actor that performs actions on the data, that receives data, or that passes data on (e.g. a Web service or a database).

- **Data Instance:** The *Legal Requirement Exhaustiveness* demands to log which personal data is processed. During the execution of a process, copy actions may occur that lead to multiple data instances of one data item. Hence, the formalization must model each data instance and all instances must be clearly identifiable.

We define DiALog to model the distributed processing of data instances by means of colored Petri nets (cf. [Jensen, 1992] and Section 2.2.2).

Definition 4.1: *A Model for Distributed Auditing Logs*

The distributed processing of one data item is modeled as a colored Petri net:

$$CPN = (\Sigma, P, T, A, N, C, G, E, I)$$

where:

- *Places* $p \in P$ model **actors**.
- *Transitions* $t \in T$ in combination with
- *arcs* $a_1, a_2, .. \in A$,
- *arc expressions* $E(a_1), E(a_2), ..$, and
- the *node function* of these arcs $N(a_1), N(a_2), ..$ model **actions** (or combinations of actions).
- *Tokens* represent **data instances** and the *color* $c \in \Sigma$ of these tokens is an integer uniquely **identifying** the data instances.
- The value of the *color function* $C(p)$ for the places is the integer type.
- The *initialization* function is $\forall p \in P \setminus p_c : I(p) = \emptyset$ (p_c models the counter place, see below).
- The *guard function* for all transitions $t \in T$ is $G(t) = true$ if not defined otherwise.

Arcs connect the transitions with the places where the actions are performed. The *arc expressions* describe how data instances are moved. The labels of transitions consist of the category of an action and its purpose (written: $category_{purpose}$).

The markings (without the token of the counter; the counter is introduced below) represent the distribution of data instances of the observed data item

in the process. Each data instance receives another, unique value allowing for unique identification of the data instance. By using the integer value of a token as unique identifier of the represented data instance, we are able to clearly distinguish the single instances. Due to the statelessness of places, additional information is needed to create new unique identifiers. To this end, we use the counter that is modeled as part of the colored Petri net. The counter is represented by the place p_c, whose initialization function is specified as: $I(p_c) = \{1\}$

The following list describes how to model the different action types by means of colored Petri nets. Each category of action has to be modeled by a specific combination of places, arcs, node functions and arc expressions as defined below. How to combine the actions to a complex process specification is described in Section 4.2.3. We do not define a specific granularity for modeling the data processing. DiALog allows for modeling at any detail level (cf. *Requirement Level of Granularity*; e.g. a Web server may be modeled as one entity or as several entities (Web services, DBMS, etc.)). Two example models are depicted in Figure 4.13 and Figure 4.14. Both show the same snippet of the processing of Jane Doe's helath record, but with a different granularity.

- **Create action:** The processing of a data item starts with the creation of its first data instance (e.g. in the scenario, the health record is created in step 1). In our model, we represent data instances by tokens and the first token of a data item is generated by the create action. Definition 4.2 specifies the subnet of a colored Petri net that models a create action.

Definition 4.2: *Create action as colored Petri net*

Be $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ a colored Petri net and be $p \in P$ the place representing the entity creating the data instance. We model create actions as:

- a transition $t \in T$ with
- guard function $G(t) = [u = 1]$ and
- an arc $a \in A$ with
- node function $N(a) = (t, p)$ and
- arc expression $E(a) = u$, where u is declared as `var u :` $\mathbb{N}_0$.

Be $p_c \in P$ the place modeling the counter, t is connected to the counter by two arcs $c_i, c_o \in A$ with

- the node functions $N(c_i) = (t, p_c)$ and $N(c_o) = (p_c, t)$, and
- the arc expressions $E(c_i) = u + 1$ and $E(c_o) = u$.

As depicted in Figure 4.7, the create action is defined as a transition t with three arcs (two outgoing and one incoming). One arc a leads from the transition to the place p representing the actor where the data item is created. The arc c_o is connected with the place p_c (the place modeling the counter). The transition t is protected by a guard function $[u = 1]$, which is used to control whether the counter is set to 1. If the counter is set to 1, no instance of the considered data item will have been created before and the transition can still occur. The arc c_i is used to set the counter to 2, which will be the identifier of the first copy. If the counter is set to 2 or higher, there will already be at least one instance of the data item. In this case, the first instance of the data item can not be created anymore and the guard function prevents the enabling of the transition.

- **Copy action:** During the execution of a workflow, additional data instances may be created through copying. Whenever someone accesses the health record stored in the database of the Middle Rhine Hospital, a copy is created (cf. steps 2, 3, 4, and 5 of the scenario). In our model, each copy action generates a new data instance represented by

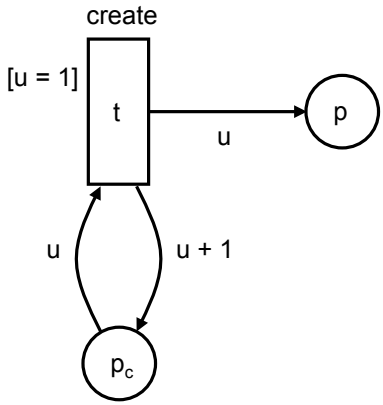

Figure 4.7: Creating the first data instances.

an additional token. Definition 4.3 defines how to model a copy action by means of colored Petri net. The colored Petri net is depicted in Figure 4.8.

Definition 4.3: *Copy action as colored Petri net*

Be $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ a colored Petri net, be $p_s \in P$ the place representing the entity copying the data instance, and be $p_d \in P$ the place representing the entity where the new token is created. We model copy actions as:

- a transition $t \in T$ and
- three arcs $a_i, a_o, a_d \in A$ with
- node functions $N(a_i) = (p_s, t)$, $N(a_o) = (t, p_s)$, and $N(a_d) = (t, p_d)$ and with

- arc expressions $E(a_i) = u$ and $E(a_o) = u$, where u is declared as `var u : ` $\mathbb{N}_0$, and $E(a_d) = v$, where v is declared as `var v : ` $\mathbb{N}_0$.

Be $p_c \in P$ the place modeling the counter, t is connected to the counter by two arcs $c_i, c_o \in A$ with

- the node functions $N(c_i) = (t, p_c)$ and $N(c_o) = (p_c, t)$, and
- the arc expressions $E(c_i) = v + 1$ and $E(c_o) = v$, where v is the same variable as in the arc expression $E(a_d)$.

As depicted in Figure 4.8 we define the copy action as a transition t with five arcs. Two arcs a_i and a_o lead to and from the place p_s that represents the actor performing the copy action. One arc a_d leads from the transition to the place p_d representing the actor where the new data instance is created. The Actor performing the copy action (p_s) and the actor where the copy is created (p_d) may be different places but can also be the same place. Likewise to the create action, the additional two arcs c_i and c_o are connected with the place p_c used to count the creations of data instances. The current value of the counter is used as identifier of the new data instance. After the new instance is created the counter is increased. As the transition has an incoming arc (a_i), a data instance must exist before the copy can be created. Hence, it is not necessary to checked whether the counter token has a value greater 1.

- **Read action:** Reading information from a data instance changes neither its content nor its location (e.g. the health record of Jane Doe is read by a researcher of the University of Koblenz in step 7 of the scenario). The modeling of the read action must respect these circumstances. We model read actions in DiALog as specified in Definition 4.4.

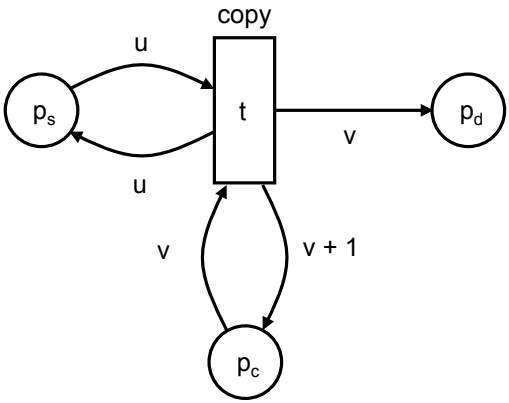

Figure 4.8: Copying of data instances.

Definition 4.4: *Read action as colored Petri net*

Be $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ a colored Petri net and be $p \in P$ the place representing the entity reading the data instance. We model read actions as:

- a transition $t \in T$ and
- two arcs $a_i, a_o \in A$ with
- node functions $N(a_i) = (p, t)$ and $N(a_o) = (t, p)$ and
- arc expressions $E(a_i) = u$ and $E(a_o) = u$, where u is declared as `var u : ` $\mathbb{N}_0$.

Figure 4.9 depicts the modeling of a read action as a transition t with two arcs a_i and a_o leading to and from the place p that represents the

actor performing the read action.

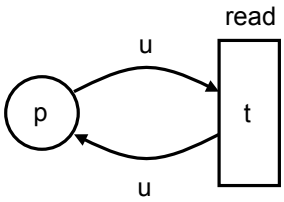

Figure 4.9: Reading data instances.

- **Update action:** Updating a data instance changes its content, as depicted in step 5 of the scenario where the health record is de-identified before it is shared. Performing an update action does not change the location of the data instance. The presented formalization does not distinguish whether the content of a data instance has been changed or not (such information can be modeled by means of versioning systems like SVN [1]). Hence, the formal representation of update actions is the same as of read actions. Definition 4.5 defines how to model an update action by means of colored Petri nets in DiALog.

Definition 4.5: *Update action as colored Petri net*

Be $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ a colored Petri net and be $p \in P$ the place representing the entity updating the data instance. We model update actions as:

- a transition $t \in T$ and
- two arcs $a_i, a_o \in A$ with

[1] http://subversion.tigris.org/, retrieved Feb 8th, 2011

- node functions $N(a_i) = (p, t)$ and $N(a_o) = (t, p)$ and
- arc expressions $E(a_i) = u$ and $E(a_o) = u$, where u is declared as `var u : ` $\mathbb{N}_0$.

Analogously to the read action, we model the update action as a transition t with two arcs a_i and a_o leading to and from the place p that represents the actor performing the update action (see Figure 4.10).

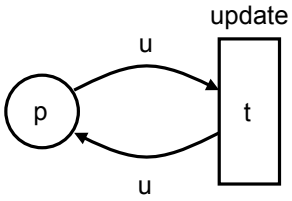

Figure 4.10: Updating data instances.

- **Delete action:** The processing of a data instance ends with its deletion. In the scenario, the University of Koblenz deletes its copy of Jane Doe's health record after finishing its research. The copies of the Middle Rhine Hospital still exist and thus the data item. The token representing the data instance is removed, when the data instance is deleted.

Definition 4.6: *Delete action as colored Petri net*

Be $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ a colored Petri net and be $p \in P$ the place representing the entity deleting the data instance. We model delete actions as:

- a transition $t \in T$ and

- an arc $a \in A$ with
- node function $N(a) = (p, t)$ and
- arc expressions $E(a) = u$, where u is declared as `var u :` $\mathbb{N}_0$.

As depicted in Figure 4.11 we model the delete action as one arc a leading from the place p (the actor deleting the data instance) to a transition t that has no outgoing arc. As it has no outgoing arc, the transition t 'consumes' the token. Deleting one data instance does not imply the deletion of all instances of a data item. A data item exists as long as one data instance of the data item exists. A data item will be deleted as soon as all instances have been deleted.

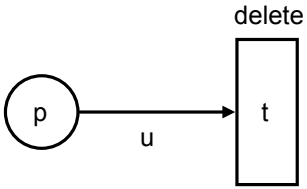

Figure 4.11: Deleting data instances.

- **Transfer action:** Whenever a data instance is transferred from one actor to another, the associated token must also be transferred. The scenario depicts many kinds of transfers between different systems (e.g. the database server and the workstation in step 1), between different applications (e.g. the database API and the database management system in step 1), and between different actors (e.g. the Middle Rhine Hospital and the University of Koblenz in step 6). Definition 4.7 defines how to model a transfer action in DiALog.

Definition 4.7: *Transfer action as colored Petri net*

Be $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ a colored Petri net, be $p_s \in P$ the place representing the entity sending the data instance, and be $p_r \in P$ the place representing the entity receiving the data instance. We model transfer actions as:

- a transitions $t \in T$ and
- two arcs $a_s, a_r \in A$ with
- the node functions $N(a_s) = (p_s, t)$ and $N(a_r) = (t, p_r)$ and
- the arc expressions $E(a_s) = u$ and $E(a_r) = u$, where u is declared as `var u : ` $\mathbb{N}_0$.

As depicted in Figure 4.12, we model the transfer action as a transition t with two arcs a_s and a_r. The arc a_s leads from the place p_s to the transition t. The place p_s represents the actor sending the data instance. The second arc a_r leads from the transition to the place p_r, which represents the receiving entity. During the transfer the data instance and thus the value of the token remains unchanged.

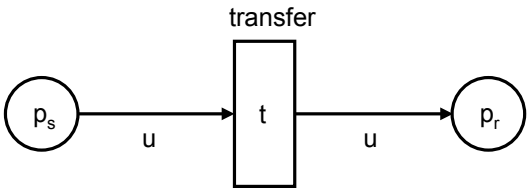

Figure 4.12: Transferring data instances.

4.2.3 Building Rules

To model the logical execution, the above actions are combined by merging the subnets defining the single actions to one combined colored Petri net. The combination must adhere to the following rules:

- *Only the above defined subnets may be used and no additional elements (e.g. transitions) are allowed.*

  Using colored Petri nets to model complex processes, one has more possibilities than using the elements defined by DiALog. To adhere to the requirements defined in Chapter 3, the model of the distributed processing of data must hold certain qualities, such as the clear identification of processing steps, of entities and of the processed data (cf. *Legal Requirement Exhaustiveness* and *Legal Requirement Identifiability*). Adding arbitrary elements of colored Petri nets may lead to various mistakes, e.g. by adding additional transitions and arcs, identifiers may be changed or copies of data instances may be created, which do not have unique identifiers. Using only the pre-defined elements of DiALog guarantees that the requirements are met and no mistakes are made.

- *The subnets are connected by means of the places that model the different involved entities (not the counter place).*

  By modeling entities as places in DiAlog, we can reconstruct the distribution of the data instances. As actions can only be performed by actors possessing the data, the transitions modeling these actions must be connected to the places modeling the performing actors (cf. *Legal Requirement Identifiability*).

- *Each model contains only one counter, which is shared by the create action and all copy actions, for generating unique data instance.*

  As the identifier used in our model must be unique for clear identification (see the *Legal Requirement Exhaustiveness*), the counter place

needs to generate such unique identifiers. If there is more than one counter, there will be no guarantee for the generation of unique identifiers. Hence, only one counter is allowed, which is not only connected with the create action but also with all copy actions.

Example 4.6: *Different Models of the Health Care Scenario*

During the admission of Jane Doe her health record is created by a member of the medical staff (`create action`) at a workstation of the Middle Rhine Hospital. The purpose of the creation of the health record is the treatment of Jane Doe. After the creation the health record is shown to a staff member (`read action`) who enters the personal information of Jane Doe into the record (`update action`). The first step of the processing ends with storing of the health record. To this end, the health record is first transfered to the server component of the administration software running on the Administration Server (`transfer action`). As the administration software has no direct access to the database, the software transfers the record to the API provided by the Database server to store the record in the database (`transfer action`). The database API itself access directly the database (`transfer action`) where the record is stored. The purpose of the transfer actions is the storing of the health record. Figure 4.13 shows a detailed model of the complete processing of Jane Doe's health record as described in Section 3.1 (to increase readability the figure does not contain the counter place). Figure 4.14 depicts the same process with another level of granularity where the Middle Rhine Hospital and the University of Koblenz are modeled as a single entity each.

4.2.4 Interaction between Data Instances

Each colored Petri net models the processing of exactly one data item. The processing of a data instances may interact with the processing of other data

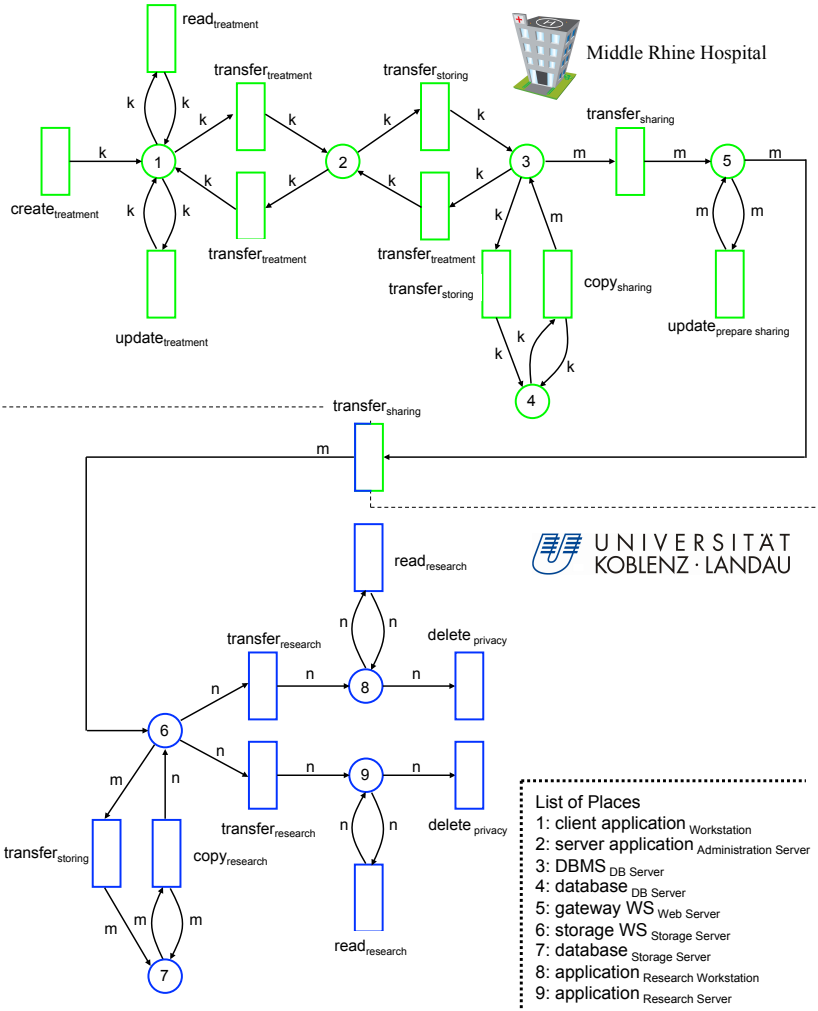

Figure 4.13: DiALog model of the processing of Jane Doe's health record.

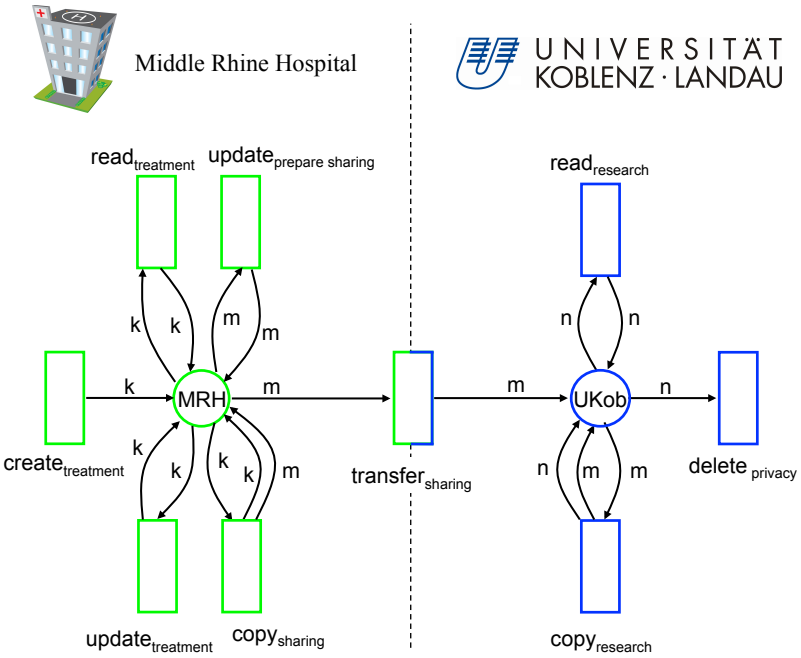

Figure 4.14: Less detailed model of the processing of Jane Doe's health record.

instances, also with data instances of other data items. Different kinds of interactions can be distinguished, depending on the effect of the interaction on the involved data instances:

1. *The interaction has no effect on the involved data instances.* For instance, reading two health records to compare their content as part of the research performed by the University of Koblenz has no effect on the health records. In this case, no special modeling is required. Each action (e.g. read action) is modeled in the colored Petri net of the data

instance the action is performed on.

2. *The interaction leads to the creation, update, or deletion of one or more of the involved data instances.* For instance, Jane Doe's health record is checked regarding as to her dietary regiment. So that, the hospital's food service updates its list of meals to cook. The modeling of the interaction is implicit as the processing of the involved data instances are modeled in the colored Petri net of the corresponding data items (in the case of a create action, a new colored Petri net must be created). The connection between both processes may be modeled only by the purposes of the single actions.

3. *The interaction leads to the merger of two (or more) data instances.* For instance, for research purposes the health records of different patients are aggregated to one big data set. The modeling is done implicitly. Merging of data instances leads to the deletion of the instances and the creation of a new, additional data instance which is the result of the merge. If data instances are merged into a persistent data instance, the persistent data instance will be updated while the other ones will be deleted.

As in all three cases the modeling of the interaction is done implicitly, some sorts of annotations (e.g. by the label of the transitions modeling the actions) can be used to represent the connection between the processing steps. However, the connections are not modeled explicitly as they do not influence the processing of the single data items.

4.2.5 Representation of Data Processing

In colored Petri nets a marking is the distribution of tokens in the net. In DiALog, the markings represent the distribution of instances of one data item. Each marking represents one state of an execution of a distributed workflow. The occurrence of a transition leads from one state of the colored

Petri net to another. Because transitions with their arcs are used to model actions, the occurrence of a transition (also defined as step, cf. Definition 2.6) represents an action. And the change of the marking represent the change of the distribution of data instances.

Definition 4.8: *Process Execution*

We define the execution of a process as the occurrence of steps Y^* in the colored Petri net.

The whole processing is a partially ordered set of steps due to parallel paths of the workflow. We define such a partial order of steps as a trace.

Definition 4.9: *Processing Trace*

Be W a (distributed) workflow modeled by means of a colored Petri net, we define a trace t of the processing of a data item as a partial order of occurring steps Y^*.

The monitored execution is a processing trace of the logical execution.

4.3 Qualities of Execution Models

In Section 4.1, we have indicated that a reconstructed execution can be used for auditing purposes instead of a global model. To this end, the reconstructed execution must be complete and sound with respect to the actually executed subsystem of the logical execution. Both, the logical execution as well as the reconstructed execution are labeled transition systems. We define a labeled transition system as common:

Definition 4.10: *Labeled Transition Systems*

A labeled transition system is a triple

$$(P, \Delta, \rightarrow)$$

where P is the set of places, Δ is the set of labels, and $\rightarrow \subseteq P \times \Delta \times P$ is the transition relation.

As only parts of the logical execution may be executed, we require to consider the executed subsystem for the definition of soundness. We define a subsystem of a labeled transition system as follows:

Definition 4.11: *Subsystems of Labeled Transition Systems:*

We define a subsystem of a labeled transition system $(P_T, \Delta_T, \rightarrow_T)$ as a triple

$$(P_S, \Delta_S, \rightarrow_S)$$

where $P_S \subseteq P_T$, $\Delta_S \subseteq \Delta_T$, and $\rightarrow_S \subseteq \rightarrow_T$ with the following stability property: $(x, \delta, x') \in \rightarrow_S$ with $x, x' \in P_T$ and $\delta \in \Delta_T$ implies $x, x' \in P_S$ and $\delta \in \Delta_S$.

Based on the definition of a labeled transition system and its subsystems we are now able to define the soundness and completeness of reconstructed executions.

4.3.1 Soundness Quality

We define soundness of a reconstructed execution to express that the behavior of the reconstructed execution matches the behavior of the logical execution. The behavior of the reconstructed execution will match if a simulation relation between the reconstructed execution and the logical execution exists.

A simulation is a binary relation defining matching behavior of transition systems (cf. [Kucera and Mayr, 1999]). A first transition system will simulate a second transition system if the first system can match all of the state changes of the second system.

Definition 4.12: *Soundness of Reconstructed Executions*

Given a logical execution L and a monitored execution M, a reconstructed execution R generated from M is sound with respect to L, if their exists a simulation Φ so that for all elements $r \in R$ there exists an element $l \in L$ so that $(r, l) \in \Phi$.

Because the reconstructed execution should match the behavior of the executed subsystem of the logical execution, for each element r of the reconstructed execution an element l of the executed subsystem is required so that (r, l) is an element of the simulation relation. Figure 4.15 depicts the soundness relation between a logical execution and the reconstructed execution. The dashed arrows between the elements of the reconstructed execution and the logical execution depict the simulation relation between the two models.

4.3.2 Completeness Quality

We define completeness of a reconstructed execution to express that the reconstructed execution models the complete behavior of the executed subsystem of the logical execution. The reconstructed execution will model the complete behavior, if a simulation relation between the executed subsystem of the logical execution and the reconstructed execution exists.

Definition 4.13: *Completeness of Reconstructed Executions*

Given a logical execution L and a monitored execution M, a reconstructed execution R generated from M is complete with respect to the executed subsystem S of L, if their exists a simulation Φ so that for all elements

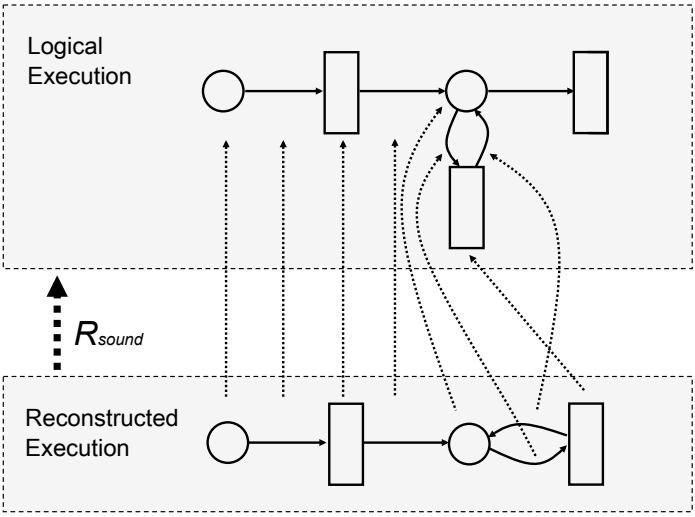

Figure 4.15: Soundness of a Reconstructed Execution.

$s \in S$ there exists an element $r \in R$ so that $(s, r) \in \Phi$.

For each element s of the executed subsystem an element r of the reconstructed execution is required so that (s, r) is an element of the simulation relation. Figure 4.15 depicts the completeness relation between an executed subsystem of a logical execution and the reconstructed execution. The dashed arrows between the elements depict the simulation relation between the executed subsystem of the logical execution and the reconstructed execution.

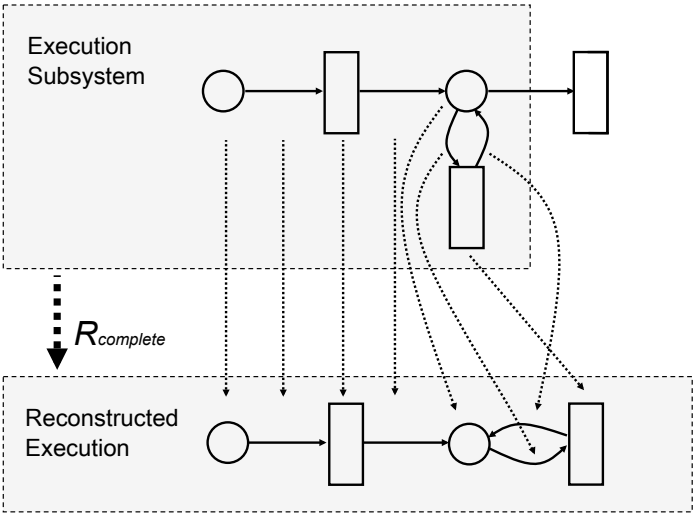

Figure 4.16: Completeness of a Reconstructed Execution.

4.3.3 Generation of Sound and Complete Reconstructed Executions

At the beginning of this chapter we have indicated that reconstructed executions are generated after the execution of the workflow and are used for auditing purposes instead of the global model. Hence, it is a requirement for the utilized monitoring mechanism to generate sound and complete reconstructed executions. We discuss this for the sticky logging mechanism in Section 5.2.

4.4 Related Work

Other work identifying the need for logging data usage in distributed environments is presented in [Weitzner et al., 2008]. The authors analyze that

access restrictions are not sufficient to achieve policy goals, because in many environments it cannot be guaranteed that a specific agent has no access to a certain piece of information. The authors demand transparency of information usage to enable accountability, but they do not present a solution.

In [Barth et al., 2007], the authors present a logic to specify and verify privacy and utility goals of business processes in non-distributed environments. The approach does not observe the processing of specific data items in distributed environments. Instead, they observe only the data communication by one single agent.

In the area of process mining, Petri nets are used to represent workflows. The approach presented in [van der Aalst, 1998] models processes as activity flows. This approach is extended by the work of [van der Aalst et al., 2004] which additionally models the control flow including data dependencies. As these approaches are designed to model activity and control flows, they are not designed to represent the distributed processing of specific data items.

In [van der Aalst et al., 2008] the authors present an approach to use message logs and BPEL specifications [Jordan et al., 2007] for conformance checking of service behavior. In difference to our work, they assume that a global model of the observed services is given by the BPEL specification. The authors of [Aldeco-Perez and Moreau, 2008] and [Aldeco-Pérez and Moreau, 2010] present an architecture for auditing private data in IT-Systems by collecting provenance information. Their architecture is also based on requirements specified by privacy laws, but does not provide a formal specification of the data processing. Compliance of business process models with object life cycles is addressed in [Küster et al., 2007]. The introduced approach provides a formal method to generate process models that are compliant with reference object life cycles. This approach does not provide a model of the distributed data processing.

The authors of [Cederquist et al., 2005] propose to use an auditing mechanism to achieve accountability in distributed environments. The auditing is done based on policies and logged actions, conditions and obligations. The logs are assigned to agents performing the actions. Neither a mechanism is

provided to make the logs accessible to the service customer nor they provide a formal model of the data processing.

To specify processes, we can use not only Petri nets, but different sorts of formalisms. Process calculi and process algebras are a family of approaches providing a formal semantics to specify processes and concurrent systems. The work presented in [Bekić, 1984] is one of the first works introducing a notion of action that can be used to specify the parallel execution of processes. The calculus of communicating systems (CSS) [Milner, 1980], the algebra of communicating processes (ACP) [Bergstra and Klop, 1984] and the calculus of mobile processes (π-calculus) [Milner et al., 1992] are three widespread extensions. Instead of colored Petri nets, we could also have used a process algebra to specify DiALog.

The Unified Modeling Language [Object Management Group, 2010] supports various sorts of charts to model workflows, such as activity charts and state charts. UML differs from Petri nets and process algebras by providing a graphical notation without formal semantics. A graphical specification of processes is supported by the Business Process Modeling Notation (BPMN) [Object Management Group, 2011]. BPMN is used to design business processes and can be translated to BPEL specifications. The Business Process Execution Language (BPEL) [Jordan et al., 2007] provides an execution language for Web services in business processes.

A different approach to distributed data processing is presented by the authors of [Ludäscher et al., 2006]. They introduce Kepler, which is a scientific workflow system for grids. The users are able to manage the data processing by this system. Kepler does not consider multiple responsible entities and that data may be processed in other workflows than originally planned.

4.5 Summary

To be able to audit the processing of a data item in a distributed environment, a global model of the execution is required. As such a global model

is most often missing, we need a formal method enabling us to still audit the distributed processing of data. This method has to be data-centric to be able to trace the processing of a data item independent of business processes, applications and performed actions. It has to support distributed environments to trace the processing across different execution environments and among multiple organizations. Existing methods do either not consider distributed environments or they do focus on agents or communication but not on general processing of single data items and all their instances. In this chapter, we have introduced DiALog, Distributed Auditing Logs. With DiALog, we provide a mean to audit the processing of data items and their instances in distributed environments.

DiALog is a formal model based on colored Petri nets. Before we have specified DiALog, we defined different execution models and specified the relations between the single models. With DiALog, we specify elements to be used to model the logical execution as well as the reconstructed execution of the data processing.

As a global model and the model of the logical execution is lacking in distributed environments, we defined the soundness and completeness of the derivation of the reconstructed execution. By means of these qualities, we can verify whether a logging mechanism can generate reconstructed executions that can be used for auditing purposes instead of the missing global model. In the following chapter, we introduce sticky logging, a logging mechanism able to generate reconstructed executions. We also verify that it fulfills the soundness and completeness qualities.

We confirm Hypothesis 1 through DiALog as it provides a formal model for distributed data processing based on which we define the soundness and completeness qualities of reconstructed executions. We have published the presented work at the International Conference on Web Services [Ringelstein and Staab, 2009] and in the Journal of Web Service Research [Ringelstein and Staab, 2010a].

5 Monitoring the Distributed Processing of Data

DiALog serves as a formal method for auditing the distributed processing of data. To generate a monitored execution that can serve as input to generate sound and complete reconstructed executions, a monitoring mechanisms is required that overcomes the organizational and technical issues we identified in Chapter 3. These issues are the *Issue of Loosely-coupled Architectures*, the *Issue of Lack of Process Awareness*, the *Issue of Recurring Processing*, and the *Issue of Distributed Process Ownership*. To collect the needed provenance information, we require a distributed mechanism for logging in distributed environments. Existing logging mechanisms, such as the Extended Log File Format [Hallam-Baker and Behlendorf, 1996] or syslog [Lonvick, 2001], are not sufficient to gain a full overview of a workflow that is distributed among multiple organizations. The main causes are the logging in single execution environments, the diversity of execution environments and the lack of standardized interfaces for exchanging logs. Because of these, aggregating distributed logs remains a challenge. Our hypothesis is that a distributed, data-centric logging mechanism is able to collect the relevant information and to provide a sound and complete reconstruction of the processing.

In the following, we present sticky logging. Sticky logging monitors the processing of data items (independent of the actual business process) attaching the logs directly to the processed data as metadata. Furthermore, sticky logging allows for the reconstruction of how the data was processed by whom and why. The sticky logging mechanism defines a mathematical

structure, it specifies a set of logging operations and algorithms based on this structure, and it defines a logging ontology.

The basic idea of sticky logging is to attach the log directly to the data as metadata. Thereby, the log is transferred together with the data along the processing path whenever the data is transferred. In a service-oriented architecture, the log is passed as part of a service call. To be able to observe the data processing and to manage the passing of logs, the sticky logging mechanism is a layer between the execution environment (e.g. JBoss) and business software (e.g. Web services). The log is returned to the service consumer after the processing making the log accessible to the person concerned.

To introduce the sticky logging mechanism, we discuss the mathematical structure of its logs, its several operations describing how to log as well as when a log needs to be transferred to whom, and its method to generate the reconstructed execution in Section 5.1. In Section 5.2, we prove that sticky logging can be used for the generation of sound and complete reconstructed executions. We specify the sticky logging ontology and give a description of a prototype implementation of the sticky logging middleware in Section 5.3. In this section, we also sketch how the sticky logging mechanism is used to inform the person concerned and we discuss a mechanism for restricting the access to the logs.

5.1 A Formal Method for Sticky Logging

The formal method of the sticky logging mechanism consists of three parts. A data structure specifies sticky logs including log entries, an algorithm describes how to monitor the execution and another algorithm defines how to reconstruct a DiALog model from a sticky log.

5.1.1 The Data Structure of Sticky Logs

Fundamental for sticky logging is that a log Λ is attached to the corresponding data instance d. Each log is connected with a token that represents a data

instance in DiALog, as specified in Chapter 4. A log is a partially ordered set of log entries λ. The log entries are used to record the performed actions on d. In DiAlog, the performance of an action is represented by the occurrence of the transitions representing the actions. The data about an action consists of the category χ of the action, the performing actor α (i.e. the corresponding place), the receiving actor β of a transfer action ($\beta = \epsilon$, if it is not a transfer or copy action), and the set of purposes Ψ of the action. To achieve the partial order of the actions, a unique identifier id is assigned to each action and the preceding action is linked by its identifier pid. We define a log entry of a sticky log by means of the following data structure:

Definition 5.1: *Data Structure of Log Entries of Sticky Logs*

A log entry λ is a sextuple $(\chi, \alpha, \beta, \Psi, id, pid)$ where:

- χ is the category of the action,
- α is the performing actor,
- β is the actor receiving a transferred or copied data instance,
- Ψ is a set of purposes of the performance,
- id is the unique identifier of the action, and
- pid is the unique identifier of the preceding action.

Using this definition we define a sticky log by the following data structure:

Definition 5.2: *Data Structure of Sticky Logs:*

A sticky log m is a tuple (d, Λ) where:

- d is a data instance and

- Λ is the set of all log entries λ that are related to d.

5.1.2 Logging the Execution

To log the processing, the logging mechanism needs to perform certain operations whenever a transition occurs (see Table 5.1). If the transition represents a create or copy action, a new log will be created. If the transition models a transfer action, the log will be transferred together with the data instance.

The processing of an instance ends with its deletion, without deleting the sticky log. Instead the log is returned to the actor with the source data instance. The actor and the source data instance are identified by references specified during the copying of the data. The returned log is merged with the log of the source instance. After merging the logs, all references contained in logs of copies of the deleted data instance have to be updated to refer to the merged sticky log. If the deleted instance is the last instance of a specific data item, the log will be returned directly to the person or organization that initially created the first instance of the data. This person or organization is responsible for answering information requests by the person concerned.

The occurrence of any action requires the extension of the log by a log entry monitoring the action (see Table 5.1). In this section, we present a mathematical operation defining the creation of the log entries and the updates to the set of all sticky logs when an action occurs. The set of all sticky logs contains the logs associated with all instances of the observed data item. A log may contain the processing of more than one data instance, such as the result of a deletion of an instance and the merge of the log with the log of the origin data instance. Listing 5.1 depicts the logging operation describing the capability of the sticky logging mechanism.

The input of the operation is the set of all sticky logs M associated with the observed data item, the occurred transition t, the arc expressions $E(a_i)$

Table 5.1: Operations of the Sticky Logging Mechanism.

Performed Action	Logging Operation
Creation of the first data instance d.	1) Creating a new sticky log Λ_d. 2) Attaching the log Λ_d to the created data instance d. 3) Creating a log entry $\lambda_{d,1}$ about the creation of the data instance. 4) Adding the log entry $\lambda_{d,1}$ to the sticky log Λ_d.
Creating a copy c of d.	1) Creating a log entry $\lambda_{d,n}$ about the copying of the data instance. 2) Creating a new sticky log Λ_c. 3) Attaching the log Λ_c to the created data instance c. 4) Creating a log entry $\lambda_{c,1}$ about the creation of the copy c. 5) Referring the log entries $\lambda_{d,n}$ and $\lambda_{c,1}$ to each other. 6) Adding the log entry $\lambda_{d,n}$ to the sticky log Λ_d. 7) Adding the log entry $\lambda_{c,1}$ to the sticky log Λ_c.
Reading, updating or transferring d.	1) Creating a log entry $\lambda_{d,n}$ about the performed action. 2) Adding the log entry $\lambda_{d,n}$ to the sticky log Λ_d.
Deleting d. (not the last data instance.)	1) Creating a log entry $\lambda_{d,n}$ about the deletion of the data instance. 2) Adding the log entry $\lambda_{d,n}$ to the sticky log Λ_d. 3) Identifying another data instance e of the data item by using a reference (of a copy action) in Λ_d. 4) Merging the logs Λ_d and Λ_e by adding all log entries of Λ_d to Λ_e. 5) Updating all references in other logs pointing to Λ_d to point to Λ_e.
Deleting of the last data instance d.	1) Creating a log entry $\lambda_{d,n}$ about the deletion of the data instance. 2) Adding the log entry $\lambda_{d,n}$ to the sticky log Λ_d. 3) Returning Λ_d to the owner of the data item (e.g. the person concerned).

of the input arc, and the places involved in the action p, p_s, p_d, and/or p_r (depending on the category of action). If the action is a copy action, the arc expression $E(a_o)$ of the output arc leading to p_d is also required. The log entry λ_d (and λ_c of copy actions) is added to the sticky log m_d (respectively m_c) of the associated data instance d (respectively c). The output is the set of all sticky logs including the updated sticky log m_d (and m_c).

As DiALog does not define a specific level of detail for observing the processing, the sticky logging mechanism does also not define a specific level of detail. Depending on the required or desired detail level, an entity may be a software, a server or workstation as well as a person or a organizational entity, etc. Analogously, data items may be single words in an address, the full address or a text document containing the address. Any granularity can be used as required by contracts or laws. The specific level must be chosen whenever the sticky logging implementation is used (see Section 5.3.3).

Listing 5.1: *Logging operator to log actions:*

The input is the set of sticky logs and the elements of the logical execution modeling the performed action.

```
01  INPUT: M: set of sticky_logs,
           t: transition,
           E(a_i): arc_expression,
           E(a_o): arc_expression,
           p: place,
           p_s: place,
           p_d: place,
           p_r: place;
```

The logical execution models data instances as tokens. When a transition occurs the value of the token is assigned to the variable n by the arc expression $E(a_i)$ of the input arc. The value of a token uniquely identifies the

data instance. If the identifier in the monitored execution is based on this value, the data instance will also be uniquely identifiable in the monitored execution. The arc expression is evaluated and the identified data instance is assigned to d:

```
02  data_instance d =
        getDataInstance(E(a_i).evaluateExpression());
```

The logical execution models the category as label (without index) of the transition. The category of the action is assigned to χ:

```
03  category χ = t.getCategory();
```

The copy action is special, because two data instances are involved. These are the source instance and the newly created data instance. We need to identify the newly created data instance. Analogously to the arc expression $E(a_i)$, the arc expression $E(a_o)$ is used to identify the newly created instance.

```
04  if (χ == "copy")
05      data_instance created_d =
        getDataInstance(E(a_o).evaluateExpression());
```

The logical execution models actors as labels of places p. Depending on the category of an action two actors may be involved, which are assigned to α and β. Transfer actions involve two actors: the sender p_s and the receiver p_r. In copy actions, the two actors, the source p_s and the destination p_d, are involved. All other actions involve only one actor.

```
06  if (χ == "transfer")
07      actor α = p_s.getActor();
08      actor β = p_r.getActor();
```

```
09  else
10      if (χ == "copy")
11          actor α = p_s.getActor();
12          actor β = p_d.getActor();

13      else
14          actor α = p.getActor();
15          actor β = ϵ;
```

The purpose is encoded as index of the label of the occurring transition t.

```
16  action_purpose Ψ = t.getLabel().getIndex();
```

An identifier of the action as well as an identifier of the preceding action needs to be logged to achieve an order of actions. Each identifier is required to be unique.

```
17  action_identifier id = createUniqueId();
18  action_identifier pid =
        getIdOfLastLogEntryOf(d);
```

After retrieving the information about the processing the log entry is created. If the action is a copy action, a second log entry for the newly created data instance will be created, too.

```
19  log_entry λ1 = (χ, α, β, Ψ, id, pid);
20  if (χ == "copy")
21      log_entry λ2 = (χ, α, β, Ψ, id, null);
```

Finally, the log entries are added to the associated sticky logs. Then the set of all sticky logs is updated and returned.

```
22  sticky_log m_1 = M.getStickyLogOf(d);
23  set_of_log_entries Λ_1 =
        m_1.getSetOfLogEntries();
24  M.updateStickyLog(d, Λ_1 ∪ {λ_1});
25  if (χ == "copy")
26      sticky_log m_2 =
            M.getStickyLogOf(created_d);
27      set_of_log_entries Λ_2 =
            m_2.getSetOfLogEntries();
28      M.updateStickyLog(created_d, Λ_2 ∪ {λ_2});

29  OUPUT: M;
```

If the person concerned requests information before the processing ends, the current log will need to be received. One can receive the current log by forwarding a request to all actors that processed copies of the data item. The forwarding can be done following the same paths as the copied data instances. Then the logs are returned and merged. The following example depicts a log entry generated by the sticky logging mechanism.

Example 5.1: *A Log Entry generated by the Sticky Logging Mechanism.*

In step 6 of our scenario (see Section 3.1), the health record of Jane Doe is transferred from the Middle Rhine Hospital ($\alpha = MRH$) to the University of Koblenz ($\beta = UKOB$). The data instance d of the health record, which is transferred, has the identifier $record\_JD_{copy\_1}$. To log the action t_4 the hospital creates a log entry describing the action by its category $\chi = transfer$, the involved actors α and β, as well as the purpose Ψ of sharing the record for research. The log entry is then added to the log Λ. Finally, the hospital connects Λ with the data instance d:

$$(record\_JD_{copy\_1}, \{('transfer',\ MRH,\ UKOB,\ 'sharing\ for\ research',\ t_4,\ t_3)\})$$

5.1.3 Reconstructing the Execution

Based on the information logged by the sticky logging mechanism, a transition system modeling the processing of one data item can be reconstructed. The reconstruction of the transition system can be done by using the information contained in the above-introduced data structure. Listing 5.2 depicts the reconstruction operation for a log entry.

The reconstruction takes the set of all sticky logs M of one data item as input and creates a colored Petri net, which models the actors involved in the processing and actions performed on the data. Each sticky log m of the set of all sticky logs M is selected and the corresponding places, transitions, arcs, and node functions are specified.

Listing 5.2 depicts an algorithm implementing the mathematical operation describing the reconstruction of one log entry. The input of the operation is the log entry in combination with the already reconstructed parts of the colored Petri net modeling the reconstructed execution. Not the complete colored Petri net, but only the required information is passed. The required information is the sets of places, the set of transitions, and the set of arcs as well as the node function. The output is the extended part of the colored Petri net modeling the processed log entry.

Listing 5.2: *Reconstructing operation*

The Input is the log entry and the sets of already reconstructed places P, transitions T, arcs A, and the node function N.

```
01  INPUT: λ: log_entry,
```

```
P: set of places,
T: set of transitions,
A: set of arcs,
N: node_function;
```

The first step of the reconstruction is the creation of the places p_i and p_o for the involved actors. The actors are specified by α and β of the log entry λ. Each place represents a certain actor. The actors are distinguished by the unique identifier of the actors, as defined by the monitored execution. The reconstructed places are added to the set of places.

```
02  get α from λ;
03  place p_α = createPlaceModeling(α);
04  P = P ∪ {p_α};

05  get β from λ;
06  if (β != ϵ)
07      place p_β = createPlaceModeling(β);
08      P = P ∪ {p_β};
09  else
10      place p_β = p_α;
```

Then a transition t modeling the logged action is reconstructed and added to the set of transitions T. For actions which occur multiple times only one transition is created. Actions are identical, if their input places, their output places, their category, and their purposes are the same. The action is identified in the log entry by the value of id, which uniquely identifies the transition.

```
11  get id from λ;
12  transition t = createTransitionModeling(id);
13  T = T ∪ {id};
```

For the transition the input and output arcs a_i and a_o are created and added to the set of arcs A. The node functions of these arcs $N(a_i)$ and $N(a_o)$ are defined as $N(a_i) = (p_i, t)$ and $N(a_o) = (t, p_o)$. In the case that $\beta = \epsilon$, α specifies p_i and p_o of the transition.

```
14  arc a_i = createArc();
15  arc a_o = createArc();
16  A = A ∪ {a_i, a_o}
17  N(a_i) = (p_α, t)
18  N(a_o) = (t, p_β)
```

Finally, the updated sets and node function are returned.

```
19  OUTPUT: P, T, A, N
```

5.2 Proof of Soundness and Completeness

A sound and complete reconstructed execution can be used for auditing purposes as discussed in Section 4.3. We prove that the sticky logging mechanism can create reconstructed executions that have these qualities.

Proposition 5.1:

A reconstructed execution created by means of the sticky logging mechanism is sound and complete (as defined in Section 4.3) regarding the executed subsystem of the logical execution. □

We prove the proposition by induction over the structural length of the

colored Petri net modeling the logical execution. The basis of the induction is a logical execution consisting only of a create action. We prove that the proposition holds for this *minimal (not empty)* logical execution. As induction step, we extend a given logical execution which fulfills the proposition by one action. We prove the proposition by extending the workflow by each category of action.

Proof 5.1: *Soundness and Completeness of Sticky Logging by Induction*

Basis of Induction

Each processing of a data item starts with the creation of the first data instance. The basis of the induction is a workflow consisting only of one step, a create action creating the first data instance.

The logical execution of the first step is L_1 and consists of one transition t_1 with the label $create$, one place p_1 that represents the actor, and an arc a_1 leading from t_1 to p_1 ($N(a_1) = (t_1, p_1)$) with the arc expression u. This logical execution is depicted in Figure 5.1. The logical execution consists of two more arcs and the additional place p_c modeling the counter for creating unique identifiers.

The execution happens by the occurrence of the create transition. This step ($i = 1$) creates a new token with a new, unique value ($= X$) and adds this token to p_1. The monitoring (according to Listing 5.1) of the logical execution leads to the following monitored execution M_1:

$$M_1 = \{(X, (\text{“}create\text{”}, actor_{\text{modeled by } p_1}, \epsilon, \text{“}purpose\text{”}, 1, 0))\}$$

The reconstruction (according to Listing 5.2) based on the monitored execution leads to the reconstructed execution R_1. The reconstruction leads to a colored Petri net with the following sets: the set of places $P' = \{p'_1\}$, the set of transitions $T' = \{t'_1\}$, and the set of arcs $A' = \{a'_1\}$ with the node function $N'(a'_1) = (t'_1, p'_1)$.

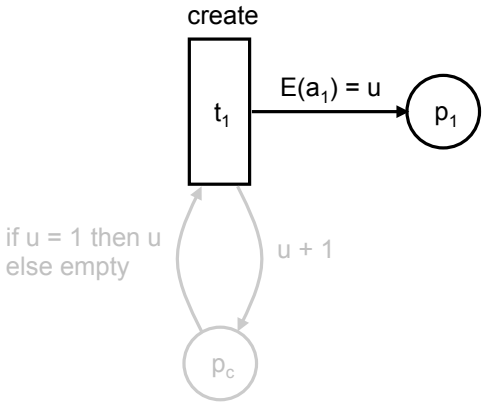

Figure 5.1: Induction Basis: Single Create Action.

Soundness: The following relation:

$$\Phi_1 = \{(t'_1, t_1), (p'_1, p_1), (a'_1, a_1)\}$$

is a simulation between L_1 and R_1. Hence, the reconstructed execution is sound with respect to the logical execution.

Completeness: The parts of the colored Petri net used to model the counter are not part of the actual executed subsystem S_1 of the logical execution. The following relation:

$$\Phi_2 = \{(t, t'), (p, p'), (a, a')\}$$

is a simulation between R_1 and S_1. Because Φ_2 can be specified, the reconstructed execution is complete with respect to the executed subsystem of the logical execution.

Induction Step

Given a reconstructed execution R_n, which is sound and complete with respect to the logical execution L_n, we show that a reconstructed execution R_{n+1} is also sound and complete with respect to the associated logical execution L_{n+1}. Whereas, L_{n+1} extends the logical execution L_n by one additional action. In detail, we have to prove the soundness and completeness of adding an action of each category.

- **Read actions:** We start by adding a read action. Before we add the action we choose one existing place p_k out of P_n as actor. Adding a read action to L_n, adds one transition t_{n+1} with the label $read$ and two arcs a_{m+1} and a_{m+2} leading from p_k to t_{n+1} and back to p_k. The node function for a_{m+1} and a_{m+2} is defined as $N(a_{m+1}) = (p_k, t_{n+1})$ and $N(a_{m+2}) = (t_{n+1}, p_k)$. The arc expression of both arcs is u. Figure 5.2 depicts the added parts of the colored Petri net.

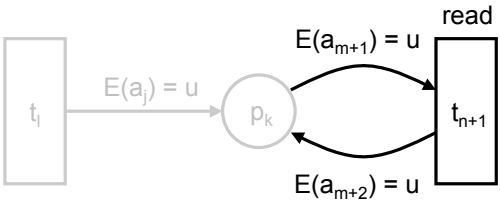

Figure 5.2: Induction Step: Read Action.

Monitoring the occurrence of t_{n+1}, which reads a data instance represented by a token with the value X at p_k, in step $i+1$ creates the log entry m_{i+1}:

$$m_{i+1} = (X, (\text{“}read\text{”}, actor_{modeled\ by\ p_k}, \epsilon, \text{“}purpose\text{”}, i+1, i))$$

The log entry m_{i+1} is added to the monitored execution M_i leading to M_{i+1}:

$$M_{i+1} = M_i \cup \{m_{i+1}\}$$

The reconstruction is an iterative process. The reconstruction of R_{n+1} starts by processing M_i. M_i leads to R_n, which is sound and complete with respect to L_n (basis of induction). After M_i, the last iteration step processes the newly added log entry m_{i+1}. Because R_n was reconstructed from M_i, it consists of the set of places P'_n, the set of transitions T'_n, and the set of arcs A'_n with the node function N'_n. R_{n+1} extends these sets. The set of places remains unchanged:

$$P'_{n+1} = P'_n$$

because the log entry contains only the actor modeled by p', which is already element of P'. However, the new transition t'_{n+1} is added to T'_n:

$$T'_{n+1} = T'_n \cup \{t'_{n+1}\}$$

The arcs are added to A'_n leading to:

$$A'_{n+1} = A'_n \cup \{a'_{m+1}, a'_{m+2}\}$$

Soundness: Let $\Phi_{1,n}$ be the simulation between L_n and R_n. The following relation

$$\Phi_{1,n+1} = \Phi_{1,n} \cup \{(t'_{n+1}, t_{n+1}), (a'_{m+1}, a_{m+1}), (a'_{m+2}, a_{m+2})\}$$

is a simulation between L_{n+1} and R_{n+1}. Thus, the reconstructed execution is sound with respect to the logical execution L_{n+1}.

Completeness: Let $\Phi_{2,n}$ be the simulation between R_n and the executed subsystem S_n of the logical execution. The occurrence of t_{n+1} leads to the executed subsystem S_{n+1}. The following relation

$$\Phi_{2,n+1} = \Phi_{2,n} \cup \{(t_{n+1}, t'_{n+1}), (a_{m+1}, a'_{m+1}), (a_{m+2}, a'_{m+2})\}$$

is a simulation between R_{n+1} and S_{n+1}. Because $\Phi_{2,n+1}$ can be specified, the reconstructed execution is complete with respect to the executed subsystem of the logical execution L_{n+1}.

- **Update and actions:** The soundness and completeness proofs of update actions can be shown analogously to the proofs of read actions. The induction step can be seen in Figure 5.3.

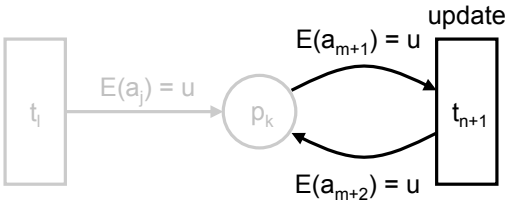

Figure 5.3: Induction Step: Update Action.

- **Copy actions:** The next action is the copy action. Before we add the action we choose one existing place p_k out of P_n as actor. Adding a copy action to L_n adds one transition t_{n+1} with the label *copy*, the place p_d (where the new instance is created) and three arcs a_{m+1}, a_{m+2} and a_{m+3} leading from p to t_{n+1}, from t_{n+1} to p_k, and from t_{n+1} to p_d. The node function for a_{m+1}, a_{m+2}, and a_{m+3} is defined as $N_{n+1}(a_{m+1}) = (p_k, t_{n+1})$, $N_{n+1}(a_{m+2}) = (t_{n+1}, p_k)$ and $N_{n+1}(a_{m+3}) = (t_{n+1}, p_d)$. The arc expression of a_{m+1} and a_{m+2} is u and the one of a_{m+3} is v. Two arcs connecting the transition with

the counter place are added. These are irrelevant for the following proof as they do not model the workflow itself and are not part of the executed subsystem. Figure 5.4 depicts the extension of the workflow model by a copy action.

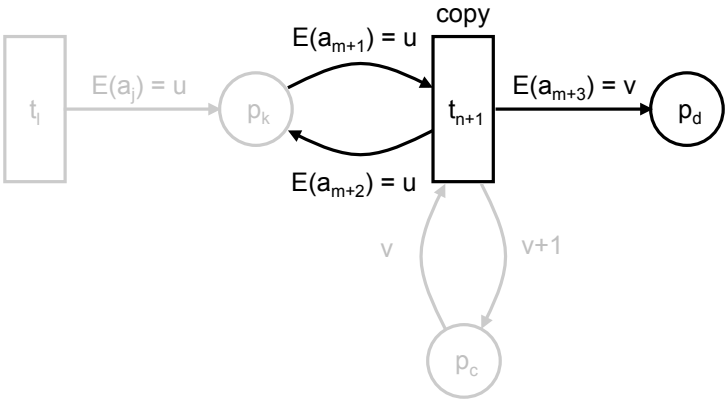

Figure 5.4: Induction Step: Copy Action.

Monitoring the occurrence of t_{n+1} (by means of the operator specified in Listing 5.1), which occurs in step $i+1$ and copies a data instance represented by a token with the value X at p_k and creating a new instance represented by a token with the value Y at p_d, creates the log entries $m_{X,i+1}$ and $m_{Y,1}$. $m_{X,i+1}$ is the log entry associated to X and $m_{Y,1}$ is the log entry associated to Y.

$$m_{X,i+1} =$$
$$(X, (\text{“}copy\text{”}, actor_{modeled\ by\ p_k}, actor_{modeled\ by\ p_d}, \text{“}purpose\text{”}, i+1, i))$$
$$m_{Y,1} =$$
$$(Y, (\text{“}copy\text{”}, actor_{modeled\ by\ p_k}, actor_{modeled\ by\ p_d}, \text{“}purpose\text{”}, i+1, i))$$

The log entry $m_{X,i+1}$ is then added to the monitored execution $M_{X,i}$ leading to $M_{X,i+1}$: The log entry $m_{Y,1}$ is then added to the monitored execution $M_{Y,0}$, which is empty, leading to $M_{Y,1}$:

$$M_{X,i+1} = M_{X,i} \cup \{m_{X,i+1}\}$$
$$M_{Y,1} = M_{Y,0} \cup \{m_{Y,1}\}$$

The reconstruction (cf. Listing 5.2) of R_{n+1} starts by processing the sticky logs of both data instances $M_{X,i}$ and $M_{Y,0}$ leading to E_i. Afterwards, the last iteration step processes the newly added log entries $m_{X,i+1}$ and $m_{Y,1}$. R_n is sound and complete with respect to L_n (basis of induction) and consists of the set of places P'_n, the set of transitions T'_n, and the set of arcs A'_n with the node function N'_n. R_{n+1} extends these sets. The reconstructed place p'_d may model an actor having been involved in the workflow already or not. This does not affect the following proof as all actors are uniquely identifiable leading to the same reconstructed places. Thus, p'_d is added to P'_n leading to:

$$P'_{n+1} = P'_n \cup \{p'_d\}$$

The new transition t'_{n+1} is also added to T'_n:

$$T'_{n+1} = T'_n \cup \{t'_{n+1}\}$$

Finally, the arcs are added to A'_n leading to:

$$A'_{n+1} = A'_n \cup \{a'_{m+1}, a'_{m+2}, a'_{m+3}\}$$

Soundness: Let $\Phi_{1,n}$ be the simulation between L_n and R_n. The following relation:

$$\Phi_{1,n+1} = \Phi_{1,n} \cup$$
$$\{(p'_d, p_d), (t'_{n+1}, t_{n+1}), (a'_{m+1}, a_{m+1}), (a'_{m+2}, a_{m+2}), (a'_{m+3}, a_{m+3}))\}$$

is a simulation between L_{n+1} and R_{n+1}. Thus, the reconstructed execution is sound with respect to the logical execution L_{n+1}.

Completeness: Let $\Phi_{2,n}$ be the simulation between R_n and the executed subsystem S_n of the logical execution. The occurrence of t_{n+1} leads to the executed subsystem S_{n+1}. The following relation:

$$\Phi_{2,n+1} = \Phi_{2,n} \cup$$
$$\{(p_d, p'_d), (t_{n+1}, t'_{n+1}), (a_{m+1}, a'_{m+1}), (a_{m+2}, a'_{m+2}), (a_{m+3}, a'_{m+3}))\}$$

is a simulation between R_{n+1} and S_{n+1}. Because $\Phi_{2,n+1}$ can be specified, the reconstructed execution is complete with respect to the executed subsystem of the logical execution L_{n+1}.

- **Transfer actions:** Before we add a transfer action to the workflow we choose one existing place p_k out of P_n as actor. Adding the transfer action to L_n adds one transition t_{n+1} with the label $transfer$, the place p_r (receiving the transfered instance) and two arcs a_{m+1} and a_{m+2} leading from p_k to t_{n+1} and from t_{n+1} to p_r. The node function for a_{m+1} and a_{m+2} is defined as $N(a_{m+1}) = (p_k, t_{n+1})$ and $N(a_{m+2}) = (t_{n+1}, p_r)$. The arc expressions of a_{m+1} and a_{m+2} are u. The extended colored Petri net is depicted in Figure 5.5.

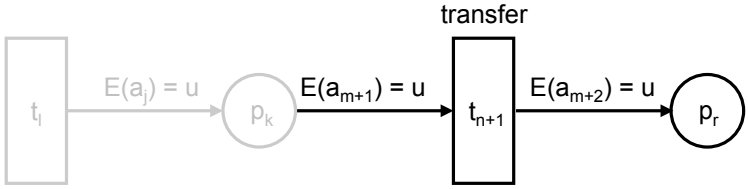

Figure 5.5: Induction Step: Transfer Action.

The occurrence of t_{n+1} transfers a data instance represented by a token with the value X from p_k to p_r. t_{n+1} occurs in step $i+1$ and creates the log entry m_{i+1}.

$$m_{i+1} = (X, (\text{“}transfer\text{”}, actor_{modeled\ by\ p_k}, \\ actor_{modeled\ by\ p_r}, \text{“}purpose\text{”}, i+1, i))$$

The log entry m_{i+1} is added to the monitored execution M_i leading to M_{i+1}:

$$M_{i+1} = M_i \cup \{m_{i+1}\}$$

R_{i+1} is reconstructed by starting with the sticky log $M_{X,i}$. Followed by processing the newly added log entry $m_{X,i+1}$. R_n is sound and complete with respect to R_n (basis of induction) and consists of the set of places P'_n, the set of transitions T'_n, as well as the set of arcs A'_n with the node function N'_n. R_{n+1} extends these sets and the node function. The reconstructed place p'_r may model an actor, perhaps not having been involved in the workflow so far. This does not affect the following proof because all actors are uniquely identifiable leading to the same reconstructed places. Thus, p'_r is added to P'_n leading to:

$$P'_{n+1} = P'_n \cup \{p'_r\}$$

The new transition t'_{n+1} is also added to T'_n:

$$T'_{n+1} = T'_n \cup \{t'_{n+1}\}$$

Finally, the arcs are added to A'_n leading to:

$$A'_{n+1} = A'_n \cup \{a'_{m+1}, a'_{m+2}\}$$

Soundness: Let $\Phi_{1,n}$ be the simulation between L_n and R_n. The following relation

$$\Phi_{1,n+1} = \Phi_{1,n} \cup \{(p'_r, p_r), (t'_{n+1}, t_{n+1}), (a'_{m+1}, a_{m+1}), (a'_{m+2}, a_{m+2}))\}$$

is a simulation between L_{n+1} and R_{n+1}. Hence, the reconstructed execution is sound with respect to the logical execution L_{n+1}.

Completeness: Let $\Phi_{2,n}$ be the simulation between R_n and the executed subsystem S_n of the logical execution. The occurrence of t_{n+1} leads to the executed subsystem S_{n+1}. The following relation:

$$\Phi_{2,n+1} = \Phi_{2,n} \cup \{(p_r, p'_r), (t_{n+1}, t'_{n+1}), (a_{m+1}, a'_{m+1}), (a_{m+2}, a'_{m+2})\}$$

is a simulation between R_{n+1} and S_{n+1}. Because $\Phi_{2,n+1}$ can be specified, the reconstructed execution is complete with respect to the executed subsystem of the logical execution L_{n+1}.

- **Delete actions:** Before we add a delete action to the workflow we choose one existing place p_k out of P_n as actor. Applying the delete action to L_n adds one transition t_{n+1} with the label $delete$ and one arc a_{m+1} leading from p_k to t_{n+1}. The node function for a_{m+1} is defined as $N(a_{m+1}) = (p, t_{n+1})$. The arc expression of a_{m+1} is u. The extended colored Petri net is depicted in Figure 5.6.

  In step $i + 1$, t_{n+1} occurs. In this step, the data instance represented by a token with the value X from p_k is deleted. Monitoring the occurrence creates the log entry m_{i+1}.

  $$m_{i+1} = (X, (\text{“}delete\text{”}, actor_{modeled\ by\ p_k}, \epsilon, \text{“}purpose\text{”}, i + 1, i))$$

  The log entry m_{i+1} is added to the monitored execution M_i leading to M_{i+1}:

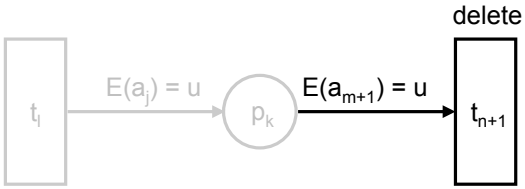

Figure 5.6: Induction Step: Delete Action.

$$M_{i+1} = M_i \cup \{m_{i+1}\}$$

R_{n+1} is reconstructed by starting with the sticky log $M_{X,i}$. Afterwards, the last iteration step processes the newly added log entry $m_{X,i+1}$. R_n is sound and complete with respect to R_n (basis of induction) and consists of the set of places P'_n, the set of transitions T'_n, as well as the set of arcs A'_n with the node function N'_n. R_{n+1} extends these sets. As no new place is added, the set P' stays unchanged:

$$P'_{n+1} = P'_n$$

The new transition t'_{n+1} is added to the set T':

$$T'_{n+1} = T'_n \cup \{t'_{n+1}\}$$

Finally, the arcs are added to A' leading to:

$$A'_{n+1} = A'_n \cup \{a'_{m+1}, a'_{m+2}\}$$

Soundness: Be $\Phi_{1,n}$ the simulation between L_n and R_n. The following relation

$$\Phi_{1,n+1} = \Phi_{1,n} \cup \{(t'_{n+1}, t_{n+1}), (a'_{m+1}, a_{m+1}))\}$$

is a simulation between L_{n+1} and R_{n+1}. Hence, the reconstructed execution is sound with respect to the logical execution L_{n+1}.

Completeness: Be $\Phi_{2,n}$ the simulation between R_n and the executed subsystem S_n of the logical execution. Performing t_{n+1} leads to the executed subsystem S_{n+1}. The relation:

$$\Phi_{2,n+1} = \Phi_{2,n} \cup \{(t_{n+1}, t'_{n+1}), (a_{m+1}, a'_{m+1})\}$$

is a simulation between R_{n+1} and S_{n+1}. Because $\Phi_{2,n+1}$ can be specified, the reconstructed execution is complete with respect to the executed subsystem of the logical execution L_{n+1}.

By means of the basis of the induction we have shown that a minimal logical execution can be monitored by the sticky logging approach leading to a reconstructed execution, which is sound and complete with respect to the logical execution. By means of the induction step we have proved that this also holds after adding the execution of any category of action to an existing logical execution. We have proved that a reconstructed execution created by means of the sticky logging mechanism is sound and complete (as defined in Section 4.3) regarding the executed subsystem of the logical execution. ■

5.3 The Sticky Logging Mechanism

In section 5.1, we have depicted a mathematical structure that specifies the information required to be logged by the sticky logging mechanism. And we have presented an algorithm describing how to generate log entries. In this section, we introduce a semantic formalism to monitor distributed data processing that implements the mathematical structure and formal algorithm. To

demonstrate its feasibility, we implement the sticky logging mechanism as JBoss message handler. However, such a distributed monitoring mechanism rises various issues of accountability, confidentiality, security and privacy, which we address later on.

5.3.1 Semantic Monitoring

To enable a semi-automated analysis of sticky logs, we introduce a semantic formalism (cf. *Requirement Well-defined Semantics*). We chose OWL [Motik et al., 2009] as well as RDF [Klyne et al., 2004] and use data types in compliance with the XML schema definition (XSD) [Peterson et al., 2006] (cf. *Requirement Standardized Interfaces*). The semantic formalism consists of the sticky logging ontology. The sticky logging ontology introduces the upper level concepts `StickyLoggingConcept` and `DomainConcept`. The sub-concepts of `StickyLoggingConcept` define the sticky logging formalism. These are the concepts `DataInstance`, `Action`, `Entity`, `LogEntry` and `ActionType` with various sub-concepts representing the different sorts of actions. The `DomainConcept` has two sub-concepts `PurposeCategory` and `CommunicationMean`, which serve as upper-level concepts for all concepts defined in domain ontologies and referred to by sticky logs. The concepts of the sticky logging ontology and their relations to each other are depicted in Figure 5.7. In the following we introduce the various concepts and their properties.

The monitoring mechanism needs to clearly identify the processed data item and its sources (cf. *Legal Requirement Exhaustiveness*). Data items may have multiple instances, e.g. through copying. A sticky log is attached to each instance clearly identifying the different instances. Logs of different instances need to be combined whenever a data instance is deleted. Hence, all associated data instance require to be clearly identified in the sticky log. To represent the instance of a data item, we introduce the concept `DataInstance`.

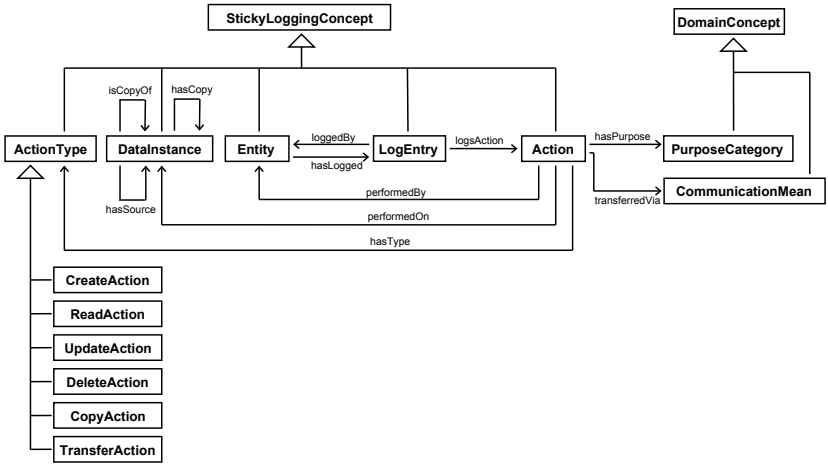

Figure 5.7: Sticky Logging Ontology.

Concept Definition 5.1: `DataInstance`

The concept `DataInstance` has the following properties:

- `hasCopy` [0..n][1]: If this data instance has been copied, this property will link to the resources describing the copy.
- `hasSource` [0..n]: If the data item of the described instance contains data of another data item (e.g. merging of two instances or extracting from another data item), the value of this property will be the URI of the source.
- `hasTimeStamp` [1]: At that point in time the information about this data instance has been added to the log. Given two different descrip-

[1]The value given in brackets specifies the cardinalities of the properties.

tions of the same data instance, this information can be used to determine which one is more up to date.

- `hasUUID` [1]: A unique identifier clearly identifying the data instance. To assure that the identifier is unique we use Universally Unique Identifiers (UUID) [Leach et al., 2005].

- `isCopyOf` [0..1]: If this data instance is a copy of another instance of the same data item, this property will link to the resource describing the original instance.

- `isDeleted` [0..1]: After deleting the data instance, the boolean value of this property must be set to 'true'.

- `isPrimary` [0..1]: A boolean value, which is used to indicate whether this instance is the original instance or whether it is a copy. The original instance is the instance that is created first.

and the following axiomatization in OWL:

$$
\begin{aligned}
\texttt{DataInstance} \sqsubseteq\ & \texttt{StickyLoggingConcept} \sqcap \\
& \forall\, \texttt{hasCopy.DataInstance} \sqcap \\
& \forall\, \texttt{hasSource.DataInstance} \sqcap \\
& \forall\, \texttt{hasTimeStamp.date} \sqcap \\
& =1\, \texttt{hasTimeStamp.date} \sqcap \\
& \forall\, \texttt{hasUUID.uuid} \sqcap =1\, \texttt{hasUUID.uuid} \sqcap \\
& \forall\, \texttt{isCopyOf.DataInstance} \sqcap \\
& \leq 1\, \texttt{isCopyOf.DataInstance} \sqcap \\
& \forall\, \texttt{isDeleted.boolean} \sqcap
\end{aligned}
$$

≤ 1 isDeleted.boolean $\sqcap$
$\forall$ isPrimary.boolean $\sqcap$
≤ 1 isPrimary.booelan

Listing 5.3 depicts a snippet of the log that is attached to the instance of Jane Doe's health record created by the Middle Rhine Hospital. The snippet contains the triples describing the data instance.

Listing 5.3: *Example of a Data Instance.*

```
:record_JD_1  rdf:type          sl:DataInstance
:record_JD_1  sl:hasUUID        "50a410f0-bc8a-..."
:record_JD_1  sl:isPrimary      "true"
:record_JD_1  sl:hasTimeStamp   "Mon Nov 23 11:58.."
```

To increase readability, we use *sl:* as namespace for the concepts and properties defined as part of the sticky logging ontology instead of `http://west.uni-koblenz.de/stickylogging`. To enhance the readability, we replace the URIs defining the resources by short identifiers (e.g. `:record_JD_1` instead of `http://www.mrh.example/50a410f0-bc8a-4afe-a7c4-...`).

As discussed before, specific actions require to be logged. Which actions are logged depends on the agreed-upon level of granularity and is part of underlying contracts. Sticky logging describes each action that requires to be logged by its own log entry. The log entry is attached to the sticky log associated with the data instance the action is performed on. In the log entry, an action is represented by the concept `Action`.

Concept Definition 5.2: `Action`

The concept `Action` has the following properties:

- `hasPurpose` [1..n]: Besides the category of an action its purpose is of importance, to check whether a performed action has been justified. The possible purposes of an action depend on the performing entity and on its domain. Actions in the hospital domain have purposes such as *treatment*, *adjusting medication*, etc. To enable a flexible definition of purposes, concepts defined in domain ontologies are used. The property `hasPurpose` links to a concept of such an ontology describing the purpose of the action.
- `hasSequentialNumber` [1]: An integer value holding the sequential number of the action. The sequential number is used to achieve the partial order of the performed actions.
- `hasTimeStamp` [1]: The point in time the action has been performed.
- `hasUUID` [1]: A unique identifier that clearly identifies this action.
- `performedBy` [1]: An URI pointing to the resource describing the entity that has performed the action.
- `performedOnDataInstance` [1]: This property links to the resource describing the data instance the action is performed on. The link is required to clearly connect action and data instance even after the log has been merged with logs of other data instances.
- `transferredAs` [0..1]: A string specifying the identification of a data instance during a transfer (e.g. parameter of the Web service the data instance is passed, or the attachment identifier of an e-mail).
- `transferredVia` [0..1]: If the action transfers the data instance,

this property will link to the used communication means (e.g. an e-mail message identifier, or an URI of a Web service operation).

and the following axiomatization in OWL:

$$
\begin{aligned}
\texttt{Action} \sqsubseteq\ & \texttt{StickyLoggingConcept} \sqcap \\
& \forall\, \texttt{hasPurpose.PurposeCategory} \sqcap \\
& \geq 1\, \texttt{hasPurpose.PurposeCategory} \sqcap \\
& \forall\, \texttt{hasSequentialNumber.integer} \sqcap \\
& = 1\, \texttt{hasSequentialNumber.integer} \sqcap \\
& \forall\, \texttt{hasTimeStamp.date} \sqcap = 1\, \texttt{hasTimeStamp.date} \sqcap \\
& \forall\, \texttt{hasUUID.uuid} \sqcap = 1\, \texttt{hasUUID.uuid} \sqcap \\
& \forall\, \texttt{performedBy.Entity} \sqcap = 1\, \texttt{performedBy.Entity} \sqcap \\
& \forall\, \texttt{performedOnDataInstance.DataInstance} \sqcap \\
& = 1\, \texttt{performedOnDataInstance.DataInstance} \sqcap \\
& \forall\, \texttt{transferredAs.string} \sqcap \\
& \leq 1\, \texttt{transferredAs.string} \sqcap \\
& \forall\, \texttt{transferredVia.CommunicationMean} \sqcap \\
& \leq 1\, \texttt{transferredVia.CommunicationMean}
\end{aligned}
$$

The DiALog model distinguishes six types of actions in Chapter 4. These are create, read, update, copy, transfer, and delete actions. These actions are represented by the concepts `CreateAction`, `ReadAction`, `UpdateAction`, `CopyAction`, `TransferAction`, and `DeleteAction`. All action types are sub-concepts of `ActionType` and have the following axiomatization in OWL:

$$\texttt{CreateAction} \sqsubseteq \texttt{ActionTypes}$$
$$\texttt{ReadAction} \sqsubseteq \texttt{ActionTypes}$$
$$\texttt{UpdateAction} \sqsubseteq \texttt{ActionTypes}$$
$$\texttt{CopyAction} \sqsubseteq \texttt{ActionTypes}$$
$$\texttt{TransferAction} \sqsubseteq \texttt{ActionTypes}$$
$$\texttt{DeleteAction} \sqsubseteq \texttt{ActionTypes}$$

After the deletion of a data instance the associated log is returned and merged with the log of the origin of the copy. Thus, the sticky logging uses two more actions categories the `LogReturnAction` and the `LogMergeAction`. These additional actions are not performed on data instances but on the associated logs after an instance is deleted. The action concepts have the following axiomatization in OWL:

$$\texttt{LogReturnAction} \sqsubseteq \texttt{ActionTypes}$$
$$\texttt{LogMergeAction} \sqsubseteq \texttt{ActionTypes}$$

Further required actions or sub-concepts of existing action concepts can be defined in domain ontologies (e.g. `De-identifyAction` as sub-concept of `UpdateAction`). Listing 5.4 depicts the create action performed on Jane Doe's health record in the health care scenario.

Listing 5.4: *Example of a Create Action.*

```
:action_1  rdf:type            sl:CreateAction
:action_1  sl:hasPurpose       "treatment"
:action_1  sl:performedBy      :mrh
```

```
:action_1  sl:hasUUID           "e92b0434-3cba-..."
:action_1  sl:hasTimeStamp      "Mon Nov 23 ..."
:action_1  sl:performedOnDataInstance  :record_JD_1
:action_1  sl:hasSequentialNumber      "1"
```

The explicit identification of the entity that performs actions on the data instance is crucial for achieving accountability (cf. *Legal Requirement Identifiability*). An organization has also to be clearly associated with all log entries it is responsible for. It is also of importance that log entries may not be modified or changed by another entity. We introduce the concept `Entity` to represent an entity within a sticky log. The formalism demands the use of mechanisms to sign RDF graphs as described in [Carroll, 2003].

Concept Definition 5.3: `Entity`

The concept `Entity` has the following properties:

- `hasAddress` [0..1]: A contact address of the entity that meets the requirements made by law or contract (for instance a postal address or an e-mail address).
- `hasID` [1]: A legally effective identifier of the entity (e.g. trade register number, international securities identifying number (ISIN), etc.).
- `hasLogged` [0..n]: This property is an URI linking to the resources describing the log entries the entity is responsible for.
- `hasName` [0..1]: A literal representing the name of this entity.
- `hasPGPCertificate` [1]: An URI linking to the entities PGP certificate.
- `hasSignature` [1]: The signature that signs all log entries connected with this entity.

- `hasTimeStamp` [1]: The point in time the information about this entity has been added to the log. Given two different descriptions of the same entity, this information can be used to determine which one is more up to date.
- `hasUUID` [1]: A unique identifier that clearly identifies this entity.

and the following axiomatization in OWL:

$$\begin{aligned}\texttt{Entity} \sqsubseteq\ & \texttt{StickyLoggingConcept} \sqcap \\ & \leq 1\,\texttt{hasAddress} \sqcap \\ & = 1\,\texttt{hasID} \sqcap \\ & \forall\,\texttt{hasLogged.Entity} \sqcap \\ & \leq 1\,\texttt{hasName} \sqcap \\ & \forall\,\texttt{hasPGPCertificate.uri} \sqcap \\ & = 1\,\texttt{hasPGPCertificate.uri} \sqcap \\ & \forall\,\texttt{hasSignature.string} \sqcap = 1\,\texttt{hasSignature.string} \sqcap \\ & \forall\,\texttt{hasTimeStamp.date} \sqcap = 1\,\texttt{hasTimeStamp.date} \sqcap \\ & \forall\,\texttt{hasUUID.uuid} \sqcap = 1\,\texttt{hasUUID.uuid}\end{aligned}$$

The example in Listing 5.5 depicts how the Middle Rhine Hospital identifies itself by instantiating the *Entity*-concept and setting its properties.

Listing 5.5: *Example of an Entity Description.*

```
:mrh rdf:type               sl:Entity
:mrh sl:hasName             "Middle Rhine Hospital"
:mrh sl:hasID               "ISIN US0001234567"
:mrh sl:hasPGPCertificate   "www.mrh.example/cert.asc"
:mrh sl:Signature           "HrdSDFc..."
```

```
:mrh sl:hasTimeStamp        "Mon Nov 23 11:58:19..."
```

The concept LogEntry represents one log entry. A log entry contains the record of all information about one single action performed on the processed data.

Concept Definition 5.4: LogEntry

The properties of the LogEntry-concept are

- hasTimeStamp [1]: The point in time the log entry has been added to the log.
- hasUUID [1]: A unique identifier that clearly identifies this log entry.
- loggedBy [1]: This property contains an URI that points to the resource describing the entity logging this entry.
- logsAction [1]: An URI linking to the resource describing the action logged by this entry.

and the following axiomatization in OWL:

$$\begin{aligned}\texttt{LogEntry} \sqsubseteq\ & \texttt{StickyLoggingConcept} \sqcap \\ & \forall\, \texttt{hasTimeStamp.date} \sqcap = 1\, \texttt{hasTimeStamp.date} \sqcap \\ & \forall\, \texttt{hasUUID.uuid} \sqcap = 1\, \texttt{hasUUID.uuid} \sqcap \\ & \forall\, \texttt{loggedBy.Entity} \sqcap = 1\, \texttt{loggedBy.Entity} \sqcap \\ & \forall\, \texttt{logsAction.Action} \sqcap = 1\, \texttt{logsAction.Action}\end{aligned}$$

Listing 5.6 depicts the beginning of a log entry about the create action of Jane Doe's health record (see Listing 5.4) logged by the Middle Rhine

Hospital. The last line of the listing shows the link between the logging entity (the Middle Rhine Hospital) and the logged entry.

Listing 5.6: *Example of a Log Entry.*

```
:logEntry_1  rdf:type          sl:LogEntry
:logEntry_1  sl:hasUUID        "cf2f30c0-39c1-..."
:logEntry_1  sl:logsAction     :action_1
:logEntry_1  sl:hasTimeStamp   "Mon Nov 23 11:58.."
:logEntry_1  sl:hasSequentialNumber  "1"

:mrh         sl:hasLogged      :logEntry_1
```

5.3.2 Risk Management

Monitoring executions of processes raises various issues of accountability, confidentiality, security and privacy. We shortly address some prominent issues and discuss the solution provided by sticky logging:

Accountability: One issue is that a logging entity must not be able to deny a established log entry. To guarantee that an organization is not able to do so, the sticky logging mechanism makes use of signatures. Signatures assure that a logging entity is accountable for the log entries it made. Another problem is the correctness of the log itself that is threatened by a modification of the log by any entity passing the log. The signatures are also used to assure that the log is not modified by another organization or entity. For both purposes each logging entity has to sign its log entries by means of a digital signature mechanism, e.g. the approach presented in [Carroll, 2003].

Confidentiality and Security: Another issue is the security of confidential information about the involved organizations. As the sticky log is attached to the data instances and moved along the execution path, other organizations may access the contained provenance information and reveal confidential internal matters of other involved parties. To restrict the access of third parties to the logged information, an access control mechanism is required (see *Legal Requirement Confidentiality*). We propose that each actor encrypts the logged information. If a public key infrastructure [Rivest et al., 1978] is utilized, the logging entities can use the public key of the data owner or person concerned to encrypt the log entries. The data owner or the person concerned can access the log entries by means of their private key.

Privacy and Information request: Privacy laws demand that information by request must be fulfilled. As different organizations are involved in the distributed data processing, each of these organizations monitors its part of the processing by means of the sticky logging mechanism. If the processing ends or information is requested, and the person concerned may request the information at any time and as often as they want during the processing, the log will be transferred to the person concerned. The person concerned is able to generate the transition system of the reconstructed execution from the logs representing the monitored execution (see Section 4.1). The reconstructed execution represents the parts of the workflow involved in the processing of the private data. Finally, the person concerned is able to audit whether contractual or legal agreements have been violated.

5.3.3 Prototype Description

By implementing Sticky Logging[2] we have shown its feasibility. We implemented the sticky logging mechanism as a Java-based API. The API can be integrated in any kind of software.

[2]Together with the bachelor thesis writer Martin Schnorr.

Execution environments, such as Web application servers, are common for the Web. To integrate multiple applications at once as well as to easily integrate legacy applications, the sticky logging mechanism should be part of execution environments or a layer between execution environment (i.e. JBoss) and business software (i.e. Web services). As such a layer, the sticky logging mechanism can observe the communicated data and manage the passing of logs. The prototype implements the sticky logging mechanism as a generic layer for JBoss, as depicted in Figure 5.8. As such a layer, sticky logging can not observe outgoing calls apart from calls to an observed application or execution environment. Details about the internal application logic can also not be observed. In both cases, the sticky logging API must be directly integrated into the application.

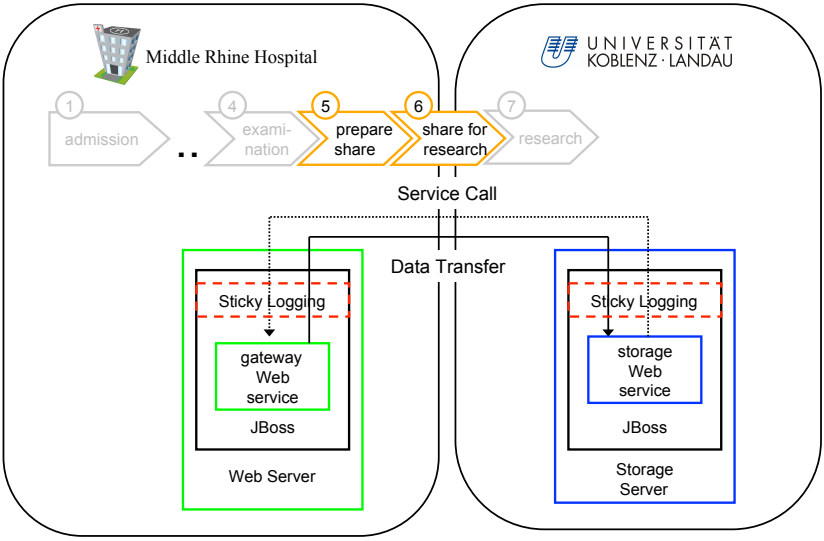

Figure 5.8: Example architecture with JBoss.

To access the layer between execution environment and Web application

or Web service, the environment of JBoss provides a handler API. Handlers are executed as part of the execution stack of Web service calls. Incoming Web service calls as well as the outgoing return messages can be intercepted through the handlers. By implementing such a handler, the sticky logging mechanism can be positioned between the JBoss and the Web service. The sticky logging mechanism can detach incoming logs from the service call and attach the updated logs to the return message of the service call. The updated logs consist of the incoming log and the log entries attached by the handler as well as by the sticky logging API called directly from the services application logic.

Technically, the prototype attaches the logs by including them into the SOAP messages [Gudgin et al., 2007] that are used to call services. The log is included using the attachment mechanism of SOAP, as depicted in Figure 5.9. After the execution of the service, the log is returned within the SOAP answer. The returned logs are merged with the log of the calling service. The logs make use of the sticky logging ontology we presented in Section 5.3.1. Sticky logging uses Sesame[3] to handle the RDF statements.

For the visualization of sticky logs, we[4] have implemented a tool named LogAnalyzer. The LogAnalyzer visualizes the log entries as a graphical processing trace. The graphical representation provides details about the entities performing actions on the observed data item and the actions they have performed. Figure 5.10 depicts a screenshot of the tool.

To show the functionality of the sticky logging mechanism, a scenario with a project partner in the project X-Media[5] has been implemented as prototype. In the scenario, an engineer collects data about an incident and writes a report. Both are forwarded to a department of the company organizing a task force. The implementation provides a client application that transfers the incident reports and data to a Web service. The Web service forwards the

[3] Sesame: http://openrdf.org/

[4] Together with the student worker Roland Naglo.

[5] "Knowledge Sharing and Reuse across Media" (X-Media, FP6-26978) funded by the Information Society Technologies (IST) 6th Framework Programme.

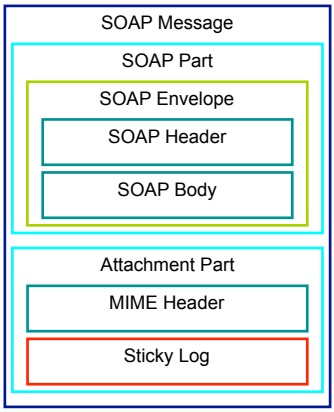

Figure 5.9: Sticky Log in a SOAP Message.

data to various experts (via e-mail) and analysis applications (via Web service). All applications, the client application as well as the Web services and e-mail clients use the sticky logging prototype. At any point in the process, the task force leader can access the monitored information via the LogAnalyzer to get an overview about the process and to audit the decisions made.

5.4 Related Work

Most of the work related to the sticky logging mechanism is also of relevance to DiALog. We discussed this work in Section 4.4. In this section, we discuss approaches that are related to aspects specific to the sticky logging.

In [Hallam-Baker and Behlendorf, 1996], the authors introduce the Extended Log File Format a logging solutions for the Web. This solution logs communication actions focusing on the communication of data by Web applications. It does not focus on the processing of the data and does not observe the processing of specific data items in distributed environments.

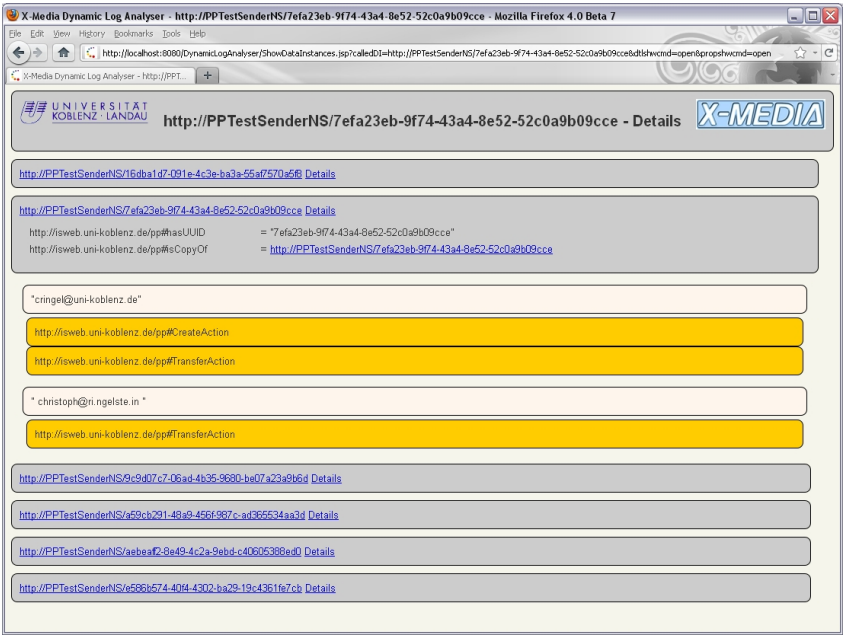

Figure 5.10: Screenshot LogAnalyzer.

Karjoth et al. introduce in [Karjoth et al., 2003] the sticky policy paradigm. When data is transmitted to an organization via Web form, the applicable policies and the user's opt-in and opt-out choices are also included in the form. If the data is transferred to another organization, the sticky policies will be transferred, as well. The work Karjoth et al. attach policies to data instead of logs. Another mechanism that attaches privacy-related information to data is the Platform for Privacy Preferences (P3P) [Wenning et al., 2006]. The P3P formalism is restricted to policies and allows only the use of a few pre-defined categories to describe the type of data and the purpose of their usage.

Other work also see the need for a specification of metadata. The Dublin

Core Metadata for Resource Discovery [Weibel et al., 1998] specifies concepts to describe properties of resources in the Web. With the concept creator it supports the specification of some provenance but processing histories can not be specified. The open provenance model specifies a graphical representation of provenance [Moreau et al., 2008]. It supports the modeling of processes and the usage of data in these processes. It does not provide concepts to specify action categories and purposes.

5.5 Summary

After we have provided DiALog in Chapter 4, we still require a monitoring mechanism. This mechanism must be able to observe the distributed data processing and to produce sound and complete reconstructed executions. In this chapter, we have introduced sticky logging, a data-centric monitoring mechanism designed for distributed environments. We have presented an architecture and a prototype of the sticky logging mechanism implementing DiALog.

The sticky logging mechanism is designed to monitor the processing of specific data items and can be used to observe contractual obligations or organization-internal policies. The mechanism consists of a data structure to store the information and of a set of operations describing how to log and how to manage the logs. Apart from these, the mechanism provides methods to reconstruct a model from a log. The feasibility of sticky logging has been shown by a prototype implementation which provides an API to realize sticky logging as a generic extension of JBoss. We have proved that reconstructed executions generated by the sticky logging mechanism are sound and complete with respect to the logical execution.

The following chapter will make use of the sticky logging mechanism to provide provenance information required for the evaluation of policies. Policies are often used to regulate usage rights of data. The policy conditions may depend on the past processing of the data, such as whether pseudonyms

are used or back-ups performed. Without extra metadata describing the provenance of the data the verification may not be possible. Sticky logging is a means to provide the provenance information required to decide whether certain policy conditions are met.

We confirm Hypothesis 2 through sticky logging as it provides a formal method for monitoring the distributed data processing which is able to generate sound and complete reconstructed executions. We have published the formal methods of sticky logging at the Workshop on Privacy Enforcement and Accountability with Semantics [Ringelstein and Staab, 2007], we reflected the idea behind this approach in an article published in Datenschutz und Datensicherheit [Ringelstein, 2007], and we published the mathematical proof of the soundness and completeness quality in the Journal of Web Service Research [Ringelstein and Staab, 2010a].

6 Provenance-aware Policies for the Distributed Processing of Data

The distributed processing of data is restricted by contractual and legal requirements for protecting data and privacy as discussed in Chapter 1 and Chapter 3. In the Internet, contracts and laws are represented as policies that *"are statements of the goals for the behavior of a system"* [Hinton and Lee, 1994]. Policy conditions frequently depend on information about the environment, e.g. the receiver of a data transfer. They also can depend on information about the data, such as the subject of a health record. The needed information can be about the previous processing called provenance. Apart from other domain knowledge representing environmental parameters, provenance information exhibits temporal structures. Some policy languages support policies containing conditions based on the domain knowledge part of provenance information, e.g. XACML [Moses et al., 2005] or Rei [Kagal et al., 2003]. However, they lack a specification of how to specify or access the temporal structure of provenance. Including such temporal domain knowledge into purely declarative yet efficiently implementable languages such as Datalog constitutes a non-trivial challenge.

The *Issue of Temporal Structures* complicates policy languages to express policies based on provenance information. To overcome this issue, the temporal structure needs to be represented and queried using expressions such as *after* or *before*. Our hypothesis is that we can express history-based policy conditions by mapping the temporal structure of the histories to a graph

structure.

To target these issues, one has to collect and provide provenance information. In closed environments, the information can be collected with various existing logging mechanisms and can be accessed by various, often proprietary, solutions. As soon as processes span multiple organizations and data is transferred across organizational boundaries, one has to capture and provide the provenance information in a standardized manner (cf. *Requirement Standardized Interfaces*). For this purpose, we combine sticky logging (see Chapter 5) with the open provenance model (OPM) [Moreau et al., 2008] for providing and representing provenance data in a standardized format.

After providing the provenance, one has to extend policy conditions to relate to provenance information and its temporal structure. Based on the open provenance model, we introduce a formal language that specifies the relation between policy condition and provenance information. We specify the language by an abstract syntax based on the syntactical structure of eXtensible Access Control Markup Language (XACML). We call our approach *Papel*: Provenance-Aware Policy definition and Execution Language. In Papel, the complex temporal structure of processing histories is modeled as graph structure.

We have to address newly emerging privacy and data protection risks that arise by making provenance information accessible to policy engines (cf. *Issue of Sensitive Information*). Such mechanisms have to balance between the need to validate policy rules using provenance information and the requirement to protect privacy and data, e.g. using log encryption (see Chapter 5). Thus, the policy maker should be enabled to choose between full or limited transparency of the forwarded provenance information. For achieving such flexibility, we introduce *attributes* and *reduced facts*. These allow the policy maker to disclose only an appropriate amount of confidential information to third parties.

To tackle the above-mentioned problems, we analyze the foundations of Papel and introduce its syntax based on the foundations, in Section 6.1. We also discuss means to protect data and privacy in this section. We define

the semantics of Papel in Section 6.2, before we discuss the Datalog implementation of our abstract syntax in Section 6.3. In Section 6.4, we analyze related work in this field.

6.1 Foundations and Syntax of Papel

In this section, we describe the foundations of Papel and the abstract syntax Papel defines to express provenance information and polices rules based on provenance information. The syntax of Papel is depicted in Table 6.1[1]. We require to model provenance information and policies. To this end, we make use of concepts of the Open Provenance Model and of the eXtensible Access Control Markup Language. To integrate both we introduce an abstract syntax extending the abstract syntax of XACML.

6.1.1 About Provenance

Various monitoring mechanisms can be used to collect provenance information. We choose sticky logging (cf. Chapter 5) as it allows for attaching logs directly to the processed data and automatically forwards the logs along with the data. Other approaches may also be used, e.g. [Hallam-Baker and Behlendorf, 1996].

The collected provenance information must be shared in an agreed format. One standard for provenance information is the *Open Provenance Model (OPM)* [Moreau et al., 2008] that represents provenance of data processing as graph structure. OPM defines the term *processing* as an *"action or series of actions performed on or caused by artifacts, and resulting in new artifacts"* [Moreau et al., 2008]. In Papel, we use primitives to define provenance information (`Primitive` is the start symbol of the provenance part of the Papel syntax; see Table 6.1). Papel represents the graph structure of OPM by the `step` primitive, which represents single processing steps (see

Table 6.1: Syntax of Papel.[1]

Papel Syntax for Provenance Information:

Syntax Element	EBNF syntax
Primitive	Step | ReducedFact | Attribute ;
Step	"step ("Data", "Actors", "InvolvedAgents", "Category", "Purpose", "ID", "PIDs")." ;
ReducedFact	"reduced ("Data", "(Actors | "hidden")", "(InvolvedAgents | "hidden")", "(Category | "hidden")", "(Purpose | "hidden")", "ID", "PIDs")." ;
Attribute	"attribute ("Data", "Name", "Value", "ID")." ;

Papel Syntax for Policies:

Syntax Element	EBNF syntax
Rule	Permission | Restriction | Assignment ;
Permission	"permit (ID) IF " Condition "." ;
Restriction	"deny (ID) IF " Condition "." ;
Assignment	"assignment (ID) IF " Condition "DO" SetAttribute | SetReducedFact "." ;
Condition	Primitive | ("(NOT" Primitive | Condition | "permit (ID)" | "deny (ID)" ")") | ("(" Primitive |Condition | "permit (ID)" | "deny (ID)" BooleanOperator Primitive | Condition | "permit (ID)" | "deny (ID)" ")") | (Step | ReducedFact "AFTER" Step | ReducedFact ")") ;
SetAttribute	"set_attribute ("Data", "Name", "Value", "ID")." ;
SetReducedFact	"set_reduced ("Data", "(Actors | "hidden")", "(InvolvedAgents | "hidden")", "(Category | "hidden")", "(Purpose | "hidden")", "ID", "PIDs")." ;
BooleanOperator	"AND" | "OR" | "XOR" ;

Definition 6.1), and the connection of these steps.

Definition 6.1: *Processing Step*

We define the representation of a processing step as a 7-tuple with the following syntax:

```
step (Data, Actors, InvolvedAgents, Category,
      Purpose, ID, PIDs)
```

where:
- `Data` refers to the *artifact* processed during the execution of the processing step,
- `Actors` are the *agents* controlling the processing step,
- `InvolvedAgents` are *agents* involved in the processing step that did not trigger the processing,
- `Category` is the category of the processing step,
- `Purpose` is the purpose of executing the processing step,
- `ID` is the unique *identifier* of the processing step, and
- `PIDs` are the unique *identifiers* of the directly preceding processing steps.

In OPM an *artifact* is defined as an *"immutable piece of state, which may have [..] a digital representation in a computer system"* [Moreau et al., 2008] and an *agent* is defined as a *"contextual entity acting as a catalyst of a process, enabling, facilitating, controlling, affecting its execution"* [Moreau et al., 2008]. An example of an involved agent in our running example is the University of Koblenz as receiver of the transfer of health records. The unique identifiers are specified as in OPM.

From the relationships between the attributes `ID` and `PIDs` one can easily derive a partial order $\leq_S$ over the set of processing steps. The partial order constitutes a graph structure modeling the temporal structure of the pro-

[1]The syntax of literals, such as `Name`, `Category`, etc., is not specified in the table.

cessing history, whereby some actions may occur concurrently on different computer systems. Besides these constituents of OPM, the `step` primitive also specifies the `Category` and `Purpose` of a processing step. The possible categories and the possible purposes are defined in a domain specific ontology.

Example 6.1: *Example of a Processing Step.*

As depicted in Figure 3.1, the first processing step is the creation of Jane Doe's health record by the hospital. The purpose of creating the record is the treatment of Jane Doe:

```
step (record_JD, {mrh}, {}, create, treatment,
      1, {0})
```

This example and the following examples make use of logical constants `record_JD`, `mrh`, `create` and `patient_treatment`, which must be defined in domain ontologies and corresponding databases. In the scenario, these are provided by the Middle Rhine Hospital. For readability of the run-through example, we use integers to denote the identifiers though in general such a simplification is not possible.

We use the `step` primitive and the `reduced` primitive (see Definition 6.6) to specify the processing provenance and the `attribute` primitive (see Definition 6.4) to specify attributes and their value (see Table 6.1). The information about the single processing steps, attribute assignments and reduced facts are collected during the execution of a process and represents the processing history.

Definition 6.2: *Processing History*

Let S be the set of all processing steps and let A be the set of all attribute assignments and let R be the set of all reduced facts, a history H is defined

as a set of processing steps, attribute assignments and reduced facts:

$$H \subseteq S \cup A \cup R$$

Over each of the three sets S, A, and R a partial order is defined ($\leq_S$, $\leq_A$, and $\leq_R$). The transitive closure of the union of these order relations defines the global partial order $\leq_H$ over the elements of H ($\leq_H = (\leq_S \cup \leq_A \cup \leq_R)^*$).

Not all entries of the history may be transferred to all other agents. A system owner may decide to disclose only partial provenance information to subsequent policy engines in the processing line in order to protect data or privacy. The reader will find more details on this in the corresponding section 6.1.5, where we also define attributes and reduced facts in detail. Next, we explain the core of Papel, i.e. the application of policies on fully transparent provenance information.

6.1.2 About Policies

To specify policies for inter-organizational exchange of data, Papel uses an existing policy language as starting point. Papel builds on concepts from the *eXtensible Access Control Markup Language (XACML)* [Moses et al., 2005], which is a standard defining an XML based policy framework. XACML supports three policy elements, i.e. permission (*permit*), restriction (*deny*) and obligation (cf. Chapter 7). XACML policies define rules by connecting a set of subjects (actors) with a set of targets (data) and by specifying the conditions of the rule. If the conditions are fulfilled, XACML rules will result in a given effect, which is *permit* or *deny*.

In Papel, we also use rules to specify policies (`Rule` is the start symbol of the policy part of the Papel syntax, cf. Table 6.1). Policy rules (see Definition 6.3) are provided by the person concerned or rights owner. Papel models permissions as rules specifying which sorts of processing steps are permitted.

Likewise, restrictions are modeled as rules specifying which processing steps are denied. Permissions are specified by means of `permit` rules and restrictions by means of `deny` rules. The rules consist of the name, which indicates the type of policy (`permit` or `deny`), and the body of the rule, which defines the conditions (see Section 6.1.3), and is specified after the `IF` statement.

Definition 6.3: *Policy Rules*

Policy rules are defined as the functions $permit : \Phi \rightarrow \{true,\ false\}$ and $deny : \Phi \rightarrow \{true,\ false\}$, where Φ is the set of all identifiers. The rules have the following syntax:

```
permit (ID) IF Condition.
deny (ID) IF Condition.
```

where:
- `ID` is the parameter to pass the identifier of a processing step that will be checked if it is permitted or denied, and
- `Condition` is the logical expression specifying whether a processing step is permitted or denied.

The parameter `ID` specifies the processing step that will be checked whether it is permitted or denied (e.g. `permit (3)` will check if the step with the identifier `3` is permitted). By using the variable parameter `ID`, the policy engine of an agent may match the condition of a rule against elements from the history known by this agent.

The semantics of Papel define that all processing steps will be denied if not explicitly permitted. Denial can be used to overwrite or restrict the explicit permissions. Conflicting `permit` and `deny` rules have to be interpreted as denial overwriting permissions (see Example 6.2).

Papel specifies a third type of rules that we call `assignment` rules.

As `permit` and `deny` rules, `assignment` rules may depend on a `condition` and have the parameter `ID`, which refers to the current processing step. Assignment rules specify the change of an attribute value by use of the `set_attribute` primitive. The `assignment` rule and `set_attribute` primitive are discussed in detail in Section 6.1.5.

6.1.3 Condition Statements

A permit or deny rule will be effective if its condition is true. The intuitive semantics of a condition is that it will be true for the corresponding `ID` if the corresponding step, reduced fact or attribute assignment matches the constraints formulated in the condition. In Papel, conditions can be composed using the following logical operators: `NOT`, `AND`, `OR`, `XOR` and `AFTER` (see Example 6.5). Parentheses are used to specify the interpretation order of complex statements. By the `AFTER` operator, one can access the partial order of the processing steps in the history and relate them to the status of the system at a particular point in time (in the subsequent section the reader will see a detailed example).

Example 6.2: *Policy 1 in Papel*

`Policy 1` is created in step 1 of the scenario (see Section 3.1) after Jane Doe is admitted as a patient. This example formalizes `Policy 1` as a set of three rules:

The Middle Rhine Hospital is allowed to use Jane Doe's health record to treat her as a patient.

```
permit (ID) IF step (record_JD, {mrh}, _, _,
                     treatment, ID, _).
```

They are not allowed to use it for any other purpose nor are they allowed to

transfer her health record to any other organization or entity.

```
deny (ID) IF step (record_JD, {mrh}, _, _,
                   Purpose, ID, _)
             AND NOT (Purpose = treatment).

deny (ID) IF step (record_JD, {mrh}, _, transfer,
                   _, ID, _).
```

In the example we make use of the logical constant `mrh`, which represents the Middle Rhine Hospital. The `Data` parameter of the `step` primitive is set to `record_JD`, which is the unique identifier of Jane Doe's health record. Thus, the three rules of `Policy 1` are directly connected to the health record of Jane Doe.

The example uses unnamed variables indicated by _. The _ represents another unnamed variable each time it is used. An unnamed variable matches all possible values of its type.

The first rule of the example permits the processing of Jane Doe's health record by the Middle Rhine Hospital for treatment purposes. The second rule explicitly denies all processing steps that do not have the purpose of treating Jane Doe as a patient. In Papel, restrictions override permissions (see Section 6.2.5). The third rule of this example denies the transfer of `record_JD`, even if it is for research or treatment purposes, and restricts the permission of the first rule.

Example 6.3: *Policy 3 in Papel*

This example depicts `Policy 3` that allows the Middle Rhine Hospital to use Jane Doe's health record for research purposes.

```
permit (ID) IF step (record_JD, {mrh}, _, _,
```

```
                    research, ID, _).
```

This rule still conflicts with the second rule of `Policy 1` as implemented in Example 6.2. So, we change this policy rule to:

```
deny (ID) IF step (record_JD, {mrh}, _, _, Purpose,
                   ID, _)
             AND NOT (Purpose = treatment)
             AND NOT (permit(ID)).
```

In some instances a restriction should be overridden by a permission as in aboves example. This can be achieved by adding `AND NOT permit (ID)` or more complex statements to the condition, which define exceptions.

6.1.4 Connecting Provenance and Policies

Based on the modeling of provenance information and the specification of policies that disregard provenance information, one may combine both to specify policies that take provenance information into account. To demonstrate the connection between policies and provenance information in Papel, we discuss the following specification of policy `Policy 6`:

Example 6.4: *Policy 6 in Papel*

The following example depicts the implementation of `Policy 6` of the running example. *The patient demands that her health record is de-identified before it is transfered:*

```
permit (ID) IF step (record_JD, _, _, transfer, _,
                     ID, {PID})
```

```
        AND step (record_JD, _, _, update,
                  de-identify, PID, _).
```

We have used the `step` primitive before (see Example 6.2) to relate to the actual processing step identified by `ID`. However, in Example 6.4 we use the `step` primitive to address provenance information, as well. Permission will be granted only if the provenance information contains the description of a matching step. In Section 6.2, a detailed definition of the semantics of Papel is given.

The second step primitive refers to a preceding processing step. By using the variable `PID`, we can specify that the de-identification step has to be performed *directly* before the processing step to be permitted. If the variable is not used, any de-identification step performed at anytime will fulfill the condition.

Using the `AFTER` operation allows us to address the temporal structure of the history in a more flexible manner. `a AFTER b` specifies that the element `a` of the known history has to occur after the element `b` of the known history as specified by the partial order ($\leq_H$). In difference to Example 6.4, the policy in Example 6.5 does not require that the de-identification must directly precede the data transfer.

Example 6.5: *Policy 6 using the AFTER Operator*

The patient demands that her health record is transferred only after it has been de-identified:

```
permit (ID) IF (step (record_JD, _, _, transfer, _,
                      ID, _)
            AFTER step (record_JD, _, _, update,
                        de-identify, _, _)).
```

In this example, the first line of the policy refers to the current process-

ing step referred to by the variable `ID`. The processing step is performed on `record_JD`. Through the combination by the `AFTER` operator, the second part in the example is referring to a step preceding `ID`, which is also performed on `record_JD`. The permission will be granted only if the provenance information contains the description of a matching step.

This example policy formulation would still allow for the possibility that `record_JD` was first de-identified, then re-identified and finally transferred. If one wanted to rule out transfer steps in such a case, one would have to add an explicit deny rule or query a variable status as allowed by the formulation of policies with attributes (see Section 6.1.5).

As the following example depicts, by the `AFTER` operator, we can access the partial order of longer paths of processing steps in the history.

Example 6.6: *Policy 5 in Papel*

The following example illustrates the implementation of `Policy 5`. *Permit the transfer after the confirmation of the access approval:*

```
permit (ID) IF (step (record_JD, {ukob}, _, access,
                      _, ID, _)
            AFTER (step (record_JD, {mrh}, _, _,
                         confirmation, _, _)
            AFTER step (record_JD, {patient}, _, _,
                        access_approval, _, _))).
```

6.1.5 Data and Privacy Protection

The provenance information can be used as source of information about the processing history. However, it may contain sensitive data, such as the provenance information about adjusting Jane Doe's cancer medication. While full

encryption of the provenance information may render policy execution impossible, partial encryptions may trade-off between the need for checking policy compliance and security concerns (see the *Contractual Requirement Accessibility*). To overcome these issues, Papel introduces *attributes* as well as *reduced facts*.

Attributes are used to specify provenance information that can be expressed by a value, e.g. de-identification status or modification counter. They can be used to carry an attribute value along a set of processing steps in the history until the attribute value is changed again.

Definition 6.4: *Attributes*

We define the representation of an attribute as a 4-tuple with the following syntax:

```
attribute (Data, Name, Value, ID)
```

where:
- `Data` refers to the data instance the attribute belongs to,
- `Name` is the name of the attribute,
- `Value` is its assigned value,
- `ID` identifies the processing step.

Likewise, we define the representation of the set attribute predicate as a 4-tuple with the following syntax:

```
set_attribute (Data, Name, Value, ID)
```

where `Data`, `Name`, `Value` and `ID` are specified as before.

The value of the attribute is assigned by the actor performing the processing step. The assignment is done according to the `assignment` rules (see

Definition 6.5) defined by the creator of the policies. Each attribute has exactly one specific value at a time, possibly undefined. The latest setting of the attribute preceding a specific point in time specifies its value. If the attribute `de-identified` is set to `true` in step 2 and not set again until step 6, it will still have the value `true` in step 6. If an attribute is set concurrently to different values, its value will be undefined at the point in time when the two concurrent histories merge again.

Definition 6.5: *Assignment Rules*

We represent assignment rules by the following syntax:

```
assignment (ID) IF Condition DO Assignment.
```

where:
- `ID` is the parameter to pass the identifier of a processing step that triggers the assignment,
- `Condition` is the condition specifying whether an assignment is triggered or not, and
- `Assignment` is the assignment specifying the setting of attributes and reduced facts.

The following example depicts the use of attributes and assignment rules.

Example 6.7: *Policy 6 with Attributes*

This example illustrates the permit rule and the assignment rule required to implement `Policy 6` using the attribute `de-identified` assigned to the `record_JD`. The policy rule of this example will grant the transfer of `record_JD` only, if it is not re-identified between the de-identification and the transfer steps: *The patient demands that her health record will be transferred only if it is de-identified at the time of transfer:*

```
permit(ID) IF (step (record_JD, _, _, transfer, _,
                     ID, _) AND
           attribute (record_JD, de-identified,
                      true, ID)).
```

If the executed processing step de-identifies the record, set the attribute named `de-identified` *to* `true`*, and if the record is re-identified, set the attribute to* `false`*:*

```
assignment(ID) IF step (record_JD, _, _, _,
                        de-identify, ID, _)
               DO set_attribute (record_JD,
                                 de-identified,
                                 true, ID).

assignment(ID) IF step (record_JD, _, _, _,
                        re-identify, ID, _)
               DO set_attribute (record_JD,
                                 de-identified,
                                 false, ID).
```

Attributes may not be expressive enough to provide all required information, e.g. to answer the question whether the last de-identification has been performed by a specific actor. In this case, someone who performs logging can use reduced facts in Papel. A reduced fact is a step description that is reduced to the necessary and contains only the required information. Differently than the full description, the information conveyed by the reduced fact must not be encrypted to not hamper querying the information.

Definition 6.6: *Reduced Facts*

We define the representation of a reduced fact as a 7-tuple with the following

syntax:

```
reduced (Data, Actors, InvolvedAgents, Category,
         Purpose, ID, PIDs)
```

where:

- `Data` refers to the *artifact* processed during the execution of the processing step,
- `Actors` are the *agents* controlling the processing step,
- `InvolvedAgents` are *agents* that did not trigger the processing step, but are involved,
- `Category` is the category of the processing step,
- `Purpose` is the purpose of executing the processing step,
- `ID` is the unique *identifier* of the processing step, and
- `PIDs` are the unique *identifiers* of the directly preceding processing steps.

Likewise, we define the `set_reduced` primitive of as a 7-tuple with the following syntax:

```
set_reduced (Data, Actors, InvolvedAgents,
             Category, Purpose, ID, PIDs)
```

where `Data`, `Actors`, `InvolvedAgents`, `Category`, `Purpose`, `ID` and `PIDs` are specified as for the reduced primitive.

To protect data or privacy, the parameters `Data`, `Actors`, `InvolvedAgents`, `Category`, and `Purpose` can be set selectively to the special zero value `hidden`. The parameters `ID` and `PIDs` must not be set to `hidden`. These two parameters are required to reproduce the (partial) order of processing steps.

Example 6.8: *Reduced Facts*

The following example depicts the provenance information of a processing step of the department for *nuclear medicine* (Step 4 in our scenario): *The processing step updates the health record by adding a new cancer medication:*

```
step (record_JD, {nuclear_medicine}, {},
      update, new_cancer_medication, 3, {2})
```

If this information is not relevant to the analysis of the University of Koblenz, the Middle Rhine Hospital will encrypt the original information about the processing step and will provide reduced provenance information fulfilling `Policy 7`:

```
reduced (record_JD, hidden, hidden, update, hidden,
         3, {2})
```

6.2 Execution Semantics of Papel

In this section, we define the semantics of Papel. The semantics of Papel specifies whether a given execution step violates a given set of policy rules under consideration of a given history. The interpretation of a policy rule in Papel shall be true, if and only if the condition of the rule is true. We define a Tarskian semantics mapping syntactic elements of Papel onto subsets and relations over a universe U. Based on these semantics we define when a set of policies is fulfilled under a given history H (see Definition 6.2). First we define minimal models of a history H:

Definition 6.7: *A Minimal Model of a History*

We define a minimal model M of a history H as a model to which no strictly smaller Herbrand model of the history H exists [Lloyd, 1993].

A minimal model of a history is only a model of the history and of subparts of the history. Based on the definition of a minimal model of a history we define when a set of policies is fulfilled:

Definition 6.8: *Fulfilling a Set of Policy Rules*

We define that a set of policy rules R is fulfilled with respect to a history H if each minimal model M of the history H is also a model of the set of policy rules R.

6.2.1 The Basic Semantics

In Papel, the universe U is the set:

$$U = \Delta \cup \Gamma \cup X \cup \Psi \cup \Phi \cup N \cup V,$$

where Δ is the set of artifacts (e.g. data instances), Γ is the set of agents (including actors), X is the set of categories, Ψ is the set of purposes, Φ is the set of processing step identifiers, N is the set of attribute identifiers, and V the set of attribute values. These subsets of U are mutually disjoint. For the following definitions be $P(Z)$ the power set of the set Z.

Definition 6.9: *The Partial Interpretation Function I*

We define I to map atomic elements of Papel onto the (parts of) our universe U as follows:

$$\texttt{Data}^I \in \Delta,$$
$$\texttt{Actors}^I \subseteq \Gamma,$$
$$\texttt{InvolvedAgents}^I \subseteq \Gamma,$$
$$\texttt{Category}^I \in X,$$
$$\texttt{Purpose}^I \in \Psi,$$
$$\texttt{ID}^I \in \Phi,$$
$$\texttt{PIDs}^I \subseteq \Phi$$
$$\texttt{Name}^I \in N, \text{ and}$$
$$\texttt{Value}^I \in V.$$

The predicate representing the performed processing steps $step$, which are elements of the history H, is interpreted as $step^I \subseteq U^7$:

$$step^I \subseteq \Delta \times P(\Gamma) \times P(\Gamma) \times X \times \Psi \times \Phi \times P(\Phi)$$

the predicate representing the reduced facts $reduced$, which are elements of the history H, is interpreted as $reduced^I \subseteq U^7$:

$$reduced^I \subseteq \Delta \times P(\Gamma) \times P(\Gamma) \times X \times \Psi \times \Phi \times P(\Phi)$$

and the predicate representing the attribute assignments $attribute$, which are elements of the history H, is interpreted as $attribute^I \subseteq U^5$:

$$attribute^I \subseteq \Delta \times N \times V \times \Phi \times P(\Phi).$$

The partial interpretation function I must satisfy the following constraints:

- Each identifier $\phi \in \Phi$ must clearly identify one processing step (injective function from ϕ to $step$):

  $\forall \phi \in \Phi : (\delta_1, \alpha_1, \beta_1, \chi_1, \psi_1, \phi, \rho_1), (\delta_2, \alpha_2, \beta_2, \chi_2, \psi_2, \phi, \rho_2) \in step^I \Rightarrow \delta_1 = \delta_2 \wedge \alpha_1 = \alpha_2 \wedge \beta_1 = \beta_2 \wedge \chi_1 = \chi_2 \wedge \psi_1 = \psi_2 \wedge \rho_1 = \rho_2.$

- Likewise, at each processing step specified by $\phi \in \Phi$ each attribute has exactly one value at a time:

$$\forall \phi \in \Phi : (\delta,\ \mu,\ \nu_1,\ \phi,\ \rho_1), (\delta,\ \mu,\ \nu_2,\ \phi,\ \rho_2) \in attribute^I \Rightarrow \nu_1 = \nu_2 \wedge \rho_1 = \rho_2.$$

The processing history is defined as partial order $\geq \subseteq \Phi \times \Phi$ of processing steps, reduced facts, and attribute assignments. A tuple $(\phi,\ \eta)$ is element of $\geq$, if the processing step specified by η precedes the processing step specified by ϕ or if $\eta = \phi$. Before we define $\geq$, we define the relation $\succ \subseteq \Phi \times \Phi$ of directly preceding processing steps, reduced facts, and attribute assignments and the relation $> \subseteq \Phi \times \Phi$ of strictly preceding processing steps.

Definition 6.10: *Immediately Preceding Processing Steps*

We define a processing step s_ζ as immediately preceding another processing step s_ϕ if s_ϕ is the successor of s_ζ:

$$\succ = \{(\phi,\ \zeta) | \exists (\delta_s,\ \alpha_s,\ \beta_s,\ \chi_s,\ \psi_s,\ \phi,\ \rho_s) \in step^I \wedge \zeta \in \rho_s \vee \exists ((\delta_a, \mu_a,\ \nu_a,\ \phi,\ \rho_a) \in attribute^I \wedge \zeta \in \rho_a\}.$$

A processing step precedes another processing step, if a trace of consecutive processing steps exists that connects both steps:

Definition 6.11: *Strict Partial Order of Processing Steps*

We define the strict partial order $> \subseteq \Phi \times \Phi$ of processing steps, reduced facts and attribute assignments as:

$$> = \{(\phi,\ \eta) | (\phi \succ\ \eta) \vee \exists \zeta_i \in \Phi :\ (\phi \succ\ \zeta_i) \wedge (\zeta_i \succ\ \eta) \vee \exists \zeta_1, .., \zeta_n \in \Phi :\ (\phi \succ\ \zeta_1) \wedge (\zeta_1 \succ\ \zeta_2) \wedge .. \wedge (\zeta_n \succ\ \eta)\}.$$

Finally, we define the (non-strict) partial order of processing steps.

Definition 6.12: *Partial Order of Processing Steps*

We define the partial order $\geq \subseteq \Phi \times \Phi$ of processing steps, reduced facts and attribute assignments as:

$$\geq = \{(\phi,\ \eta) | (\phi = \eta) \vee (\phi > \eta)\}.$$

6.2.2 Logical expressions

In Section 6.1.3, we introduced the following syntactical elements `NOT`, `AND`, `OR`, and `XOR`. Their semantics are defined in the same manner as the corresponding boolean expressions (see Table 6.2).

Definition 6.13: *The Relations: 'not', 'and', 'or' and 'xor'*

Let e be a logical expression, we define the not-relation as $not \subseteq \{true,\ false\}$, where:

$$(not(e))^I = \begin{cases} \text{true} & \text{if } e^I = false, \\ \text{false} & \text{else.} \end{cases}$$

Let e and f be logical expressions, we define the and-relation as $and \subseteq \{true,\ false\}$, where:

$$(and(e, f))^I = \begin{cases} \text{true} & \text{if } e^I \wedge f^I = true, \\ \text{false} & \text{else.} \end{cases}$$

Let e and f be logical expressions, we define the or-relation as $or \subseteq \{true,\ false\}$, where:

$$(or(e,f))^I = \begin{cases} \text{true} & \text{if } e^I \vee f^I = true, \\ \text{false} & \text{else.} \end{cases}$$

Let e and f be logical expressions, we define the xor-relation as $xor \subseteq \{true,\ false\}$, where:

$$(xor(e,f))^I = \begin{cases} \text{true} & \text{if } (e^I \wedge \neg f^I) \vee (\neg e^I \wedge f^I) = true, \\ \text{false} & \text{else.} \end{cases}$$

We introduce the syntactical element `AFTER` to address the temporal structure of the history. The `AFTER` statement is defined by the following relation:

Definition 6.14: *The after-Relation*

Let s_1 and s_2 be processing steps or reduced facts, we define the $after$-relation as $after^I \subseteq (step^I \cup reduced^I) \times (step^I \cup reduced^I)$, where:

$$(after(s_1,s_2))^I = \begin{cases} \text{true} & \text{if } s_1 = (..,id_1,..) \wedge s_2 = (..,id_2,..) \wedge \\ & (id_1^I > id_2^I), \\ \text{false} & \text{else.} \end{cases}$$

6.2.3 Processing Steps, Reduced Facts and Attributes

In the following, we extend the interpretation function I (see Definition 6.9) by the interpretation of non-atomic syntactic expressions of Papel.

A processing step is specified by the *step* predicate. The following definition specifies how the *step* predicate is interpreted by the interpretation function I.

Table 6.2: Condition Statements.

Syntax	Natural language semantics
`A`	A is true
`NOT A`	A is not true
`A AND B`	A and B are true
`A OR B`	A or B (non-exclusive or) are true
`A XOR B`	either A or B (exclusive or) is true
`B AFTER A`	first A and then B (in the given order[2]) are true
`IF Condition`	if the $Condition$ is fulfilled

Definition 6.15: *Interpretation of the Step Predicate*

Let I be the partial interpretation function, let _ be an unspecified parameter, let S be a substitution of variables to Papel terms, let e be the logical expression in which the $step$ predicate occurs and let $(eS)^I$ be $true$, the interpretation of the $step$ predicate is defined by the function $(step)^I : U^7 \rightarrow \{true, false\}$:

$$
\begin{array}{l}
(step(\delta,\ \alpha,\ \beta,\ \chi,\ \psi,\ \phi,\ \rho))^I = \\
\begin{cases}
\text{true} & \text{if } \exists(\delta',\ \alpha',\ \beta',\ \chi',\ \psi',\ \phi',\ \rho') \in step^I : \\
 & (\delta^I = \delta' \vee \delta = \_ \vee \{\delta \mapsto \delta'\} \subseteq S) \wedge \\
 & (\alpha^I = \alpha' \vee \alpha = \_ \vee \{\alpha \mapsto \alpha'\} \subseteq S) \wedge \\
 & (\chi^I = \chi' \vee \chi = \_ \vee \{\chi \mapsto \chi'\} \subseteq S) \wedge \\
 & (\psi^I = \psi' \vee \psi = \_ \vee \{\psi \mapsto \psi'\} \subseteq S) \wedge \\
 & (\phi^I = \phi' \vee \phi = \_ \vee \{\phi \mapsto \phi'\} \subseteq S) \wedge \\
 & (\rho^I = \rho' \vee \rho = \_ \vee \{\rho \mapsto \rho'\} \subseteq S), \\
\text{false} & \text{else.}
\end{cases}
\end{array}
$$

[2]The processing history can be represented as a directed graph. Thus, processing steps can be in order if they can be connected by a path in the graph.

We distinguish three different sorts of terms specifying parameters of $step$ predicates. These are named identifiers (e.g. `record_JD`), unspecified parameters (_) and variables (e.g. `ID`). In the first case, the interpretation of the term must equal the regarding element of the step $s' \in step^I$ (e.g. $\texttt{record\_JD}^I = \delta'$). In the second case, each element of the associated type will match. In the last case, a substitution S must exist that substitutes the variable specified by the term with the regarding element of the step (e.g. $\{\texttt{ID} \mapsto \phi'\} \subseteq S$). After performing the substitution, the interpretation of the logical expression e, in which the $step$ predicate occurs, must be $true$.

Similar to the $step$ predicate, the $reduced$ predicate is interpreted by the interpretation function I as defined by the following definition.

Definition 6.16: *Interpretation of the Reduced Fact Predicate*

Let I be the partial interpretation function, let S be a substitution of variables to Papel terms, let e be the logical expression in which the $reduced$ predicate occurs and let $(eS)^I$ be $true$, the interpretation of the $reduced$ predicate is defined by the function $(reduced)^I : U^7 \rightarrow \{true, false\}$:

$$
\begin{array}{ll}
 & (reduced(\delta,\ \alpha,\ \beta,\ \chi,\ \psi,\ \phi,\ \rho))^I = \\
\left\{\begin{array}{l} \text{true} \\ \\ \\ \\ \\ \\ \\ \\ \text{false} \end{array}\right. & \begin{array}{l} \text{if } \exists(\delta_s,\ \alpha_s,\ \beta_s,\ \chi_s,\ \psi_s,\ \phi_s, \rho_s) \in step^I : \\ (\delta^I = \delta_s \vee \delta = hidden\ \vee\ \{\delta \mapsto \delta_s\} \subseteq S) \wedge \\ (\alpha^I = \alpha_s \vee \alpha = hidden\ \vee\ \{\alpha \mapsto \alpha_s\} \subseteq S) \wedge \\ (\chi^I = \chi_s \vee \chi = hidden\ \vee\ \{\chi \mapsto \chi_s\} \subseteq S) \wedge \\ (\psi^I = \psi_s \vee \psi = hidden\ \vee\ \{\psi \mapsto \psi_s\} \subseteq S) \wedge \\ (\phi^I = \phi_s \vee \{\phi \mapsto \phi_s\} \subseteq S) \wedge \\ (\rho^I = \rho_s \vee \{\rho \mapsto \rho_s\} \subseteq S)\ , \\ \text{else.} \end{array}
\end{array}
$$

In difference to processing steps, the reduced facts must be explicitly set.

Apart the interpretation of the $reduced$ predicate, we require to define the interpretation of the $set\_reduced$ predicate that is used to specify the setting of reduced facts.

Definition 6.17: *Interpretation of the Setting Predicate for Reduced Facts*

Let I be the partial interpretation function, let $(\delta_s,\ \alpha_s,\ \beta_s,\ \chi_s,\ \psi_s,\ \phi_s,\ \rho_s) \in step^I$ be the processing step described by the reduced fact, be $\phi_r^I = \phi_s$ and $\rho_r^I = \rho_s$, and be $\delta_r^I = \delta_s \vee \delta_r = hidden$, $\alpha_r^I = \alpha_s \vee \alpha_r = hidden$, $\beta_r^I = \beta_s \vee \beta_r = hidden$, $\chi_r^I = \chi_s \vee \chi_r = hidden$, and $\psi_r^I = \psi_s \vee \psi_r = hidden$. We define the interpretation of the $set\_reduced$ predicate by the following function $(set\_reduced)^I : U^7 \rightarrow \{true, false\}$:

$$(set\_reduced(\delta_r,\ \alpha_r,\ \beta_r,\ \chi_r,\ \psi_r,\ \phi_r,\ \rho_r))^I = \begin{cases} \text{true} & \text{if } (\delta_r^I,\ \alpha_r^I,\ \beta_r^I,\ \chi_r^I,\ \psi_r^I, \phi_r^I,\ \rho_r^I) \in reduced^I, \\ \text{false} & \text{else.} \end{cases}$$

An attribute is specified by the $attribute$ predicate. The following definition specifies interpretation of this predicate by the interpretation function I.

Definition 6.18: *Interpretation of Attribute Predicate*

Let I be the partial interpretation function, we define the interpretation of the $attribute$ relation by the following function $(attribute)^I : U^5 \rightarrow \{true, false\}$:

$$(attribute(\delta, n, v, \phi, \rho))^I =$$

$$\begin{cases} \text{true} & \text{if } (\delta^I, n^I, v^I, \phi^I, \rho^I) \in attribute^I, \\ \text{true} & \text{if } (\delta^I, n^I, v^I, \phi^I, \rho^I) \notin attribute^I \wedge \\ & \exists(\delta^I, n^I, v^I, \phi_l, \rho_l) \in attribute^I \wedge (\phi^I > \phi_l) \wedge \\ & \neg\exists(\delta_m, n_m, v_m, \phi_m, \rho_m) \in attribute^I : \\ & (\phi^I > \phi_m) \wedge\ (\phi_m > \phi_l), \\ \text{false} & \text{else.} \end{cases}$$

An attribute has the value assigned in the current step, or if no value is assigned in the current step, it will have the value that has been assigned last. The assignment of an attribute is specified by the $set\_attribute$ predicate. The interpretation of this predicate is defined as follows:

Definition 6.19: *Interpretation of the Setting Predicate of Attributes*

Let I be the partial interpretation function, we define that the $set\_attribute$ predicate is interpreted by the following function $(set\_attribute)^I : U^5 \rightarrow \{true, false\}$:

$$(set\_attribute(\delta, n, v, \phi, \rho))^I =$$
$$\begin{cases} \text{true} & \text{if } \forall \eta \in \Phi :\ \eta \succ \phi^I \rightarrow (\delta^I, n^I, v^I, \eta, \{\phi^I\}) \in attribute^I, \\ \text{false} & \text{else.} \end{cases}$$

The result of the interpretation of the $set\_attribute$ predicate will be true if the value of the attribute is updated in all directly succeeding processing steps.

6.2.4 Permission, restriction and assignment

Permissions, restrictions, and assignments are defined as predicates on Φ. The predicates are defined by the functions $permit : \Phi \rightarrow \{true,\ false\}$, $deny : \Phi \rightarrow \{true,\ false\}$, and $assignment : \Phi \rightarrow \{true,\ false\}$. We define these functions and their interpretation in the following.

Permit rules specify processing steps that are allowed to be performed. We define the $permit$ function in a way that it is $true$ for all steps that are permitted and $false$ for all steps that are not permitted. Considering only permit rules, 'not permitted' does not imply that a rule is denied. The interpretation order of Papel specifies that all steps, which are not explicitly permitted, are considered as denied.

Definition 6.20: *Permit Function*

We define $permit : \Phi \rightarrow \{true,\ false\}$ as a function that will be $true$ if the processing step identified by ID is permitted. If it is not permitted, it will be $false$.

The interpretation of the permit predicate is specified by the following definition:

Definition 6.21: *Interpretation of Permit Predicate*

Let $p_1,\ p_2,\ ..,\ p_n$ be the logical expressions specifying the conditions of all rules defining permissions. The predicate $permit$ is interpreted as follows:

$$(permit(ID))^I = \begin{cases} \text{true} & \text{if } \exists\ (\delta,\ \alpha,\ \beta,\ \chi,\ \psi,\ \phi,\ \rho) \in step^I : \\ & ID^I = \ \phi \wedge (p_1^I \vee p_2^I \vee .. \vee p_n^I) = true \\ \text{false} & \text{else} \end{cases}$$

Prohibited processing steps are specified by deny rules. So, we define the $deny$ function to be $true$ for all denied steps and $false$ for all not denied steps. Considering only deny rules, 'not denied' does not imply that a rule is permitted. And the interpretation order of Papel specifies that all steps, which should be permitted, must be explicitly permitted.

Definition 6.22: *Deny Function*

$deny : \Phi \rightarrow \{true,\ false\}$ is a function that will be $true$ if the processing step identified by ID is denied. If it is not denied, it will be $false$.

The interpretation of the $deny$ predicate is defined as follows:

Definition 6.23: *Interpretation of Deny Predicate*

Let $d_1,\ d_2,\ ..,\ d_m$ be the logical expressions specifying the conditions of all rules defining restrictions, $deny$ is interpreted as follows:

$$(deny(ID))^I = \begin{cases} \text{true} & \text{if } \exists\ (\delta,\ \alpha,\ \beta,\ \chi,\ \psi,\ \phi,\ \rho) \in step^I : \\ & ID^I = \ \phi \wedge\ (d_1^I \vee d_2^I \vee .. \vee d_m^I) = true \\ \text{false} & \text{else} \end{cases}$$

The setting of attributes is specified by assignment rules. But, the assignment rules do not set the attributes. We define the $assignment$ function to be $true$ for all processing steps that do not violate assignment rules and $false$ for all steps violating assignment rules.

Definition 6.24: *Assignment Function*

$assignment : \Phi \rightarrow \{true,\ false\}$ is a function that will be $true$ if the processing step identified by ID does not violate an assignment. If it violates

an assignment, it will be $false$.

The interpretation of the $assignment$ predicate is defined as follows:

Definition 6.25: *Interpretation of Assignment Predicate*

Let $c_1,\ c_2,\ ..,\ c_k$ be the logical expressions specifying the conditions of all rules defining assignments and be $a_1,\ a_2,\ ..,\ a_k$ the $set\_attribute$ and $set\_reduced$ predicates of all rules defining assignments, the predicate $assignment$ is interpreted as follows:

$$(assignment(ID))^I = \begin{cases} \text{true} & \text{if } \exists\ (\delta,\ \alpha,\ \beta,\ \chi,\ \psi,\ \phi,\ \rho) \in step^I : ID^I = \ \phi \wedge \\ & ((\neg c_1^I \vee a_1^I) \wedge (\neg c_2^I \vee a_2^I) \wedge .. \wedge (\neg c_k^I \vee a_k^I)) = true \\ \text{false} & \text{else} \end{cases}$$

6.2.5 Fulfilling Policies

Policies will be fulfilled with respect to a history if each minimal model of a history is also a model of the policies (see Definition 6.8). This definition in combination with the Papel semantics have the conclusion that a processing history will fulfill a set of policy rules, if all processing steps in the history are permitted by permissions ($permit^I$) and not prohibited by restrictions ($deny^I$) and if no assignment ($assignment^I$) has been violated.

6.3 Implementation

Implementing Papel, we have demonstrated its feasibility. In the implementation the provenance information is provided as a database. To access

the provenance information, we use the database query language Datalog. We have chosen Datalog to provide a general implementation with a formal grounding. We also introduce the sticky policy framework. The sticky policy framework extends the sticky logging framework with policies. Through the combination of both, accessing provenance information is eased.

6.3.1 Implementing Papel with Datalog

In our implementation, we use facts to specify the provenance information and rules to query the information. We do the implementation by a mapping. Each Papel primitive is mapped to a Datalog fact. For each identifier in Papel (e.g. `record_JD`) an unique atom in Datalog is created. For each policy rule in Papel a Datalog rule is created (see the implementation example below).

To validate a set of policies the following preparation steps are required:

1. A Datalog fact is generated for each processing step, reduced fact and attribute assignment that is described by the provenance information. The facts are added to the database;

2. All Datalog rules required to specify the semantics of Papel are added to the database;

3. The Datalog rules specifying the policies are added to the database; and

4. A Datalog fact specifying the processing step that should be performed is added to the database.

After this preparation, the database is ready to be queried by rules which verify whether processing steps are permitted or denied or whether assignments have been violated. In the following, we explain the preparation steps in detail.

Mapping Provenance Information to Facts: Starting with the `step` primitive, we use Datalog facts to specify the information about processing steps. Listing 6.1 depicts the provenance information of the health care scenario as Datalog facts. Reduced facts (`reduced` primitive) are represented analogously by Datalog (see line 3 of Listing 6.1). Attribute assignments (`attribute`) are also translated to Datalog facts (see line 5 of Listing 6.1).

Listing 6.1: *Provenance Information represented as Datalog Facts*

```
1: step(record, mrh, _, create, treatment, 1, 0).
2: step(record, mrh, _, update, examination, 2, 1).
3: reduced(record, hidden, hidden, create, hidden,
           3, 2).
4: step(record, mrh, _, deidentified, privacy,
        4, 3).
5: attribute(record, deidentified, true, 4).
6: step(record, mrh, ukob, transfer, research,
        5, 4).
7: step(record, ukob, _, read, research, 6, 5).
```

The additional semantics of the `reduced` primitive are captured by a rule introduced below (see Listing 6.2). Also the `attribute` primitive requires further Datalog rules to implement its full semantics defined by Papel (see below and Listing 6.3).

In Papel, we may use sets in primitives specifying provenance information, e.g. for multiple actors `step(_, {mrh, ukob}, _, _, _, 6, 5)` or multiple preceding processing steps `step(_, mrh, _, _, _, 9, {8, 7})`. The sets are interpreted as elements of the corresponding type (see Definition 6.15). Thus, each set is implemented as a single constant in Datalog (e.g. `mrh_ukob`) beside the set of preceding processing steps. As the

preceding processing steps specify the structure of the graph, we must be able to address the single preceding steps. Each `step` primitive (likewise for reduced facts) that has more than one preceding processing step is represented by multiple Datalog facts; one for each preceding processing step.

Implementing Papel's Semantics with Datalog Rules: After adding facts representing the provenance information, we add Datalog rules formalizing the semantics of Papel. The first rule enables the querying of reduced facts. The rule (see Listing 6.2) provides the mapping from the `reduced` to the `step` primitive.

Listing 6.2: *Implementation of the Mapping between Reduced Facts and Steps*

```
8: step(Data, Actors, InvolvedAgents, Category,
        Purpose, ID, PIDs) :-
   reduced(Data, Actors, InvolvedAgents, Category,
           Purpose, ID, PIDs).
```

The `attribute` predicate of Papel specifies the assignment of a value to an attribute. Each attribute has exactly one specific value at a time, possibly undefined. The attribute's latest setting preceding a specific point in time specifies that attribute's value at that point in time. This circumstances are addressed by the Datalog implementation of the `check_attribute` predicate depicted in Listing 6.3, which allows for checking whether an attribute has a specific value at a certain point in time (see line 18 of Listing 6.5 for an application of the predicate). The `changed_later` predicate is used to validate whether an attribute has been changed between to assignments and is used as helper predicate by the `check_attribute` predicate.

Listing 6.3: *Getting Attribute Value*

```
 9: changed_later(Data, Name, Value, PID, ID) :-
      attribute(Data, Name, Value, ID),
      attribute(Data, Name, NewerValue, IID),
      after(ID, IID), after(IID, PID),
      not(Value = NewerValue).

10: check_attribute(Data, Name, Value, ID) :-
      attribute(Data, Name, Value, ID).

11: check_attribute(Data, Name, Value, ID) :-
      attribute(Data, Name, Value, PID),
      after(ID, PID),
      not(changed_later(Data, Name, Value, PID,
          ID)).
```

To address the temporal structure of processing histories, we have introduced the `AFTER` operator in Papel. The `AFTER` operator is used to access the processing history. In the implementation, we can access the graph structure of the processing history by means of the step identifiers `ID`. The identifier `ID` in combination with the identifier `PID` of the preceding processing step define a partial order of the processing steps, which defines directed paths through the execution history. To express arbitrary paths, we implement a general `path` rule (see line 12 and 13 in Listing 6.4). To query for *direct* preceding processing steps, we implemented a `directPath` rule analogously (see line 14). The implementation of the `AFTER` operator (see line 15) is an application of this `path` rule.

Listing 6.4: *Implementation of the AFTER Operator*

```
12: path(Begin, End) :- step(_, _, _, _, _, End,
                             Begin).
13: path(Begin, End) :- path(Begin, Intermediate),
                        path(Intermediate, End).

14: directPath(Begin, End) :- step(_, _, _, _, _,
                                   End, Begin).

15: after(ID, PID) :- path(PID, ID).
```

Apart from the `AFTER` operator, Papel uses the boolean operators `NOT`, `AND`, `OR`, and `XOR`. These can be expressed by standard means of Datalog.

Specifying Policies in Datalog: The implementation of the policies of our scenario is achieved by Datalog rules. The conditions are specified as the body of the rule. The following listing depicts the implementation of the `Policy 4` and `Policy 6` of the health care scenario. `Policy 4` specifies that the University of Koblenz is allowed to process health records for research purposes, but the University of Koblenz is not allowed to transfer the health records (see lines 16 and 17). `Policy 6` demands that the transfer of the health record will only be allowed if the record has been de-identified (see line 18). The rule implementing `Policy 6` makes use of the `check_attribute` predicate we introduced above.

Listing 6.5: *Policy 4 and 6 in Datalog*

```
16: permit(ID) :- step(record_JD, ukob, _, _,
                       research, ID, _).
```

```
17: deny(ID)    :- step(record_JD, ukob, _,
                        transfer, _, ID, _).

18: permit(ID) :- step(record_JD, _, _, transfer,
                       _, ID, _),
                  check_attribute(record_JD,
                                  deidentified,
                                  true, ID).
```

Beside `permit` and `deny` rules, Papel specifies `assignment` rules. As assignments in Papel are used to set attributes or specify reduced facts, in the implementation we write the assignment part of the rule (after the `DO`) as head of the Datalog predicate and the condition as its body. The following example depicts the Datalog implementation of the assignment rules of Example 6.7.

Listing 6.6: *Assignment of the Deidentifaction Attribute*

```
19: attribute(record_JD, deidentified, true,
              ID) :-
      step(record_JD, _, _, update, deidentify,
           ID, _).

20: attribute(record_JD, deidentified, false,
              ID) :-
      step(record_JD, _, _, update, reidentify,
           ID, _).
```

As for provenance information, Papel allows for using sets in conditions, e.g. `step(_, {mrh, ukob}, _, _, _, ID, _)`. As Datalog does not natively support such sets and the semantics of Papel define such sets as sin-

gle elements of the corresponding type (see Definition 6.15), those sets are represented as single constants in Datalog. An exception to this rule is the set of preceding processing steps, which is implemented by specifying multiple predicates analogous to `step` primitives specifying provenance information (see also in aboves paragraph on 'Mapping Provenance Information to Facts').

This issue is aggravated in condition statements that use variables to address processing steps and preceding processing steps. Such conditions require a more extensive implementation to assure that the unification works as intended. For instance, the condition statement: `step(record_JD, _, _, update, _, ID, {8, A})`. We have to assure that the unification mechanism of Datalog does not substitutes A with 8, or if more variables are given, will not substitute multiple variables with the same processing step identifier. To overcome this issue, we have to explicitly specify that the variables and given constants are different. We achieve this by specifying restricting constraints, e.g. $A\ != 8$.

Validating Processing Steps: After the previous preparation steps, we can use the implementation to request whether a processing is (or has been) allowed or not. As Papel defines an interpretation order in Section 6.2.5, we must consider the following three rules expressing the policy validation strategy:

1. If at least one permission is true, the action will be permitted.
2. If at least one restriction is true, the action will be prohibited, even if it has been permitted.
3. If permissions and restrictions are true, the action is prohibited (restriction $>$ permission).

The following rule considers the validation strategy and is used to query whether the processing step identified by the given ID will be allowed or not:

Listing 6.7: *Is a Processing Step Allowed*

```
21: isAllowed (ID) :- permit (ID), not (deny (ID)).
```

The following query is executed eventually, where id is an identifier of an actual processing step:

Listing 6.8: *Is a Processing Step Allowed*

```
isAllowed (id).
```

6.3.2 Sticking Policies to Data Items

In this section, we introduce the sticky policy framework that extends the sticky logging framework by policies. Through the combination, accessing provenance information is eased.

Each policy is associated with one data item. Hence, we define a sticky policy as a pair of data items and policies, as a sticky log is a pair of data instances and logs. In a distributed process multiple instances of a data item may exist. To each data instance a copy of the policies is attached. The attaching is achieved by adding the policy to the data instances as part of the RDF graph containing the log. A policy may be specified by a resource describing the policy or by an URI pointing to an external resource (e.g. Web address) that contains the policy. Thus, each time a new data instance is created not only a new log needs to be created, but also all policies and links are copied and added to the RDF graph of the new log.

Changes of policies need to be propagated along the processing path. The propagation is achieved by means of the references that are part of the sticky

logs. This way no additional information is required and up-to-date policies are directly accessible with the data.

The policies are attached to the associated data by including them into the RDF graph of the sticky log, like depicted by Listing 6.9. The listing shows the attaching of `Policy 3` of the health care scenario to Jane Doe's health record. By means of `Policy 3` Jane Doe allows the Middle Rhine Hospital to use her health record for research.

Listing 6.9: *Attaching a Policy to a Data Item*

```
:Record_JD sp:protectedBy  :Policy3
:Policy3   sp:expressedBy "permit(ID) IF step(
                             record_JD, {mrh}, _,
                             _, research, ID, _)."
```

Alternatively, references can be used to point to resources containing policies. This method can be used for generic policies (as `Policy 1`) that are not expected to change or, if they change, the updated version is of relevance (e.g. policies based on laws). The server hosting the policy must be trust worthy (e.g. government server).

Listing 6.10: *Attaching a Generic Policy*

```
:Record_JD sp:protectedBy :Policy1
:Policy1   sp:specifiedAt "http://mrh.ex/policy"
```

6.4 Related Work

Many general purpose policy languages exist that are applicable for our purpose. Papel is based on one of these existing languages. It uses the main

policy elements 'restriction' (deny) and 'permission' (permit), as defined in the *eXtensible Access Control Markup Language (XACML)* [Moses et al., 2005]. Policies, which are based on complex environmental knowledge such as provenance information, can not easily be specified using XACML. Thus, Papel extends the expressiveness of XACML conditions to cover information about the processing history of data.

As a foundation for more complex policy conditions we use the work of Kagal et al. [Kagal et al., 2003]. They present a domain-centric and entity-centric policy language named Rei. Rei provides a mechanism to model speech acts and to delegate policies. The conditions in Rei are specified in Prolog. However, Rei does not define how to model conditions based on provenance information. Papel extends the work of Rei by defining a formalism for conditions based on provenance information and by providing a model-theoretical semantics for the temporal structures and the state changes. New language primitives such as `AFTER` and `assignment` have been added in such parsimonious manner that Papel can be captured nicely in a purely declarative Datalog implementation.

Related work in the area of temporal logic, linear temporal logic in particular, is used to validate the compliance of histories with specified policies. The authors of [Bauer et al., 2009] describe access control policies that relate to histories of transactions using their policy language $PTLTL^{FO}$. As $PTLTL^{FO}$ is a general purpose language, it does not define a specific semantics for provenance or data flows. However, $PTLTL^{FO}$ might probably be used to implement Papel.

Other related work is the *Web Services Policy 1.5 - Framework (WS-Policy)* [Vedamuthu et al., 2007] that provides a model and syntax to specify policies about entities in a Web services-based system. As WS-Policy is used to specify policies about entities, e.g. Web services, and not about processed data, it is not in the target of the WS-Policy framework to model conditions based on provenance information.

Table 6.3 gives an overview of related work in the field of policy languages. The table compares the different properties of policy languages that

are crucial for connecting policies and provenance information. In our example, the Middle Rhine Hospital is required to link policies to Jane Doe's health record to make them available for the University of Koblenz. To fulfill the *Contractual Requirement Availability*, policies require to be associated with a specific data item (Property 1). As the hospital and Jane Doe must be able to express provenance-aware policies (Property 2), the *Contractual Requirement Expressiveness* asks for policy conditions that can relate to knowledge about the data. Some scenarios require data flow (Property 3) and access control policies (Property 4). In our example, one requires both access control and data flow control to express `Policy 4`. The availability of the provenance data (see *Contractual Requirement Accessibility*) generates new risks for privacy and data protection. The policy language should be able to protect confidential provenance information (Property 5), e.g. to maintain privacy and data protection. The table depicts whether the language has a syntax definition (Property 6), a formal semantics (Property 7) and an implementation (Property 8).

Table 6.3 shows that Papel provides an original and significant contribution by enlarging policy languages with provenance access. Other dimensions exist that are not treated by Papel but may be combined with Papel. Some policy languages define policies of actors (cf. [Vedamuthu et al., 2007]) or transactions (cf. [Bauer et al., 2009]) not resources. Other approaches provide methods to enforce policy compliance in closed environments, such as organizations (cf. [Ashley et al., 2003]) or data silos (cf. [Gandon and Sadeh, 2004]). Other languages consider credentials to gain access rights (cf. [Becker and Sewell, 2004] and [Wang et al., 2002]), support roles and role delegation (cf. [Sandhu et al., 1996], [Becker and Sewell, 2004] and [Hilty et al., 2005]), provide algorithms to identify violated policies (cf. [Accorsi and Wonnemann, 2010]) or provide a role-based access control framework for workflow systems allowing a clear separation of permission aspects and workflow aspects (cf. [Wainer et al., 2003]). Various approaches, such as [Byun et al., 2005], [Byun and Li, 2008] and [Ni et al., 2010], extend the role based access control model (RBAC). Those extensions cover privacy

Table 6.3: Comparison of Approaches Related to Papel.

Approach ╱ # Property	1 Linked to Individual Data Items	2 Provenance Information	3 Data flow Policies	4 Access Control Policies	5 Protection of Provenance Information	6 Syntax Definition	7 Formal Semantics	8 Implementation
Papel	✓	✓	✓	✓	✓	✓	✓	✓
XACML [Moses et al., 2005]	-	-	-	✓	-	✓	-	✓
EPAL [Ashley et al., 2003]	-	-	-	✓	-	✓	✓	✓
WS-Policy [Vedamuthu et al., 2007]	-	-	-	(1)	-	✓	-	✓
Rei [Kagal et al., 2003]	-	-	-	✓	-	✓	(✓)	✓
XrML [Wang et al., 2002]	✓	-	-	✓	-	✓	-	✓
Casandra [Becker and Sewell, 2004]	-	-	-	✓	-	✓	✓	✓
e-Wallet [Gandon and Sadeh, 2004]	✓	-	-	✓	-	✓	(2)	✓
IFAudit [Accorsi and Wonnemann, 2010]	-	✓	✓	-	-	✓	✓	✓
PTLTL FO [Bauer et al., 2009]	-	✓	-	✓	-	-	✓	-
RBAC [Sandhu et al., 1996]	✓	-	-	✓	-	✓	✓	-
W-RBAC [Wainer et al., 2003]	✓	-	-	✓	-	✓	✓	-
[Hilty et al., 2005]	✓	-	-	✓	-	✓	✓	-

(1) WS-Policy provides data usage policies.

(2) e-Wallet provides a formal policy processing algorithm.

aspects providing a privacy-based access control model.

A different use of provenance information is depicted in [Naja et al., 2010]. The authors collect the provenance of decision processes in multi agent systems. They use the collected information to understand the made decisions and to check compliance with policies. Many other applications for provenance exists. An overview of the use of provenance in the Web is given in [Moreau, 2010]. The authors of the survey [Cheney et al., 2009] discuss various provenance solutions and means for databases, such as the provenance of queries. In scientific workflows, provenance is used to reconstruct and comprehend results. The surveys [Simmhan et al., 2005] and [Bose and Frew, 2005] introduce and compare approaches in this field.

That actions have a certain purpose is also considered by other approaches. In [Tschantz et al., 2011], the authors present a method for planning with purpose. They assume an environment, where all possible actions are specified beforehand. The approach assigns each performed action a probability that it has served a certain purpose. The purpose is derived from the given state of the system and is used to detected conspicuous actions. In [Agrawal et al., 2002], the authors take a step forward inspired by the Hippocratic Oath. Their vision is a system that takes responsibility for the performed actions. The authors derive the challenges and issues that arise from the implementation of their vision.

6.5 Summary

Conditions of policies may depend on the previous processing of data. Such policies define restrictions depending on the processing history. Existing policy languages do not specify how to access the temporal structure of provenance information and how to model state changes occurring in such historical data. To solve these issues, we have introduced Papel, a provenance-aware policy definition and execution language.

We have based Papel on existing means to express provenance (OPM) and

policies (XACML). Papel introduces a method for mapping the temporal structure of processing histories to a graph structure. We have specified the `AFTER` operator to access the partial order of processing steps in the processing history. Through the mapping and the `AFTER` operator, one is easily able to specify policies conditions relating to provenance information. By implementing Papel in Datalog we have not only shown its feasibility but also demonstrated that it can be implemented by means of a purely declarative logic programming language. This shows that Papel is less complex than temporal logic.

As certain risks arise from providing provenance information with data in distributed environments, we have identified the requirement to consider means for data protection. We have analyzed how data protection can be achieved and how data protection can complicate the interpretation of policy conditions. To accomplish both, we have presented means to validate compliance with given policies even if the required provenance information is confidential.

Through Papel we confirm Hypothesis 3 by providing a syntax and semantics for policy conditions that can take the temporal structures of processing histories into account. The results of the work presented in this chapter have been published at the International Conference on Business Process Management [Ringelstein and Staab, 2010b] and in the IEEE Internet Computing Journal [Ringelstein and Staab, 2011].

7 Meeting Your Future Obligations with Care

We have discussed how contracts and laws set rules for the processing of data, in Chapter 1 and Chapter 3. In Chapter 6, we have shown how policy languages provide means to restrict the current processing of data through permission and denial rules. As specified by the *Contractual Requirement to Fulfill Obligations*, sometimes the current processing does not need to be restricted, but contracts or laws demand a certain future reaction, such as to delete all health records after a certain time. Some policy languages foresee the possibility to specify obligations, which consist of *trigger conditions* and *obligatory processing steps*. The obligatory processing steps must be performed in future after the trigger condition has been fulfilled.

In the case of permissions and denials, actors can directly determine whether planned processing steps violate rules. Regarding obligations, a decision made now may lead to a violation in future by preventing the execution of obligatory processing steps as described by *Issue of Invalid Future Executions* that we introduced in Chapter 3. Hence, the future must be considered while deciding which processing step should be performed next, as the selected step must not prevent the fulfillment of obligatory processing steps. Existing policy languages that allow for specifying obligations, such as [Bradshaw et al., 2003, Lupu and Sloman, 1999, Moses et al., 2005], do not provide procedures to decide whether there is a future execution meeting all obligations.

In Chapter 3 we discussed the *Issue of Valid Processing Steps*. Because of this issue, the actor has to verify the feasibility of at least one future execu-

tion path that fulfills all obligations. This path has to prevent the violation of obligations under consideration of the given permission and denial rules. As an execution can consist of multiple parallel paths, we discuss execution graphs in the following. We call an execution graph that fulfills all obligations and does not violate any permission or denial a *valid execution graph*. The subgraph of a valid execution graph containing all future processing steps is called the *destiny* of the data processing.

The problem is to decide whether a destiny exists at the current point in the execution. One requires a decision procedure to compute this. With Care we provide a formal decision procedure by reducing the given decision problem to the problem of deciding reachability in colored Petri nets [Jensen, 1992, Mayr, 1981]. We translate the given policies and the history to a colored Petri net before we decide reachability within the colored Petri net.

To provide a solution, we first discuss the application and the problems of obligations in the health care scenario of Chapter 3 in Section 7.1. As the current state of an obligation depends on information about the processing history, e.g. has the obligation rule been instantiated, we have defined Care in such a way that it extends Papel. We give an overview of the required refinements of the syntax and semantics of Papel in Section 7.2. Based on these refinements, Care defines syntax and semantics of future obligations in Section 7.3. In Section 7.4, we formalize the decision problem and define the translation to colored Petri nets. Before we conclude, we discuss related work in Section 7.5.

7.1 Violating Obligations

In the case of permissions and denials, actors can directly determine whether planned processing steps violate rules or not. Accessing a health record without explicit permission is such a violation. An obligation rule may be violated in future by a decision made now, such as deleting a health record that should have been stored for auditing purposes later. To avoid violations,

one must consider the future while deciding which processing step should be performed next. The selected step must neither prevent the fulfillment of obligatory processing steps in future nor violate any other policy rule.

To discuss our solution, we use the following terminology:

- **Possible Processing Steps:** The execution of processing steps in an environment can be restricted. An example of such a restriction is that an entity can only process data instances during their life span. In our scenario, Jane Doe's health record cannot be read before it is created or after it is deleted. Processing steps that can be executed under consideration of environment restrictions are called *possible processing steps*.

- **Allowed Processing Steps:** *Allowed processing steps* are possible processing steps that do not violate permit or deny rules.

- **Valid Processing Steps:** We call allowed processing steps that do not render instantiated obligations unfulfillable *valid processing steps*.

- **Execution** and **Execution Graphs:** The processing of data consists of performing multiple processing steps called the *execution*. The progression of the execution specifies a partial order of processing steps. A graph representing this partial order is called an *execution graph*. An execution graph of processing Jane Doe's health record is depicted in Figure 7.1. In the figure, the processing steps are modeled as arrows.

- **History:** The history is the execution graph of all processing steps performed until a point in time. In the figure, the vertical, dotted line represents the current time. On the left side of the line is the history consisting of the already performed processing steps s_1 to s_6 and s_8.

- **Future Execution Graph:** The graph of all possible future processing steps constitutes the *future execution graph*. The future execution

graph is a partial order of processing steps that is disjunct from the history and where all processing steps appear after processing steps of the history. In Figure 7.1, the future execution graph is on the right side of the dotted line.

- **Destiny:** The destiny is the subgraph of the future execution graph containing all valid processing steps. In Figure 7.1 the subgraph between the dashed lines represents the destiny.

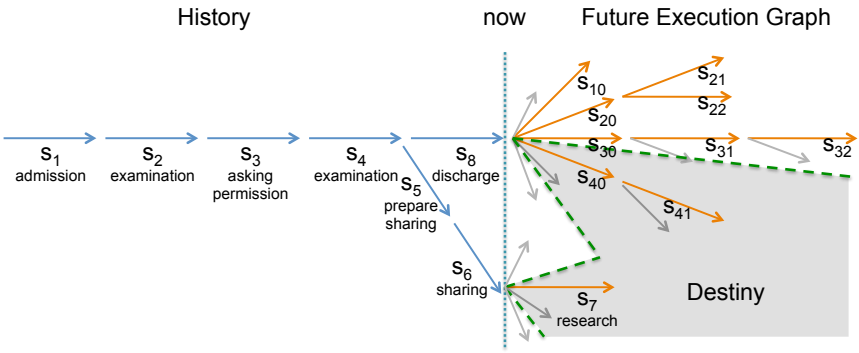

Figure 7.1: History, Future and Destiny of the Scenario.

In Chapter 3, we have introduced the health care scenario. The processing starts with the creation of the health record in step s_1 depicted in Figure 7.1. After the creation the record has been read and updated by different departments of the hospital in (partially) parallel steps s_2 to s_6.

In the last step of the process described in the scenario in Section 3.1.2, Jane Doe is discharged by a nurse. The nurse transfers the health record to the administration in step s_8. By discharging Jane Doe, the obligation specified by `Policy 2.2` is instantiated and the health record has to be transfered to her. After the instantiation of the obligation various possible processing steps can be chosen. Depending on the choice, certain problems

may occur that hamper the fulfillment of this obligation. In the following, we give some examples for both valid and invalid executions which are also shown in Figure 7.1.

- ***Invalid Execution by Policy Violation:*** The archive hands out the record to Jane Doe (step s_{10}). This processing step fulfills the obligation specified by `Policy 2.2` but violates `Policy 8.2`.
- ***Invalid Execution by Obligation Violation:*** As everyone is allowed to hand out Jane Doe's health record, the archive transfers the record to Bob (step s_{20}). Bob has to delete the health record (step s_{21}) following the obligation specified by `Policy 9.2`, even if the record has to be transferred to Jane Doe (step s_{22}) following `Policy 2.2`. In this case, the already instantiated obligation specified by `Policy 2.2` will become violated if the obligation specified by `Policy 9.2` is fulfilled. On the other hand, the obligation of `Policy 9.2` will be violated if the obligation of `Policy 2.2` is fulfilled.
- ***Invalid Execution by Loop:*** The archive transfers the record to a nurse (step s_{30}). Following `Policy 8.1` the nurse has to transfer the record to the archive (step s_{31}), which may transfer it again to a nurse (step s_{32}). By transferring the record back to the archive, the nurse fulfills the obligation specified by `Policy 8.1` and does not violate the obligation specified by `Policy 2.2`. As long as the nurse does not give the record to the patient, the process will be in a loop and obligation specified by `Policy 2.2` will remain active and unfulfilled.
- ***Valid Execution:*** A physician may release the record to Jane Doe (step s_{41}) after receiving it from the administration (step s_{40}). This execution graph describes a destiny.

The shorter arrows indicate additional possible future processing steps not discussed in this scenario.

The aim of Care is to provide a procedure to decide the existence of a destiny of Jane Doe's health record. The procedure has to be based on the given policy rules and the processing history.

7.2 Refining Papel

To express policy rules and the processing history, Care is based on Papel (see Chapter 6). Care refines Papel by introducing new concepts, like destiny and obligations, but also by refining existing concepts. In this section, we give a short overview of the refined syntax and semantics. This overview focuses on the definitions required by Care, i.e. processing steps, history, and permit and deny rules. We introduce the syntax of processing steps as we skip the specification of the purpose to increase readability. The purpose is not required for our further considerations. We add the specification of an involved data instance as it is required by merge actions, which we newly introduce in Care. In Care, Definition 6.1 is refined as follows:

Definition 7.1: *Care Syntax of Processing Steps*

We define the representation of a processing step as a 7-tuple with the following syntax:

```
step (Data, SecondData, Actor, Receiver,
      Category, ID, PIDs)
```

where:
- `Data` refers to the *data instance* processed during the execution of the processing step,
- `SecondData` refers to the *data instance* created by a copy action or consumed by a merge,
- `Actor` is the *entity* controlling the processing step,
- `Receiver` is the *entity* receiving a data transfer,

- `Category` is the category of the processing step,
- `ID` is the unique *identifier* of the processing step, and
- `PIDs` are the unique *identifiers* of the directly preceding processing steps.

As in Papel, `ID` is the identifier of the current processing step and `PIDs` is the set of identifiers of the directly preceding steps. Both model the directed graph of partially ordered processing steps.

Example 7.1: *Provenance Information of a Processing Step in Care*

After Jane Doe's stay at the hospital ends, the nurse Alice transfers her health record to the administration:

```
step (record_jd, NULL, alice, administration,
      transfer, 8, {4}).
```

In the example, `record_jd`, `alice` and `administration` are instances of concepts defined in the domain of the Middle Rhine Hospital, while `transfer` is an action category as specified by DiALog (see Chapter 4).

The semantics of Papel is defined by a model theoretical interpretation function. In Care, we refine the universe U. The universe is the set $U = \Delta \cup \Gamma \cup X \cup \Phi$, where Δ is the set of data instances, Γ is the set of entities, X is the set of action categories and Φ is the set of processing step identifiers. These subsets of U are mutually disjoint. The corresponding interpretation function I is defined as follows:

Definition 7.2: *Interpretation Function*

The interpretation function I defines a mapping from atomic elements of Papel to (parts of) the universe U. I is defined to map atomic elements of Papel onto the (parts of) our universe U as follows:

$$\texttt{Data}^I \in \Delta,$$
$$\texttt{SecondData}^I \in \Delta,$$
$$\texttt{Actor}^I \in \Gamma,$$
$$\texttt{Receiver}^I \in \Gamma,$$
$$\texttt{Category}^I \in X,$$
$$\texttt{ID}^I \in \Phi, \text{ and}$$
$$\texttt{PIDs}^I \subseteq \Phi.$$

Processes are executed by actors performing actions on the data. Performing an action on a data item constitutes a processing step. The performed processing steps occur in partial order. Papel defines the interpretation of processing steps as given in Definition 6.15. As Care refines the syntax of processing steps, it also refines the definition of the interpretation:

Definition 7.3: *Interpretation of Processing Steps*

Let I be the interpretation function, let _ be an unspecified parameter, let S be a substitution of variables to Papel terms, let e be the logical expression in which the $step$ predicate occurs and let $(eS)^I$ be $true$, the interpretation of the $step$ predicate is defined by the function $(step)^I : U^7 \to \{true, false\}$:

$$(step(\delta,\ \eta,\ \alpha,\ \beta,\ \chi,\ \phi,\ \rho))^I =$$
$$\begin{cases} \text{true} & \text{if } \exists(\delta',\ \eta',\ \alpha',\ \beta',\ \chi',\ \phi',\ \rho') \in step^I : \\ & (\delta^I = \delta' \vee \delta = \_ \vee \{\delta \mapsto \delta'\} \subseteq S) \wedge \\ & (\eta^I = \eta' \vee \eta = \_ \vee \{\eta \mapsto \eta'\} \subseteq S) \wedge \\ & (\alpha^I = \alpha' \vee \alpha = \_ \vee \{\alpha \mapsto \alpha'\} \subseteq S) \wedge \\ & (\chi^I = \chi' \vee \chi = \_ \vee \{\chi \mapsto \chi'\} \subseteq S) \wedge \\ & (\phi^I = \phi' \vee \phi = \_ \vee \{\phi \mapsto \phi'\} \subseteq S) \wedge \\ & (\rho^I = \rho' \vee \rho = \_ \vee \{\rho \mapsto \rho'\} \subseteq S), \\ \text{false} & \text{else.} \end{cases}$$

Each $s \in step^I$ is uniquely identified by one ϕ with the following constraint $\forall s_i, s_j \in step^I : \phi_{s_i} \neq \phi_{s_j} \vee s_i = s_j$.

The history assembles the parts of a process execution that have been completed so far, i.e. its provenance. We refine the Papel definition of a history H by defining the history as partially ordered set of already performed processing steps:

Definition 7.4: *History*

We define a history of a data processing as partial order $\leq_H \subseteq \Phi \times \Phi$ of processing steps. A tuple $(\rho,\ \phi)$ is an element contained in $\leq_H$, if the processing step specified by ϕ has been performed and if the processing step specified by ρ precedes the processing step specified by ϕ or if $\rho = \phi$. We define $\Omega_H \subseteq \Phi$ to be the set of processing step identifiers in the history $\Omega_H = \{\phi \mid (\phi,\ \rho) \in \leq_H \vee (\rho,\ \phi) \in \leq_H \}$.

Papel uses policies to manage the processing of data. In Papel, policies are rules specifying permitted and denied processing steps. The following example shows policy rules from our scenario specified in the refined syntax of Papel:

Example 7.2: *Permission rule of Policy 2.1 and denial rule of Policy 8.2*

Everyone is permitted to transfer Jane Doe's health record to Jane Doe:

```
permit (ID) step (record_jd, _, _, jane_doe,
                  transfer, ID, _).
```

The archive is denied to transfer health records to patients.

```
deny (ID) step (H, _, archive, P, transfer, ID, _)
          AND instance_of (H, health_record)
          AND instance_of (P, patient).
```

The variable `ID` identifies the current processing step. The relation

`instance_of` will be evaluated to true if the term assigned to the variables `H` and `P` specify instances of the `health_record` and `patient` concepts. This requires domain knowledge, which must be given using a formalism such as Datalog or first order logic. The `step` primitives will be evaluated to true if steps $s \in step^I$ exist that match the parameters of the primitives, whereby the variable '_' specifies a new unnamed variable each time it is used.

We introduce a *normal form* of policy rules. Policies in normal form have a specific syntactical structure.

Definition 7.5: *Policy Normal Form*

We define permit and deny rules to be in normal form, if their condition does not use disjunction and if they have the following syntax:

```
{permit, deny} (ID) Step_pattern[ AND Constraint].
```

In normal form, the conditions of permit rules do not use negation.

By means of the `ID` variable the current processing step is identified. Policy conditions without disjunction can only address one current processing step. The current step is specified by a step primitive. We call this specification the *step pattern* and the rest of the condition the *constraint*, which is optional.

In Papel, conditions are composed by boolean operators including negation. In normal form, the use of the negation operator is not allowed in permit rules. This does not affect the expressiveness as the negation can be expressed by means of deny rules. The rule `permit (ID) step (H, _, archive, _, transfer, ID, _) AND NOT instance_of (H, health_record)` can be expressed by the two rules `permit (ID) step (_, _, archive, _, transfer, ID, _)` and `deny (ID) step (H, _, archive, _,`

`transfer, ID, _) AND instance_of (H, health_record).`

Policy conditions can consist of multiple disjunctive parts. Policies in normal form only consist of conditions without disjunction. Policy rules with disjunction can be decomposed to policy rules without disjunction by transforming their conditions into disjunctive normal form. The permit rule '`permit (ID) IF step (record_jd, _, A, B, transfer, ID, _) AND (instance_of(A, nurse) OR instance_of (A, physician)) AND instance_of(B, patient)`' uses disjunction. This rule can be expressed by two single permit rules. The first permits nurses to transfer the record `record_jd` to a patient and the second rule permits for physicians to do the transfer.

In Papel, we have defined policy rules by functions ($\Phi \rightarrow \{true, false\}$) expressing whether a processing step is permitted or denied (see Definition 6.20 and 6.22). The interpretation order of permission and denial defines that a processing step will be allowed if it is permitted and not denied (denial overrides permission) (see Section 6.2.5). From this interpretation order, we can derive the $is\_allowed$ function. This function expresses whether a processing step is allowed under the given set of policies and the provenance specified in the history $\leq_H$.

Definition 7.6: *Allowed Processing Steps*

Be ϕ_s the identifier of the processing step, be $p_1,\ p_2,\ ...,\ p_n$ the statements defining the conditions of permit policies, and be $d_1,\ d_2,\ ..,\ d_m$ the statements defining the conditions of deny policies, we define the $is\_allowed : step^I \rightarrow \{true, false\}$ function as follows:

$$is\_allowed(s) = \begin{cases} \text{true} & \text{if } (p_1(\phi_s)^I \vee p_2(\phi_s)^I \vee ... \vee p_n(\phi_s)^I = true) \wedge \\ & \neg(d_1(\phi_s)^I \vee d_2(\phi_s)^I \vee ... \vee d_m(\phi_s)^I = true) \\ \text{false} & \text{else} \end{cases}$$

This summary of Papel focuses on the parts refined by Care. For a complete definition of syntax and semantics of permit and deny rules as well as the `AFTER` operator to address provenance information in policy conditions, please refer to the definition of Papel in Chapter 6. Papel provides means to express purposes as well as attributes, and protection mechanisms for provenance information. Care can be easily extended by these.

7.3 Extending Syntax and Semantics Towards Obligations

While Papel focuses on extending policies towards provenance-awareness and by this towards the history of the data processing, Care looks into the future of the data processing, its destiny. To express the future, we extend syntax and semantics of Papel by obligations, based on the definitions in Section 7.2.

7.3.1 Care Syntax

Obligations are rules specifying trigger conditions and obligatory processing steps. A condition of an obligation rule that is fulfilled instantiates the obligation. An obligatory processing step will be fulfilled if it has been executed finally. Even if Care is action-based, one can compare this to the **F**uture operator of temporal logic [Pnueli, 1981]. As the definition of processing steps models the graph structure of the execution, one can also specify obligatory processing steps that must be performed in n steps after the instantiating of the obligation (cf. the **N**ext operator of temporal logic).

Definition 7.7: *Obligation Rule*

We define the syntax of an obligation rule as :

```
obligation IF Trigger_condition
           DO {AT ANYTIME, AFTERWARDS}
              Obligatory_processing_steps.
```

where:

- `Trigger_condition` is the condition specifying when an obligation is activated, and
- `Obligatory_processing_steps` specifies the obligatory processing steps.

Both `Trigger_condition` and `Obligatory_processing_steps` are specified by means of expressions using Boolean operators as well as the `AFTER` operator to combine `step` primitives and further primitives defined in the domain, such as the `instance_of` primitive in Example 7.2. The `AFTER` is used to address the partial order of processing steps in the history, as introduced in Papel in Chapter 6. `AT ANYTIME` and `AFTERWARDS` define so called *time constraints*. `AFTERWARDS` specifies that the obligatory processing step must be performed after the obligation is instantiated, e.g. *a back-up must be created after an update is performed*. While an obligation using `AT ANYTIME` will be also fulfilled if the obligatory processing step has been performed before the condition has been triggered. The following example depicts obligation of Policy 2.2 from the scenario:

Example 7.3: *Obligation of Policy 2.2*

Jane Doe demands to receive her record after her discharge:

```
obligation IF step (record_jd, _, _, _, discharge,
                    _, _)
           DO AFTERWARDS step (record_jd, _, _,
                               jane_doe, transfer,
```

_, _) .

The first `step` primitive specifies the trigger condition after the `IF`. Behind the `AFTERWARDS` the second primitive specifies a pattern for the obligatory processing step.

7.3.2 Care Semantics

The definitions introduced in this section specify the semantics of Care. The axioms stipulate how performing processing steps constitute an execution graph. In the following axioms and definitions, we use '_' to represent Skolem constants. Each occurrence of '_' represents another unnamed Skolem constant.

Like the history of performed processing steps, the possible future of the processing is represented by an execution graph:

Definition 7.8: *Future Execution Graph*

The future execution graph is defined as a partial order $\leq_F \subseteq \Phi \times \Phi$ of processing steps that is disjunct from the history $\leq_H \cap \leq_F = \emptyset$.

If the history is not empty $\leq_H \neq \emptyset$, all processing steps in $\leq_F$ will appear after processing steps of the history:

$$\forall(\rho,\ \phi) \in \leq_F \colon (\exists\,(\zeta,\ \eta) \in \leq_H \colon \eta < \rho) \wedge (\nexists\,(\zeta,\ \eta) \in \leq_H \colon \rho \leq \eta).$$

If the history is empty $\leq_H = \emptyset$, the first step of $\leq_F$ has to be a create action $(\delta, \epsilon, \alpha, \epsilon, create, \phi, \emptyset)$ and all other steps must appear after this step :

$$\forall(\rho,\ \_) \in \leq_F \colon \phi \leq_F \rho.$$

We define $\Omega_F \subseteq \Phi$ to be the set of identifiers of the processing steps in the future execution graph $\Omega_F = \{\phi \mid (\phi,\ \rho) \in \leq_F\ \vee(\rho,\ \phi) \in \leq_F\ \}$.

The conjunction of history and future execution is the partial order $\leq_E$ defining the execution graph: $\leq_E = \leq_H \cup \leq_F$. We define the subgraph $\leq_A$ of $\leq_E$ consisting of the allowed processing steps as: $\leq_A = \{(\phi,\ \rho) | (\phi,\ \rho) \in \leq_E \wedge\ is\_allowed(\phi) = true\}$. Ω_E and Ω_A are defined accordingly.

The structure of the execution graph $\leq_E$ is specified by the execution of processing steps. The structure depends on the action category of the executed processing steps (see Figure 7.2). In Care, seven upper-level categories of actions are specified, which are a refinement of the six action categories defined in DiALog (see Chapter 4): The set of action categories X is defined as follows:

$$X = \{create, read, update, transfer, delete, copy, merge\}$$

The categories $create$, $read$, $update$, $transfer$, $copy$ and $delete$ represent the same categories as in DiALog. The $merge$ action describes the merging of two or more data instances into one instance.

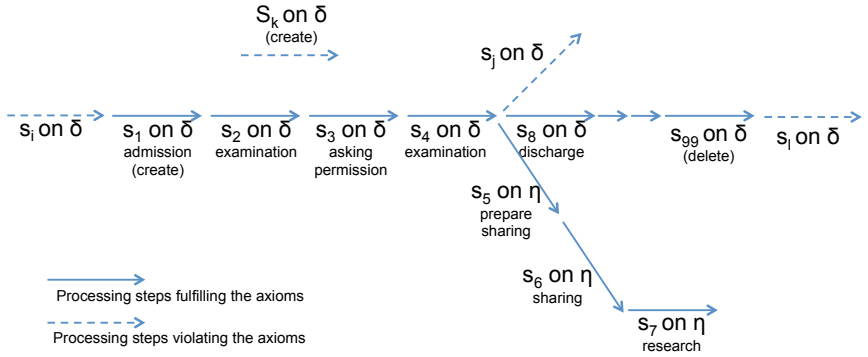

Figure 7.2: Structure of Execution Graphs.

Parallel processing steps can influence each other, such as reading and updating a data instance in parallel. Without additional specifications it is not

clear whether the read is performed on the original or on the updated data instance. The order of processing steps is crucial for the evaluation of policy conditions. For instance, if the condition requires that a read has been performed after an update action, a clear representation of the parallel processing is required. To achieve a clear representation, Care demands that the parallel processing of single data items is explicitly represented by the parallel processing of different instances of this item. Explicit parallel processing can be achieved by copying the data instance first and processing the copy parallel to the original instance. Hence, only one processing step can be performed on a data instance $\delta \in \Delta$ at a time. In Figure 7.2, the step s_j is not a valid parallel processing step performed on δ. The following axiom specifies this constraint:

Axiom 7.1: *Explicit Parallel Processing*

Let $(\delta, \_, \_, \_, \_, \phi', \rho'), (\delta, \_, \_, \_, \_, \phi'', \rho'') \in step^I$ be two processing steps performed on the same data instance $\delta \in \Delta$ and both steps occur in the execution graph $\phi', \phi'' \in \Omega_E$, the steps may not be performed in parallel:

$$\forall\, (\delta, \_, \_, \_, \_, \phi', \rho'), (\delta, \_, \_, \_, \_, \phi'', \rho'') \in step^I : \phi' \leq_E \phi'' \vee\ \phi'' \leq_E \phi'$$

The axiom stipulates that for all pairs of processing steps $(\delta, \_, \_, \_, \_, \phi', \rho')$ and $(\delta, \_, \_, \_, \_, \phi'', \rho'') \in step^I$ performed on the same data instance δ occurring in the execution graph $\phi' \in \Omega_E \wedge \phi'' \in \Omega_E$ one step of the pair must precede the other step $\phi' \leq_E \phi''$ or $\phi'' \leq_E \phi'$. If one processing step precedes another processing step, the processing steps will be in a sequential order and not parallel.

The processing of a data instance starts with its creation and cannot be processed before it is created. In Figure 7.2, the processing starts with step s_1.

Axiom 7.2: *Creation of a Data Instance*

Let $(\delta, \epsilon, \alpha, \epsilon, create, \phi, \emptyset) \in step^I$ be a processing step performed by the actor $\alpha \in \Gamma$ creating the data instance $\delta \in \Delta$, and let the processing step occur in the execution graph $\phi \in \Omega_E$, the data instance δ cannot be processed before its creation:

$$(\forall\, (\delta, \_, \_, \_, \_, \zeta, \_) \in step^I : \phi \leq_E \zeta) \wedge$$
$$(\forall\, (\_, \delta, \_, \_, \_, \zeta', \_) \in step^I : \phi \leq_E \zeta')$$

Let $(\delta, \eta, \alpha, \epsilon, copy, \phi, \rho) \in step^I$ be a processing step performed by the actor $\alpha \in \Gamma$ creating the copy $\eta \in \Delta$ of the data instance $\delta \in \Delta$, and let the processing step occur in the execution graph $\phi \in \Omega_E$, the data instance η cannot be processed before its creation:

$$(\forall\, (\eta, \_, \_, \_, \_, \zeta, \_) \in step^I : \phi \leq_E \zeta) \wedge$$
$$(\forall\, (\_, \eta, \_, \_, \_, \zeta', \_) \in step^I : \phi \leq_E \zeta')$$

The axiom stipulates that from the occurrence of any processing step $(\delta, \_, \_, \_, \_, \zeta, \_) \in step^I$ performed on δ or η in the execution graph $\zeta \in \Omega_E$ follows that the step occurs after the creation or copy step $\phi \leq_E \zeta$. Analogously, the axiom stipulates that all processing steps $(\_, \delta, \_, \_, \_, \zeta, \_) \in step^I$ involving δ or η are performed after the creation or copy step. If a processing step performed on δ or η occurs before the creation or copy, this axiom will be violated. In Figure 7.2, step s_i is performed before the creation step s_1 and violates the axiom. The axiom will also be violated if another processing step creates the same data instance, such as step s_k in Figure 7.2.

After the creation of a data instance, it can be processed during its life span. Care postulates that a processing step uniquely identified by $\phi \in \Phi$ and performed on δ will only be executable by an actor $\alpha \in \Gamma$, if the actor

is in possession of δ. We first define the $\prec_\delta$ relation to specify the nearest preceding processing steps performed on the same data instance δ in $\leq_E$.

Definition 7.9: *Nearest Preceding Processing Steps*

Let $\delta \in \Delta$ be a data instance, and let $<_E \subseteq \leq_E$ be the strict sub-relation of the execution graph. The nearest preceding processing step relation $\prec_\delta \subseteq \Phi \times \Phi$ of δ is defined as follows:

$$\begin{aligned}\prec_\delta = \{(\rho,\phi) \mid \\ ((\exists(\delta,\_,\_,\_,\_,\rho,\_),(\delta,\_,\_,\_,\phi,\_) \in step^I : \rho <_E \phi) \vee \\ (\exists(\delta,\_,\_,\_,\_,\rho,\_),(\_,\delta,\_,\_,\phi,\_) \in step^I : \rho <_E \phi) \vee \\ (\exists(\_,\delta,\_,\_,\_,\rho,\_),(\delta,\_,\_,\_,\phi,\_) \in step^I : \rho <_E \phi) \vee \\ (\exists(\_,\delta,\_,\_,\_,\rho,\_),(\_,\delta,\_,\_,\phi,\_) \in step^I : \rho <_E \phi)) \wedge \\ (\nexists(\delta,\_,\_,\_,\_,\zeta,\_) \in step^I : (\rho <_E \zeta <_E \phi) \wedge \\ (\nexists(\_,\delta,\_,\_,\_,\zeta',\_) \in step^I : (\rho <_E \zeta' <_E \phi)\}\end{aligned}$$

The axiom stipulates that a pair of different $(\rho <_E \phi)$ processing steps will be in $\prec_\delta$ if both are performed on δ or involve δ and if no other processing step ζ exists that is performed on δ or involves δ and occurs between ρ and ϕ $(\rho <_E \zeta <_E \phi)$.

The environment of the data processing may restrict the access of entities to data instances. In the scenario, Jane Doe cannot read her record without a hospital member having passed it to her. If an entity can perform actions on a data instance, we will call the entity to be in possession of the data instance. In Care, policy conditions and obligatory processing steps may require an explicit modeling of the change of possession by transfer actions. Care stipulates that each data instance can only be possessed by one entity at a time and the possession can only be changed by transfer actions.

Definition 7.10: *Possession*

Let $s = (\delta, \eta, \alpha, \beta, \chi, \phi, \rho) \in step^I$ be a processing step. We define the

possession relation $pos_\phi \subseteq \Gamma \times \Delta$ after performing s as follows:

- if $\chi \in \{create, read, update, copy, merge\}$, $(\alpha, \delta) \in pos_\phi \ \wedge \forall \gamma \in \Gamma : ((\gamma, \delta) \in pos_\phi) \rightarrow (\gamma = \alpha)$,
- if $\chi \in \{copy\}$, $(\alpha, \eta) \in pos_\phi \wedge \forall \gamma \in \Gamma : ((\gamma, \delta) \in pos_\phi) \rightarrow (\gamma = \alpha)$,
- if $\chi \in \{transfer\}$, $(\beta, \delta) \in pos_\phi \ \wedge \forall \gamma \in \Gamma : ((\gamma, \delta) \in pos_\phi) \rightarrow (\gamma = \beta)$, and
- if $\chi \in \{delete\}$, $\nexists \gamma \in \Gamma : (\gamma, \delta) \in pos_\phi$.

α can only perform an action on δ if the nearest preceding processing steps leads to α possessing δ. In the case of a merge action, α has to also possess η. From Axiom 7.1 follows, that only one nearest preceding processing step exists for each data instance.

Axiom 7.3: *Possession of Data Instances*

Let $(\delta, \eta, \alpha, \beta, \chi, \phi, \rho) \in step^I$ with $\chi \in X \backslash \{create\}$ be the processing step to be performed, the nearest preceding processing step $\phi' \prec_\delta \phi$ performed on δ has to lead to $(\alpha, \delta) \in pos_{\phi'}$. If $\chi \in \{merge\}$, the nearest preceding processing step $\phi'' \prec_\eta \phi$ performed on η has to lead to $(\alpha, \eta) \in pos_{\phi''}$.

As a data instance cannot be processed before its creation, a data instance cannot be processed after its deletion, such as step s_l in Figure 7.2:

Axiom 7.4: *Deletion of a Data Instance*

Let $(\delta, \epsilon, \alpha, \epsilon, delete, \phi, \rho) \in step^I$ be the processing step deleting the data instance $\delta \in \Delta$ performed by the actor $\alpha \in \Gamma$, and let the processing step occur in the execution graph $\phi \in \ \Omega_E$, δ cannot be processed after it has been deleted:

$$(\forall\ (\delta, \_, \_, \_, \_, \zeta, \_) \in step^I : \zeta \leq_E \phi) \wedge$$
$$(\forall\ (\_, \delta, \_, \_, \_, \zeta, \_) \in step^I : \zeta \leq_E \phi)$$

Let $(\delta, \eta, \alpha, \epsilon, merge, \phi, \rho) \in step^I$ be the processing step consuming the data instance $\eta \in \Delta$ as part of a merge with the data instance $\delta \in \Delta$, and let the processing step occur in the execution graph $\phi \in \Omega_E$, η cannot be processed after it has been deleted:

$$(\forall\ (\eta, \_, \_, \_, \_, \zeta, \_) \in step^I : \zeta \leq_E \phi) \wedge$$
$$(\forall\ (\_, \eta, \_, \_, \_, \zeta, \_) \in step^I : \zeta \leq_E \phi)$$

The axiom stipulates that from the occurrence of processing steps $(\delta, \_, \_, \_, \_, \zeta, \_) \in step^I$ performed on δ or η in the execution graph $\zeta \in \Omega_E$ follows that they occur before the deletion respectively merge step $\zeta \leq_E \phi$. Analogously, the axiom stipulates that all processing steps $(\_, \delta, \_, \_, \_, \zeta, \_) \in step^I$ involving the data instance are performed before the deletion respectively merge step. If a processing step performed on δ or η occurs after the deletion or merge, this axiom will be violated.

Based on the definition of the interpretation of logical expressions in Papel, we define the semantics of obligation rules. We start by the interpretation of the logical expression specifying the trigger condition. The evaluation of the trigger condition is done according to the evaluation of condition statements in Papel in Section 6.2.2. A trigger condition will be evaluated to *true* if the logical expression specifying the condition is evaluated to *true*. If the logical expression contains variables, a substitution has to exist after which the expression evaluates to *true*.

The evaluation of the logical expression specifying obligatory processing steps is done accordingly. The expression will be evaluated to *true* if the specified processing steps have been executed. Again, substitutions will be used if the expression contains variables.

Based on these definitions, we formally define when an obligation is instantiated and when it is fulfilled. An obligation will be instantiated if the

logical expression specifying the trigger condition is interpreted to *true*. Given a history, an obligation can be instantiated multiple times. If the logical expression of the trigger condition uses variables, the single instances can differ. The different instances can be identified by the used substitutions. An instance will be fulfilled if the logical expression specifying the obligatory processing steps is interpreted to true under consideration of the according substitution and of the time constraint specified by the obligation rule.

Definition 7.11: *Interpretation of Obligations*

Let O be the set of all obligation rules, let $o \in O$ be a policy rule specifying an obligation $o =$ "`obligation IF Trigger_condition DO {AT ANYTIME, AFTERWARDS} Obligatory_processing_steps.`", let Σ be the set of all substitutions and let $S \in \Sigma$ be a substitution specifying an instance of o. Let Φ_T be the set of identifiers of the processing steps instantiating the trigger condition and let Φ_O be the set of step identifiers of the obligatory processing steps.

In a given execution graph $\leq_E$, a substitution will specify an instance of an obligation if the trigger condition is interpreted to $true$ after applying the substitution. The $is\_instance : O \times \Sigma \times 2^{\Phi \times \Phi} \rightarrow \{true, false\}$ function is defined as:

$$is\_instance(o, S, \leq_E) = \begin{cases} \text{true} & \text{if } ((\texttt{Trigger\_condition}\ S)^I = \\ & true) \wedge \Phi_T \subseteq \Omega_E \\ \text{false} & \text{else} \end{cases}$$

In a given execution graph $\leq_E$, an obligation o will be instantiated if at least one substitution S exists that specifies an instance of o. The $is\_instantiated : O \times 2^{\Phi \times \Phi} \rightarrow \{true, false\}$ function is defined as:

$$is\_instantiated(o, \leq_E) = \begin{cases} \text{true} & \text{if } \exists\, S \in \Sigma : is\_instance(o, S, \leq_E) = true \\ \text{false} & \text{else} \end{cases}$$

In a given execution graph $\leq_E$, an instance of an obligation will be fulfilled if a substitution $T \in \Sigma$ exists that extends $S \subseteq T$ and if the expressions specifying the trigger condition and obligatory processing steps are interpreted to $true$ after applying T. The $is\_instance\_fulfilled : O \times \Sigma \times 2^{\Phi \times \Phi} \rightarrow \{true, false\}$ function is defined as:

$$is\_instance\_fulfilled(o, S, \leq_E) =$$

$$\begin{cases} \text{true} & \text{if the time constraint is } \texttt{AT ANYTIME} \wedge \\ & \exists\, T \in \Sigma : S \subseteq T : (\texttt{Trigger\_condition}\ T)^I = true \wedge \\ & (\texttt{Obligatory\_processing\_steps}\ T)^I = true \wedge \\ & \Phi_T \cup \Phi_O \subseteq \Omega_E \\ \\ \text{true} & \text{if the time constraint is } \texttt{AFTERWARDS} \wedge \\ & \exists\, T \in \Sigma : S \subseteq T : (\texttt{Trigger\_condition}\ T)^I = true \wedge \\ & (\texttt{Obligatory\_processing\_steps}\ T)^I = true \wedge \\ & \forall (\phi_T, \phi_O) \in \Phi_T \times \Phi_O : \phi_T \leq_E \phi_O \wedge \\ & \Phi_T \cup \Phi_O \subseteq \Omega_E \\ \\ \text{false} & \text{else} \end{cases}$$

In a given execution graph $\leq_E$, an obligation rule will be fulfilled if all its instances are fulfilled. The $is\_fulfilled : O \rightarrow \{true, false\}$ function is defined as:

$$is\_fulfilled(o, \leq_E) = \begin{cases} \text{true} & \text{if } \forall S \in \Sigma : (is\_instance(o, S, \leq_E) \wedge \\ & \quad is\_instance\_fulfilled(o, S, \leq_E) = \\ & \quad true). \\ \text{false} & \text{else} \end{cases}$$

Based on these axioms and definitions, we define the destiny. We will call an execution graph closed if all obligations that have been instantiated have also been fulfilled. If an execution graph is closed, no obligations will be currently active or violated.

Definition 7.12: *Closed Execution Graph*

Let O_s be the set of specified obligation rules. We define an execution graph $\leq_E$ as closed if all obligations are either not instantiated or all instances of obligations are fulfilled:

$$closed(\leq_E) = \begin{cases} \text{true} & \text{if } \forall o \in O_s : is\_instantiated(o, \leq_E) \rightarrow \\ & \quad is\_fulfilled(o, \leq_E). \\ \text{false} & \text{else} \end{cases}$$

In our scenario depicted in Figure 7.1, the steps s_1 to s_7 do not instantiate any obligation. The execution graph is closed until the execution of s_8. By executing s_8 the obligation specified by `Policy 2.2` is instantiated. Because this obligation is not fulfilled by the steps s_1 to s_8, the execution graph is no longer closed.

The destiny is a subgraph of the future execution graph ($\leq_F$) consisting of only allowed processing steps ($\phi \in \Omega_A$), which extend the history to a closed execution graph ($\leq_C = \leq_H \cup \leq_D$):

Definition 7.13: *Destiny*

We define the destiny $\leq_D \subseteq \Phi \times \Phi$ as partial order of processing steps:

$$\leq_D = \{(\phi,\ \rho) |\ (\phi,\ \rho) \in \leq_F \wedge\ (\phi,\ \rho) \in \leq_A \wedge$$
$$\exists \leq_C \colon (\phi,\ \rho) \in \leq_C \wedge\ (\leq_H \subseteq \leq_C) \wedge\ (\leq_C \setminus \leq_H) \subseteq \leq_A) \wedge\ closed(\leq_C)\}.$$

In our scenario, a physician may release the record to Jane Doe after receiving it from the administration. In Figure 7.1, s_{41} and s_{40} represents these two processing steps. This solution fulfills the obligation specified by `Policy 2.2`. The steps s_1 to s_8 together with s_{40} and s_{41} constitute a closed execution graph. s_1 to s_8 are the history and s_{40} as well as s_{41} are part of the destiny.

The dependencies between history $\leq_H$, future execution $\leq_F$ and destiny $\leq_D$ are depicted in Figure 7.1.

7.4 Checking for Unfulfillable Obligations

Instantiated obligations will be unfulfillable if the future has no destiny. Thereby, the existence of a destiny has to observe the specifications of permit and denial rules, as well as obligation rules. To decide upon the existence of a destiny, we translate Care policy rules to colored Petri nets in order to use existing methods for deciding upon reachability in colored Petri nets. We specify the decision problem and make some basic assumptions before we define the translation.

7.4.1 Decision Problem and Procedure

Obligation rules specify obligatory processing steps that must be fulfilled after the obligation has been instantiated. At a given point in time, multiple obligations may be active and additional obligations may be activated during

the future execution. At the same time, a planned processing step may lead to a violation of obligations by triggering conflicting obligations, by rendering obligations unfulfillable or by instantiating obligations that violate policies. Hence, before performing the planned processing step, the actor should verify the existence of a destiny that follows the planned step. The problem is to decide whether a destiny exists or not, as the only information about the future processing is the given set of policy rules and the current history:

Definition 7.14: *Deciding the Existence of a Destiny*

An applicable decision procedure will return `true` if for a given set of policy rules and a given history a destiny exists, else it will return `false`.

Real world business processes must be executable in finite time implying the reachability of all obligatory processing steps in a finite partial order of steps. If a destiny exists, it must be a finite subgraph of the future execution graph. One can verify the existence of the destiny by verifying the reachability of all current and future obligatory processing steps within the finite subgraph of the future execution graph. The obligatory processing steps must be reachable by means of allowed processing steps. Search algorithms can be used to find such a sub-graph. Starting with an open domain unspecified many alternative processing steps can be performed. For instance, an unspecified number of recipients can be chosen if it is permitted to transfer a data instance to anybody. To achieve decidability, we are required to have a finite number of alternatives in each step. As the open domain is restricted by the policy rules, we use these as a starting point for our decision procedure.

Care provides a decision procedure by reducing the decision problem of the existence of a destiny to the well-defined decision problem of the reachability of nodes in colored Petri nets. The given policy rules are translated under consideration of the current history to a colored Petri net $CPN = (\Sigma, P, T, A, N, C, G, E, I)$. Care decides the existence of a destiny by deciding the reachability of the transitions representing the future obligation.

7.4.2 Assumptions

We make a few assumptions about the conditions under which the policy language Care is used. We assume the stability of the set of policy rules, i.e. that the set of given policy rules does not change during the process execution.

Assumption 7.1: *Stability of the Set of Policies*

The given set of policy rules does not change during the execution of a process.

Missing or wrong steps in the history can lead to misinterpretations of obligation rules. Hence, we assume the correctness of the history $\leq_H$:

Assumption 7.2: *Correctness of the History*

All processing steps that have been performed during the process execution are part of the history $\leq_H$ and the information about these steps is correct. The history does not contain additional processing steps that have not been performed.

To assure that instantiated obligations stay instantiated, we assume the monotony of the history $\leq_H$:

Assumption 7.3: *Monotony of the History*

By progressing the execution of a process, processing steps may only be added to the history $\leq_H$ of this process execution and may not be removed from the history.

7.4.3 Reducing the Decision Problem

To answer the decision problem, we translate the given policy rules in normal form to a colored Petri net. The translation has to comply with the axioms specified in Section 7.3.2. The inputs of the translation are the specified permit and deny rules in normal form as well as the history and the given domain knowledge. The output is the colored Petri net representing the future execution graph: $CPN = (\Sigma, P, T, A, N, C, G, E, I)$.

The modeling of CPN is based on the definition of DiALog in Chapter 4. In DiALog, processing steps are modeled as transitions and each transition is connected to the place representing the actor performing the action. Care represents actors as parts of tokens. The tokens represent the possession of data instances by entities. In Care, permit rules specify permitted processing steps which we represent as transitions in CPN. While each transition models exactly one sort of processing step in DiALog, one transition represents a set of permitted processing steps in Care. The guard expression of the transition specifies this set of processing steps. Deny rules further restrict the permitted processing steps. In CPN, these restrictions are represented by adapting the guard expressions accordingly. We can use the guard expressions to specify the allowed processing steps as syntax and semantics of Care do not support function symbols. Without function symbols, we cannot generate infinitely many elements. This circumstance leads to Care having a finite model. Because of this property, we can use Datalog as implementation, as we did for Papel in Chapter 6. In CPN, we specify the guard expressions using the Datalog semantics.

The tokens representing the possession consist of triples. They have the following syntax `(actor, data_instance, log)`. The first element represents the actor possessing the data instance. This representation of possession in the colored Petri net adheres to Axiom 7.3. As CPN models the processing of all instances of one data item, the second element identifies the possessed instance. The third element of the triple is the log associated with the data instances. A single place $p \in P$ contains all these tokens.

Policy rules specify entities by identifying named entities or by using instance of concepts, such as nurses and patients in `Policy 8.3` '*nurses must transfer health records to the archive after the patient's discharge'*. The first element of the possession triple represents either the named entity or an unnamed instance of a certain concept. In the scenario, Jane Doe is a named entity and patient is a concept. In CPN, Jane Doe is identified by a reference, such as `jane_doe`. An instance of an unnamed patient is represented by a reference to the concept $\texttt{patient}_1$. The index is used to clearly distinguish different instances of the concept.

Each permit rule is represented as transitions $t \in T$ connected to p by arcs $a \in A$ with the node function N and the arc expression function E. The guard function $G(t)$ specifies the constraints specified by the policy condition, such as '*all data instances, which are health records*' or '*all data instances, which have been updated before*'. The occurrence of a transition represents an allowed processing step. Most transitions are modeled as defined in Chapter 4. The definition of `copy` actions differs from the definition in DiALog by not having an involved agent. The refinement of the `copy` action is depicted in Figure 7.3 as well as in Definition 7.15.

Definition 7.15: *Copy Action in Colored Petri Nets*

Let $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ be a colored Petri net and let $p \in P$ be the place representing the entity copying the data instance. We model copy actions as

- a transition $t \in T$ and
- three arcs $a_i, a_o, a_d \in A$ with
- node functions $N(a_i) = (p, t)$, $N(a_o) = (t, p)$, and $N(a_d) = (t, p)$ and with
- arc expressions $E(a_i) = u$ and $E(a_o) = u$, where u is declared as `var u :` $\mathbb{N}_0$, and $E(a_d) = v$, where v is declared as

`var v :` $\mathbb{N}_0$.

Let $p_c \in P$ be the place modeling the counter, t is connected to the counter by two arcs $c_i, c_o \in A$ with

- the node functions $N(c_i) = (t, p_c)$ and $N(c_o) = (p_c, t)$, and
- the arc expressions $E(c_i) = v + 1$ and $E(c_o) = v$, where v is the same variable as in the arc expression $E(a_d)$.

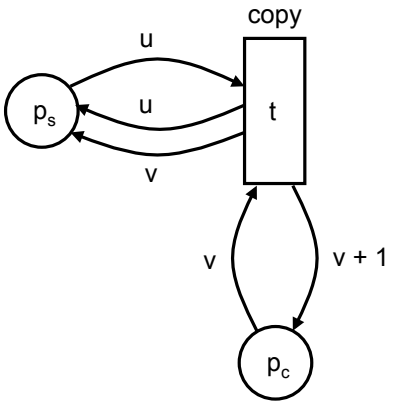

Figure 7.3: Colored Petri Net Representation of Copy Actions in Care.

Care introduces `merge` actions as action category to model the explicit parallel processing postulated by Axiom 7.1. The `merge` actions are defined by Definition 7.16 and depicted in Figure 7.4.

Definition 7.16: *Merge Action in Colored Petri Nets*

Let $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ be a colored Petri net and let $p \in P$ be the place representing the entity merging the data instances. We model merge actions as

- a transition $t \in T$ and
- three arcs $a_i, a_o, a_s \in A$ with
- node functions $N(a_i) = (p, t)$, $N(a_o) = (t, p)$, and $N(a_s) = (p, t)$ and with
- arc expressions $E(a_i) = u$ and $E(a_o) = u$, where u is declared as `var u : ` $\mathbb{N}_0$, and $E(a_s) = v$, where v is declared as `var v : ` $\mathbb{N}_0$.

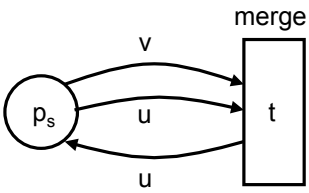

Figure 7.4: Colored Petri Net Representation of Merge Actions in Care.

Step 1 of the translation describes how to represent permissions in CPN. The condition of the permit rules $permit_i$, which are given in normal from, are translated to transitions $t_i \in T$. Based on the output of this step, Step 2 specifies the representation of denial. As deny rules restrict the permitted processing steps, they are represented by extending the guard expressions of the transitions. The extension of the expression implements the specified restriction.

An additional place $p_e \in P$ is required to implement the change of possession through transfer actions. This place contains tokens representing all

named actors as well as tokens representing variable actors of certain concepts. The named actors and concepts can be derived from the history and the given domain knowledge. The modeling of p_e and its connection with transfer actions is depicted in Figure 7.5. The initial marking of the place p_e is derived in Step 3.

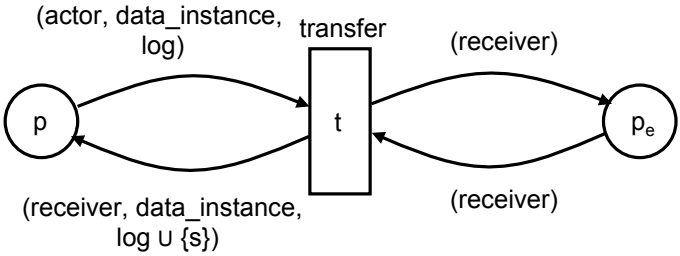

Figure 7.5: Transfer Action with Possession Change.

Considering Axiom 7.2 a data instance may not exist before it is created by a `create` or `copy` action. We implement this axiom by using a counter place p_c as defined by DiALog in Section 4.2.2. The modeling of `create` and `copy` actions with a counter place guarantees that all data instances have a unique identifier. Likewise, `delete` and `merge` actions consume the token representing the deleted data instance. By consuming the tokens the associated data instance cannot be processed anymore as postulated by Axiom 7.4. During its lifespan, the data instance is processed by actors possessing the instance.

To translate a given set of policy rules in normal form to a colored Petri net, we require the set to be finite and to consist of finitely long rules. In Papel, only policy conditions referring to a finite number of different elements are allowed. Conditions as '*infinitely many different actors have to read this record*' cannot be specified. Papel does also not support universal quantification without restricting the bound variables to a finite set of specific

elements. The rule '*all nurses have to read the record*' can only be verified if the set of all nurses is specified.

Obligation rules are not considered during the translation as the required processing steps will be already represented if they are allowed. To complete the translation the initial marking of p has to represent the current state of possession. The initial marking is derived in Step 4.

After translating all policy rules, we can decide the existence of a destiny by deciding whether a certain occurrence sequence exists. In CPN, the occurrence of a transitions simulates the execution of a processing step. Through the variables bound by the arc expressions, each occurrence of a transition clearly represents one specific processing step. An occurrence sequence represents an execution graph. If an occurrence sequence exists which represents a closed execution graph, a destiny will exist. As the reachability of transition is decidable in colored Petri nets [Mayr, 1981], the existence of such a sequence can be decided by deciding the reachability of the transitions representing the obligatory processing steps.

(Step 1) Representing Permissions in CPN

A permit rule defines permitted processing steps by specifying the affected data instances, entities, action categories and processing steps. Each permit rule $permit_i$ is translated to a transition t_i as specified in Table 7.1 and depicted in Example 7.4. In normal form, permit rules have the following syntax `permit (ID) Step_Pattern AND Constraint`. The step pattern is translated to a transition with arcs, arc expressions and guard expression. If variables occur in the condition, a substitution S is defined to match the variables with parameters specified by the arc expression of the transition. After translating the step pattern, the according substitution S_i is applied to the constraint. The resulting expression is attached to the guard expression $G(t_i) \leftarrow G(t_i)$'+(`Constraint` S_i)'.

Table 7.1: Translation of Permissions to Transitions (Part 1).

Construct	Step Pattern	CPN Elements
named `Data`	`step(`δ`, ...)`	$G(t_i) \leftarrow$ '(`data_instance` $= \delta$)'
variable `Data` w/o constraint	`step(_, ...)`	$G(t_i) \leftarrow$'$true$'
variable `Data` with constraint	`step(`D`, ...)`	$S_i \leftarrow S_i \cup \{D \mapsto$ `data_instance`$\}$
named `SecondData`	`step(...,` η`, ...)`	$G(t_i) \leftarrow$ '('$+G(t_i)+$')' $\wedge$ (`second_data_instance` $= \eta$)'
variable `SecondData` w/o constraint	`step(..., _, ...)`	
variable `SecondData` with constraint	`step(...,` E`, ...)`	$S_i \leftarrow S_i \cup$ $\{E \mapsto$ `second_data_instance`$\}$
named `Actor`	`step(...,` α`, ...)`	$G(t_i) \leftarrow$ '('$+G(t_i)+$')' $\wedge$ (`actor` $= \alpha$)'
variable `Actor` w/o constraint	`step(..., _, ...)`	
variable `Actor` with constraint	`step(...,` A`, ...)`	$S_i \leftarrow S_i \cup \{A \mapsto$ `actor`$\}$
named `Receiver`	`step(...,` β`, ...)`	$G(t_i) \leftarrow$ '('$+G(t_i)+$')' $\wedge$ (`receiver` $= \beta$)'
variable `Receiver` w/o constraint	`step(..., _, ...)`	
variable `Receiver` with constraint	`step(...,` B`, ...)`	$S_i \leftarrow S_i \cup \{B \mapsto$ `receiver`$\}$

Table 7.2: Translation of Permissions to Transitions (Part 2).

Construct	Step Pattern	CPN Elements
named `Category`	`step(..., create, ...)`	$T \leftarrow T \cup \{t_i\}$ $A \leftarrow A \cup \{a_{i,o}, a_{i,ci}, a_{i,co}\}$ $N(a_{i,ci}) \leftarrow (p_c, t_i)$ $E(a_{i,ci}) \leftarrow$ `'(data_instance, {})'` $N(a_{i,o}) \leftarrow (t_i, p)$ $E(a_{i,o}) \leftarrow$ `'(actor, data_instance, {step( data_instance, NULL, actor, NULL, create, getID(), NULL)})'` $N(a_{i,co}) \leftarrow (t_i, p_c)$ $E(a_{i,co}) \leftarrow$ `'(data_instance + 1, {})'`
named `Category`	`step(..., read, ...)`	$T \leftarrow T \cup \{t_i\}$ $A \leftarrow A \cup \{a_{i,i}, a_{i,o}\}$ $N(a_{i,i}) \leftarrow (p, t_i)$ $E(a_{i,i}) \leftarrow$ `'(actor, data_instance, log)'` $N(a_{i,o}) \leftarrow (t_i, p)$ $E(a_{i,o}) \leftarrow$ `'(actor, data_instance, log ∪{ step(data_instance, NULL, actor, NULL, read, getID(), getPIDs(log))})'`
named `Category`	`step(..., update, ...)`	$T \leftarrow T \cup \{t_i\}$ $A \leftarrow A \cup \{a_{i,i}, a_{i,o}\}$ $N(a_{i,i}) \leftarrow (p, t_i)$ $E(a_{i,i}) \leftarrow$ `'(actor, data_instance, log)'`

Table 7.3: Translation of Permissions to Transitions (Part 3).

Construct	Step Pattern	CPN Elements
		$N(a_{i,o}) \leftarrow (t_i, p)$ $E(a_{i,o}) \leftarrow$ '(actor, data_instance, log $\cup\{$ step(data_instance, NULL, actor, NULL, update, getID(), getPIDs(log))})'
named Category	step(..., transfer, ...)	$T \leftarrow T \cup \{t_i\}$ $A \leftarrow A \cup \{a_{i,i}, a_{i,o}, a_{i,ei}, a_{i,eo}\}$ $N(a_{i,i}) \leftarrow (p, t_i)$ $E(a_{i,i}) \leftarrow$ '(actor, data_instance, log)' $N(a_{i,o}) \leftarrow (t_i, p)$ $E(a_{i,o}) \leftarrow$ '(receiver, data_instance, log $\cup\{$ step(data_instance, NULL, actor, receiver, transfer, getID(), getPIDs(log))})' $N(a_{i,ei}) \leftarrow (p_e, t_i)$ $E(a_{i,ei}) \leftarrow$ '(receiver)' $N(a_{i,eo}) \leftarrow (t_i, p_e)$ $E(a_{i,eo}) \leftarrow$ '(receiver)'
named Category	step(..., copy, ...)	$T \leftarrow T \cup \{t_i\}$ $A \leftarrow A \cup \{a_{i,i}, a_{i,o}, a_{i,d}, a_{i,ci}, a_{i,co}\}$ $N(a_{i,i}) \leftarrow (p, t_i)$ $E(a_{i,i}) \leftarrow$ '(actor, data_instance, log)' $N(a_{i,ci}) \leftarrow (p_c, t_i)$ $E(a_{i,ci}) \leftarrow$ '(second_data_instance, {})' $N(a_{i,o}) \leftarrow (t_i, p)$

Table 7.4: Translation of Permissions to Transitions (Part 4).

Construct	Step Pattern	CPN Elements
		$E(a_{i,o}) \leftarrow$ '(actor, data_instance, log $\cup$ { step(data_instance, second_data_instance, actor, NULL, copy, getID(), getPIDs(log))})' $N(a_{i,d}) \leftarrow (t_i, p)$ $E(a_{i,d}) \leftarrow$ '(actor, second_data_instance, { step(second_data_instance, NULL, actor, NULL, create, getID(), NULL})' $N(a_{i,co}) \leftarrow (t_i, p_c)$ $E(a_{i,co}) \leftarrow$ '(second_data_instance + 1, {})'
named Category	step(..., merge, ...)	$T \leftarrow T \cup \{t_i\}$ $A \leftarrow A \cup \{a_{i,i}, a_{i,o}, a_{i,s}\}$ $N(a_{i,i}) \leftarrow (p, t_i)$ $E(a_{i,i}) \leftarrow$ '(actor, data_instance, log)' $N(a_{i,s}) \leftarrow (p, t_i)$ $E(a_{i,s}) \leftarrow$ '(actor, second_data_instance, second_log)' $N(a_{i,o}) \leftarrow (t_i, p)$ $E(a_{i,o}) \leftarrow$ '(actor, data_instance, log $\cup$ second_log $\cup$ {

Table 7.5: Translation of Permissions to Transitions (Part 5).

Construct	Step Pattern	CPN Elements
		`step(data_instance, second_data_instance, actor, NULL, merge, getID(), getPIDs(log))})'`
named `Category`	`step(..., delete, ...)`	$T \leftarrow T \cup \{t_i\}$ $A \leftarrow A \cup \{a_{i,i}\}$ $N(a_{i,i}) \leftarrow (p, t_i)$ $E(a_{i,i}) \leftarrow$ '`(actor, data_instance, log)`'
variable w/o constraint `Category`	`step(..., _, ...)`	In this case a transition for each action category must be added according to the translation of the different categories.
variable with constraint `Category`	`step(...,` X`, ...)`	In this case a transition for each action category must be added according to the translation of the different categories with $S_i \leftarrow S_i \cup \{X \mapsto$ *category*$\}$ where *category* is the actual category.
named `ID`	`step(...,` ϕ`, ...)`	$G(t_i) \leftarrow$ '('$+G(t_i)+$')' $\wedge$ (`getID()` $= \phi$)'
variable `ID` w/o constraint	`step(..., _, ...)`	
variable `ID` with constraint	`step(...,` V`, ...)`	$S_i \leftarrow S_i \cup$ $\{V \mapsto$ `getID()` $\}$
named `PIDs`	`step(...,` ρ`)`	$G(t_i) \leftarrow$ '('$+G(t_i)+$')' $\wedge$ (`getPIDs(log)` $= \rho$)'
variable `PIDs` w/o constraint	`step(..., _)`	

Table 7.6: Translation of Permissions to Transitions (Part 6).

Construct	Step Pattern	CPN Elements
variable `PIDs` with constraint	`step(..., R)`	$S_i \leftarrow S_i \cup \{R \mapsto$ `getPIDs(log)` $\}$
`getID()` returns the ID of the current step.		
`getPIDs(log)` derives the ID of the directly preceding steps from the history.		

Example 7.4: *Translating Permit Rules*

In this and the following examples, we depict the application of the reduction algorithm by translating the Policies 2, 8 and 9 of our scenario. We depict the detailed application of Step 1 by translating the permission defined by Policy 2.1: *Everyone is permitted to transfer Jane Doe's health record to Jane Doe:*

```
permit (ID) step (record_jd, _, _, jane_doe,
                  transfer, ID, _).
```

The result of the translation is also depicted in Figure 7.6. The step pattern of this permission rule is `step (record_jd, _, _, jane_doe, transfer, ID, _)`. We translate the named data instance δ = `record_jd` to the following guard expression: $G(t_{2.1})$ = '(`data_instance` = `record_jd`)' as specified in Table 7.1. The variable second data instance and the variable actor do not require any translation. The named receiver β = `jane_doe` is translated to the following extended guard expression: $G(t_{2.1})$ = '(`data_instance` = `record_jd`) $\wedge$ (`receiver` = `jane_doe`)'. The action category of the step pattern is `transfer`. We add the transition $t_{2.1}$ to the set of transitions $T \leftarrow T \cup \{t_{2.1}\}$. The transition

is connected to p by two arcs $a_{2.1,i}$ and $a_{2.1,o}$. The transition is also connected to p_e by two arcs $a_{2.1,ei}$ and $a_{2.1,eo}$. We update the set of arcs $A \leftarrow A \cup \{a_{2.1,i}, a_{2.1,o}, a_{2.1,ei}, a_{2.1,eo}\}$ and specify the arc function of these arcs $N(a_{2.1,i}) = (p, t_{2.1})$, $N(a_{2.1,o}) = (t_{2.1}, p)$, $N(a_{2.1,ei}) = (p_e, t_{2.1})$ and $N(a_{2.1,eo}) = (t_{2.1}, p_e)$. The arc expressions of the added arcs are specified as: $E(a_{2.1,i})$ = '`(actor, data_instance, log)`', $E(a_{2.1,o})$ = '`(receiver, data_instance, log` $\cup\{$`step(data_instance, NULL, actor, receiver, transfer, getID(), getPIDs(log))`$\}$', $E(a_{2.1,ei})$ = '`(receiver)`', and $E(a_{2.1,eo})$ = '`(receiver)`'. The identifier of the processing step is specified by the variable `ID`, which we translate to the substitution $S_{2.1} = \{$`ID` $\mapsto$ `getID()`$\}$. As the permit rule has no constraint, the substitution is not used. The identifier set of the preceding processing steps is specified by a variable without constraint and requires no translation.

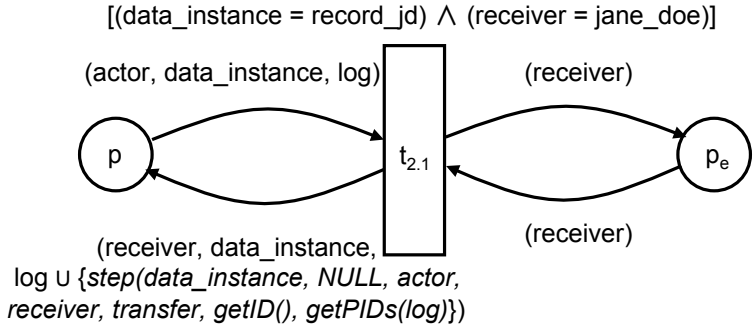

Figure 7.6: Translation result of Policy 2.1.

We translate Policy 8.3: *The staff is permitted to transfer health records to other staff:*

```
permit (ID) step (H, _, S, T, transfer, ID, _) AND
```

```
            instance_of (H, health_record) AND
            instance_of (S, staff) AND
            instance_of (T, staff).
```

analogously to the following elements of CPN:

$T \leftarrow T \cup \{t_{8.3}\}$.

$G(t_{8.3})$ = '`(instance_of (data_instance, health_record) AND instance_of (actor, staff) AND instance_of (receiver, staff))`'.

$A \leftarrow A \cup \{a_{8.3,i},\ a_{8.3,o},\ a_{8.3,ei},\ a_{8.3,eo}\}$.

$N(a_{8.3,i}) = (p, t_{8.3})$, $N(a_{8.3,o}) = (t_{8.3}, p)$, $N(a_{8.3,ei}) = (p_e, t_{8.3})$ and $N(a_{8.3,eo}) = (t_{8.3}, p_e)$.

$E(a_{8.3,i})$ = '`(actor, data_instance, log)`', $E(a_{8.3,o})$ = '`(receiver, data_instance, log` $\cup\{$`step(data_instance, NULL, actor, receiver, transfer, getID(), getPIDs(log))`$\}$)', $E(a_{8.3,ei})$ = '`(receiver)`', and $E(a_{8.3,eo})$ = '`(receiver)`'.

We also translate Policy 9.1: *Bob is permitted to delete health records:*

```
permit (ID) step (H, _, bob, _, delete, ID, _) AND
            instance_of (H, health_record).
```

to the following elements of CPN:

$T \leftarrow T \cup \{t_{9.1}\}$.

$G(t_{9.1})$ = '(actor = bob) $\wedge$ (instance_of (data_instance, health_record))'.

$A \leftarrow A \cup \{a_{9.1,i}\}$.

$N(a_{9.1,i}) = (p, t_{9.1})$.

$E(a_{9.1,i})$ = '(actor, data_instance, log)'.

Figure 7.7 depicts CPN after the translation of all three permit rules.

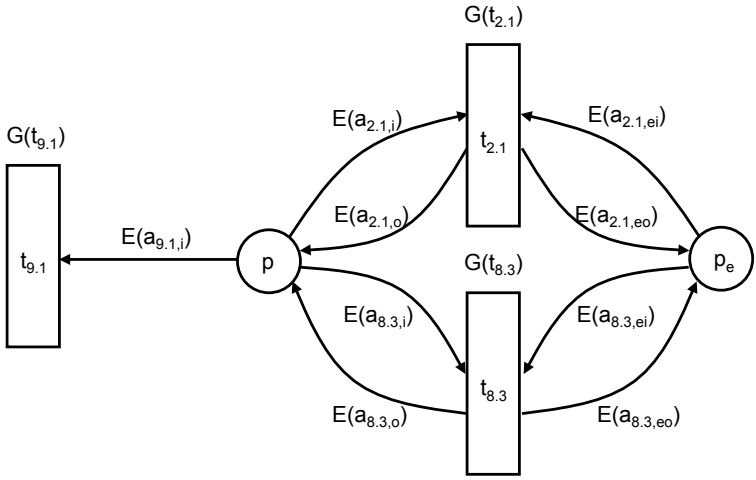

Figure 7.7: Representation of the Policy Rules 2.1, 8.3 and 9.1.

(Step 2) Representing Denial in CPN

Deny rules define restrictions to the permitted processing steps. In CPN, the permitted steps are represented by transitions t_i. We translate each deny rule $deny_j$ to an extension g_j of the guard expressions $G(t_i)$ of these transitions. If variables occur in a condition and constraint of a deny rule, a substitution S_j will be defined to substitute the variables of the constraint with the according parameters of the transition. As g_j express restrictions, they are negated and attached to the guard expressions: $\forall t_i \in T : G(t_i) = G(t_i)+\text{'}\wedge\neg(g_j \wedge (\texttt{Constraint}\ \ S_j))\text{'}$. We translate one deny rule $deny_j$ after the other as depicted in Table 7.7. Example 7.5 shows the translation of one deny rule of our scenario.

Table 7.7: Translation of Deny Rules to Guard Expressions (Part 1).

Construct	Step Pattern	CPN Elements
named `Data`	`step(`δ`, ...)`	$g_j \leftarrow\ g_j\text{'}+\text{'}(\texttt{data\_instance} = \delta)\text{'}$
variable `Data` w/o constraint	`step(_, ...)`	$g_j \leftarrow \text{'}true\text{'}$
variable `Data` with constraint	`step(`D`, ...)`	$S_j \leftarrow\ S_j \cup \{D \mapsto \texttt{data\_instance}\}$
named `SecondData`	`step(..., `η`, ...)`	$g_j \leftarrow \text{'}(\text{'}+g_j+\text{'}) \wedge (\texttt{second\_data\_instance} = \eta)\text{'}$
variable `SecondData` w/o constraint	`step(..., _, ...)`	
variable `SecondData` with constraint	`step(..., `E`, ...)`	$S_j \leftarrow\ S_j \cup \{E \mapsto \texttt{second\_data\_instance}\}$
named `Actor`	`step(..., `α`, ...)`	$g_j \leftarrow \text{'}(\text{'}+g_j+\text{'}) \wedge \text{'}(\texttt{actor} = \alpha)\text{'}$
variable `Actor` w/o constraint	`step(..., _, ...)`	$g_j \leftarrow\ g_j$

Table 7.8: Translation of Deny Rules to Guard Expressions (Part 2).

Construct	Step Pattern	CPN Elements
variable `Actor` with constraint	`step(...,` A`, ...)`	$S_j \leftarrow S_j \cup \{A \mapsto$ `actor`$\}$
named `Receiver`	`step(...,` β`, ...)`	$g_j \leftarrow$ '('$+g_j+$')' $\wedge$ (`receiver` $= \beta$)'
variable `Receiver` w/o constraint	`step(...,` _`, ...)`	
variable `Receiver` with constraint	`step(...,` B`, ...)`	$S_j \leftarrow S_j \cup \{B \mapsto$ `receiver`$\}$
named `Category`	`step(...,` χ`, ...)`	$g_j \leftarrow$ '('$+g_j+$')' $\wedge$ ($\mathit{category} = \chi$)' where *category* is the category of the transition t_i
variable `Category` w/o constraint	`step(...,` _`, ...)`	
variable `Category` with constraint	`step(...,` X`, ...)`	$S_j \leftarrow S_j \cup \{X \mapsto \mathit{category}\}$ where *category* is the category of the transition t_i
named `ID`	`step(...,` ϕ`, ...)`	$g_j \leftarrow$ '('$+g_j+$')' $\wedge$ (`getID()` $= \phi$)'
variable `ID` w/o constraint	`step(...,` _`, ...)`	
variable `ID` with constraint	`step(...,` V`, ...)`	$S_j \leftarrow S_j \cup \{V \mapsto$ `getID()`$\}$
named `PIDs`	`step(...,` ρ`)`	$g_j \leftarrow$ '('$+g_j+$')' $\wedge$ (`getPIDs(log)` $= \rho$)'
variable `PIDs` w/o constraint	`step(...,` _`)`	
variable `PIDs` with constraint	`step(...,` R`)`	$S_j \leftarrow S_j \cup \{R \mapsto$ `getPIDs(log)`$\}$

`getID()` returns the ID of the current step.

`getPIDs(log)` derives the ID of the directly preceding steps from the history.

Example 7.5: *Translating Policy 8.2*

We depict the application of Step 2 by translating `Policy 8.2`:

```
deny (ID) step (H, _, archive, P, transfer,
                ID, _) AND
          instance_of(H, health_record) AND
          instance_of(P, patient).
```

The deny rule consists of a step pattern `step (H, _, archive, P, transfer, ID, _)` and a constraint `instance_of(H, health_record) AND instance_of(P, patient)`. The translation is done as specified by Table 7.7.

The step pattern of `Policy 8.2` specifies a variable data instance `H`. As the policy specifies a constraint, the variable data instance is translated to a substitution $S_{8.2} = \{\texttt{H} \mapsto \texttt{data\_instance}\}$. The variable second data instance is specified without relation to the constraint and requires no translation. The named actor $\alpha = \texttt{archive}$ is translated to a guard expression $g_{8.2}$ ='$(\texttt{actor} = \texttt{archive})$'. A variable with constraint specifies the receiver. We extend the substitution accordingly $S_{8.2} = \{\texttt{H} \mapsto \texttt{data\_instance}, \texttt{P} \mapsto \texttt{receiver}\}$. The action category is specified as `transfer` and the according translation extends the guard expression $g_{8.2}$ ='$(\texttt{actor} = \texttt{archive}) \wedge (\textit{category} = \texttt{transfer})$'. The place holder *category* will be replaced when the guard expression is attached to each guard expressions representing processing steps in CPN. A variable specifies the identifier of the processing step. The translation of the variable extends the substitution $S_{8.2} = \{\texttt{H} \mapsto \texttt{data\_instance}, \texttt{P} \mapsto \texttt{receiver}, \texttt{ID} \mapsto \texttt{getID()}\}$. The specification of the preceding processing steps does not require any translation.

After the step pattern is translated, the variables in the constraint are substituted as specified by S_j. The resulting constraint

instance_of(data_instance, health_record) AND instance_of(receiver, patient) is added to the guard expression $g_{8.2}$ = '(actor = archive) $\wedge$ (*category* = transfer) $\wedge$ (instance_of(data_instance, health_record) AND instance_of(receiver, patient))'. The negation of $g_{8.2}$ is attached to the guard expressions of the transitions representing permitted processing steps. Based on the Example 7.4, we have one transition $t_{2.1}$. As $t_{2.1}$ is a transfer action the place holder *category* is replaced accordingly leading to the updated guard expression $G(t_{2.1})$ = '(data_instance = record_jd) $\wedge$ (receiver = jane_doe) $\wedge \neg$ ((actor = archive) $\wedge$ (transfer = transfer) $\wedge$ (instance_of(data_instance, health_record) AND instance_of(receiver, patient)))'.
The guard expressions of the transitions $t_{8.3}$ and $t_{9.1}$ are updated accordingly leading to $G(t_{8.3})$ = '(instance_of (data_instance, health_record) AND instance_of (actor, staff) AND instance_of (receiver, staff)) $\wedge \neg$ ((actor = archive) $\wedge$ (transfer = transfer) $\wedge$ (instance_of(data_instance, health_record) AND instance_of (receiver, patient)))' and $G(t_{9.1})$ = '(actor = bob) $\wedge$ (instance_of (data_instance, health_record)) $\wedge \neg$ ((actor = archive) $\wedge$ (delete = transfer) $\wedge$ (instance_of(data_instance, health_record) AND instance_of(receiver, patient)))'.

(Step 3) Representing Entities in CPN

In this step, we derive tokens representing all entities specified by instances or concepts in policy rules or the domain ontology. The derived tokens are added to the place p_e. Instances are translated to tokens representing exactly this instance, e.g. one token in p_e represents Jane Doe

($I(p_e) \leftarrow I(p_e) \cup \{$ `(jane_doe)` $\}$). For each concept a token representing an instance of the concept is added. To fulfill certain policy rules, it is necessary to change these tokens. The policy condition '*three different physicians are required to confirm the examination results*' requires the representation of three unnamed physicians. Hence, we use indices to identify different instances of a concept, e.g. for the medical staff a token $($`staff`$_i)$ is added: $I(p_e) \leftarrow I(p_e) \cup \{$ `(staff`$_i$`)` $\}$.

In policy rules, '_' specifies variable entities without constraints. Any entity matches this specification. We introduce the *everyone* concept as the according domain concept. This concept is the root concept of all other concepts. In CPN, the *everyone* concept is represented like other domain concepts.

To generate new instances of a concept a special transition $t_{generate}$ is connected with the place p_e. The connection is done by two arcs, which are an incoming arc c_i with node function $N(c_i) = (p_e, t_{generate})$ and an outgoing arc c_o with $N(c_o) = (t_{generate}, p_e)$. The arc expressions are $E(c_i) =$ `(concept`$_i$`)` and $E(c_o) =$ `(concept`$_i$`), (concept`$_{i+1}$`)`. Each occurrence of this transition generates a new token to represent another instance of the concept represented by the token enabling the occurence.

Example 7.6: *Representing Entities*

The Policy 2.1 '*Everyone is permitted to transfer Jane Doe's health record to Jane Doe*' relates to everyone and Jane Doe. To represent the *everyone* concept and the named entity Jane Doe, we add two tokens to the initial marking of p_e:

$I(p_e) \leftarrow I(p_e) \cup \{$ `(everyone`$_1$`),(jane_doe)` $\}$.

Policy 8.1 '*Nurses must transfer health records to the archive after the patient's discharge*', Policy 8.2 '*The archive is denied to transfer health records to patients*' and Policy 8.3 '*The staff is permitted to transfer health*

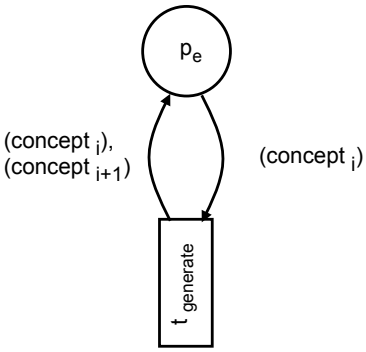

Figure 7.8: Colored Petri Net Modeling the Generation of Entity Instances.

records to other staff' introduces the *nurse*, *patient* and *staff* concepts and the named entity archive. We update the initial marking of p_e accordingly:

$I(p_e) \leftarrow I(p_e) \cup \{\texttt{(nurse}_1\texttt{)}, \texttt{(patient}_1\texttt{)}, \texttt{(staff}_1\texttt{)}, \texttt{(archive)}\}$.

The Policy 9.1 '*Bob is permitted to delete health records*' relates to Bob:

$I(p_e) \leftarrow I(p_e) \cup \{\texttt{(bob)}\}$.

Figure 7.9 depicts CPN after the translation of all policy rules including the $t_{generate}$ transition.

(Step 4) Deriving the Initial Marking

We derive the initial marking of the place p from the current distribution of

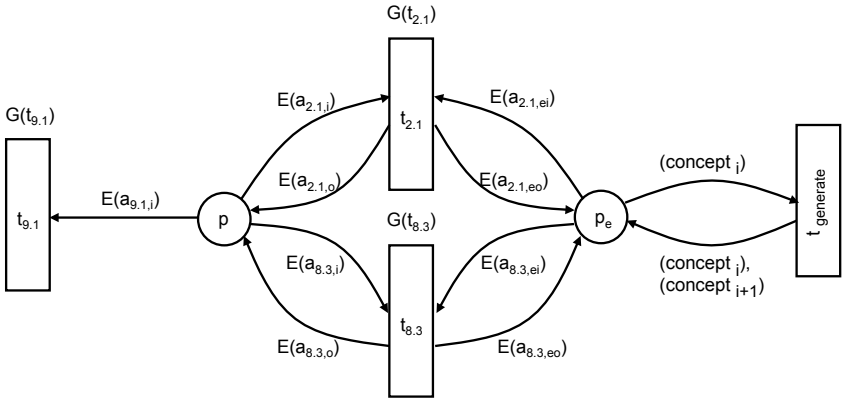

Figure 7.9: Representation of the Policies 2, 8 and 9.

the data. The current distribution of the data can be derived from the history.

Example 7.7: *Deriving the Initial Marking*

In our scenario, the record is stored at the archive after Jane Doe is discharged. As the archive is in possession of a record, we add a token (archive$_1$, record_JD, log) to the place p by specifying the initialization function of this place $I(p) \leftarrow I(p) \cup \{(\text{archive}_1, \text{record\_JD}, \text{log})\}$.

7.4.4 Discussion

As the reachability of transitions in colored Petri nets is decidable [Mayr, 1981], the existence of a destiny will be decidable if the decision problem can be reduced to the decision problem of the reachability. The introduced translation algorithm represents an implementation of the decision problem.

The translation reduces the problem to decide the existence of a destiny in a future execution graph specified by a given set of policy rules and a given history to the problem to decide the reachability of certain transitions in a colored Petri net.

Based on the specified obligation rules one can derive a transition that will only occur if the processing history of a data instance is closed. By this transition, the tokens representing data instances with a closed history are removed from the colored Petri net. Care decides the existence of a destiny by deciding the reachability of a state where all data instances have been removed.

7.5 Related Work

There are a wide range of general purpose policy languages that support obligations and are applicable for our purpose (e.g. KAoS [Bradshaw et al., 2003], Ponder [Lupu and Sloman, 1999], Rei [Kagal et al., 2003], XACML [Moses et al., 2005]). Table 7.9 gives an overview of related work in the field of policy languages supporting obligations. The table compares the different properties of policy languages that are crucial for expressing obligations and deciding the existence of a destiny. To support obligations (Property 1), a policy language has to specify a syntax (Property 1.1). Apart from a syntax definition, the policy language requires formal semantics (Property 1.2). To decide whether future obligations can be met, the formal semantics have to support future obligations (Property 1.3) and the policy language has to provide a decision procedure (Property 1.4). The availability of provenance information is crucial for interpreting obligations rules (Property 2). (Property 3) and (Property 4) depict whether the policy language specifies data flow or access control policies.

Care makes use of the main policy elements ‘denial’, ‘permission’, and ‘obligation’ as defined in the *eXtensible Access Control Markup Language (XACML)* [Moses et al., 2005]. XACML cannot easily specify policies,

Table 7.9: Comparison of Approaches Related to Care.

Approach ╱ # Property	1 Obligations	1.1 Syntax	1.2 Semantics	1.3 Future Obligations	1.4 Decision Procedure	2 Provenance	3 Data Flow	4 Access Control
Care	✓	✓	✓	✓	✓	✓	✓	✓
XACML [Moses et al., 2005]	✓	✓	-	-	-	-	-	✓
Ponder [Lupu and Sloman, 1999]	✓	✓	-	-	-	✓	-	✓
Rei [Kagal et al., 2003]	✓	✓	✓	-	-	-	-	✓
XrML [Wang et al., 2002]	✓	✓	-	-	-	-	-	✓
ODRL [Ianella, 2007]	✓	✓	(1)	-	-	-	-	✓
PTLTL FO [Bauer et al., 2009]	-	-	-	-	-	✓	-	✓
[Hilty et al., 2005]	✓	✓	✓	-	-	✓	-	✓

(1) The authors of [Pucella and Weissman, 2006] introduce a semantic definition for parts of ODRL.

which are based on complex environmental knowledge such as provenance information. As Papel extends the expressiveness of XACML conditions to relate to provenance, we based Care on Papel. Care extends Papel by introducing syntax and semantics of obligations.

Instead of using colored Petri nets [Jensen, 1992] to decide the reachability, one can use search algorithms (cf. [Russell and Norvig, 2003]) or

planning approaches, like STRIPS [Fikes and Nilsson, 1971] or ADL [Pednault, 1989]. While STRIPS and ADL are state-based, COLLAGE [Lansky, 1994] presents an action-based planning approach based on temporal logic. One can also consider temporal logic [Pnueli, 1981] instead of colored Petri nets. Temporal logic is state-based, while the obligations defined by most policy languages are action-based (e.g. [Bradshaw et al., 2003, Lupu and Sloman, 1999, Kagal et al., 2003, Moses et al., 2005, Wang et al., 2002]). Using temporal logic would require the definition of an additional translation from action-based obligations to state-based obligations. As temporal logic is a general purpose language, one would also have to define the semantics of policies and process provenance. [Hilty et al., 2005] presents a temporal logic based policy language to specify access control policies supporting obligations. The approach provides a solution to enforce obligations at runtime, but not to verify whether obligations are met in future.

A field related to Care is compliance checking of business process models. The authors of [Namiri and Stojanovic, 2007] use patterns to check compliance of process models with rules at design time of the process. A similar approach is presented in [Goedertier and Vanthienen, 2006]. The authors use an extension of deontic logic [von Wright, 1951, Føllesdal and Hilpinen, 1971] a formalism to express permissions and obligations. They also check the compliance of processes with rules at design time. In [Awad et al., 2010], the authors translate a set of rules specified in linear temporal logic to a Büchi automaton [Büchi, 1966]. The automaton is used to check whether the specified rules contain cyclic dependencies, contradictions or data issues, e.g. the data has to have different states at the same time. In [Ghose and Koliadis, 2007] and [Awad et al., 2009], the authors present solutions to resolve such violations at run-time. These approaches use pattern-based or planning-based strategies to automatically determine a solution. All these compliance checking approaches assume that a model of the business process and the future execution is given.

Many other dimensions may be relevant in different settings for policy definitions. These dimensions are not treated by Care but can be com-

bined with Care. There are policy languages that define policies for actors (cf. [Vedamuthu et al., 2007]) or transactions (cf. [Bauer et al., 2009]) and not for resources. Other languages model usage rights (cf. [Ianella, 2007, Wang et al., 2002]). Some approaches provide methods to enforce policy compliance in closed environments, such as organizations (cf. [Ashley et al., 2003]) or data silos (cf. [Gandon and Sadeh, 2004]). Further languages consider credentials to gain access rights (cf. [Becker and Sewell, 2004, Wang et al., 2002]), relate access control policies to histories of transactions (cf. [Bauer et al., 2009]), support roles and role delegation (cf. [Becker and Sewell, 2004, Hilty et al., 2005]), provide algorithms to identify violated policies (cf. [Accorsi and Wonnemann, 2010]), or support the specification of time intervals in obligations (cf. [Hilty et al., 2005]). Some policy languages specify constraints on other policy languages (cf. [Speiser and Studer, 2010]).

7.6 Summary

The application of obligations in dynamic environments raises various issues, such as to decide whether an instantiated obligation can be fulfilled in the future. To solve this issue, Care first defines the destiny of a data processing. The destiny is defined as the subgraph of the future execution graph that meets all obligations and consists only of valid future processing steps. An instantiated obligation will be unfulfillable if the future has no destiny. The existence of a destiny has to obey the specifications of permit and denial rules, as well as obligation rules.

We have based Care on Papel considering existing means to express obligations (XACML). With Care, we extend Papel by specifying the syntax and semantics of obligations. Care defines the problem to decide whether a destiny exists and provides an according decision procedure using reduction. The decision procedure defines a translation from Care policy rules to colored Petri nets to use existing decision procedures for deciding upon reachability of nodes. By deciding the reachability of nodes we can decide

upon the existence of a destiny.

Through Care we confirm Hypothesis 4 by providing syntax and semantics for obligation rules and by providing a decision procedure to answer the question whether all future obligations can be met.

8 Conclusion

"The Journey is the Destination."
-Confucius

In this thesis we have presented approaches to manage the distributed data processing. Challenges have been addressed in the area of data provenance (past), restriction of data processing (now) and the destiny of data processing (future). For our research we have exploited state transition systems (colored Petri nets) to provide an improved model of the distributed processing of data items. Simulation relations have been specified to define soundness and completeness relations between execution models. We have used semantic methods to provide a well-defined, data-centric monitoring mechanism for distributed environments. A formal syntax of a provenance-aware policy language has been provided and we have exploited model theory and interpretation functions to define the semantics of the language. To base a procedure to decide the existence of a destiny on the well-defined decision problem of reachability in colored Petri nets, we have used reduction.

8.1 Findings and Research Contribution

Through our research we have made various contributions in different areas.

- **Process Modeling:** We have provided a new formalism capturing a novel aspect to the field of process models, DiALog. With DiALog,

we add a means to model the data centric view of data processing to the features of existing process models. The features of the other process models cover the modeling of the control flow as well as the data flow of processes. These models do not focus on modeling the processing of single data items that may span multiple processes. Apart from the data centric approach, we have designed DiALog to model the processing in distributed environments. Also other models allow for modeling distributed environments. The combination with the data centric modeling approach and the use of colored Petri nets leads to DiALog visualizing the distribution of all instances of one specific data item in distributed environments.

- **Qualities of Execution Models:** We have identified different categories of execution models. Each of these models describes a different aspect of the execution of the data processing (logical, technical, etc.). We set the different models in relation to each other and defined soundness and completeness qualities. By the soundness and completeness qualities we can prove whether an instance of one execution model of a certain category can be used to make statements that normally require an instance of another category.

- **Process Auditing:** DiALog provides a new view on the processing of data items that vary from existing process models. This allows for a novel auditing approach that institutes the data centric point of view. Auditing from this point of view servers the auditing of data privacy and data protection issues. We have provided the formal grounding to use other execution models instead of the global model.

- **Process Mining:** Existing means to mine the processing of data are often restricted to single systems or agents. Mining methods for distributed environments concentrate on the data communication between the involved entities but not on other actions performed on the data.

With sticky logging we have added a method for distributed and data-centric process mining to this field.

- **Process Provenance:** Processing histories are part of provenance information consisting either of linear traces or complex temporal structures. In distributed environments, data may be processed in parallel. Linear processing traces, in difference to temporal structures,are not sufficient to express processing histories with parallel paths. On the other hand, temporal structures require complex reasoning. With Papel, we have extended existing approaches to express processing histories by reducing the temporal structure to a graph structure and by defining an after operator to access the temporal aspect of the processing history.

- **Policy Languages (Provenance):** Policy languages provide various means to base policy conditions on metadata. However, either they do not consider processing histories at all or they are general purpose languages. Even if the general purpose languages can be used to express provenance based policy conditions, the existing approaches do not define any semantics to express or access the temporal structure of processing histories. With Papel, we have provided a novel approach that extends other approaches by providing a formal definition of the semantics required to address provenance-based policy conditions considering processing histories.

- **Policy Languages (Obligations):** Existing policy languages foresee the possibility to specify obligations. Some languages also provide means to enforce obligations at runtime. They do not consider the future processing and thus do neither specify an according decision problem nor decision procedure. With Care, we have given a definition of the problem to decide whether the current processing step allows to meet all future obligations or whether it renders obligations unfulfillable. We have provided a procedure to answer the decision

problem by reducing it to a well-defined decision problem in colored Petri nets.

Apart from the single findings, we have introduced integrated means to manage the distributed processing of data. These are DiALog, sticky logging, Papel and Care. All approaches are integrated with each other. Sticky logging uses DiALog as formal model for process provenance and the reconstruction mechanism fulfills the qualities defined by DiALog. Papel uses sticky logging to gather and provide the provenance information required to evaluate policy conditions. Care extents Papel to specify obligations for the future processing and to decide whether those future obligations can be met. We have provided one integrated approach to validate the hypotheses we made in Chapter 1.

8.2 Outlook and Future Work

This thesis is based on requirements derived from current laws and end user agreements. Constantly new ways to use the Internet emerge, such as cloud computing. The demand of data owners and persons concerned to manage the processing of their data also changes. Such changes are initiated by adjustments in the perception of data privacy as caused by the introduction of services, such as Google Street View[1] in Europe. The architecture of the Internet itself changes as well. For instance, by establishing social network sites, e.g. Facebook[2], as a new layer between the transportation and application layer.

These developments change the way data is processed and they affect how the processing is perceived. Even if cloud computing and social network sites are less distributed, data owners and persons concerned may have the impression that they are more 'cloudy'. New risks arise from those develop-

[1] Google Street View, `maps.google.com`, retrieved: Dec. 14th, 2010
[2] Facebook, `www.facebook.com`, retrieved: Dec. 14th, 2010

ments. Our work already constitutes a first step to audit the data processing in other Web environments as cloud computing and social network site.

Another prominent risk by using social network sites is the risk of data items influencing the processing of each other. Data stored in the profile of a social networking site may influences the users credit rating for online shopping. Our approach is also a first step to monitor such influences and to restrict the flow of information in the Internet.

Laws and contracts will adapt as answer to such changes eventually. However, it can not be predicted whether these changes will have an impact on the demands of data owners or the persons concerned to manage the distributed processing of their data.

Bibliography

[104th United States Congress, 1996] 104th United States Congress (1996). Health Insurance Portability and Accountability Act (HIPAA) of 1996 (Public law 104-191).

[Accorsi and Wonnemann, 2010] Accorsi, R. and Wonnemann, C. (2010). Auditing workflow executions against dataflow policies. In *BIS 2010: Proceedings of the 13th International Conference on Business Information Systems*, pages 258–269, New York, NY, USA. ACM.

[Agrawal et al., 2002] Agrawal, R., Kiernan, J., Srikant, R., and Xu, Y. (2002). Hippocratic databases. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 143–154. VLDB Endowment.

[Aldeco-Perez and Moreau, 2008] Aldeco-Perez, R. and Moreau, L. (2008). Provenance-based Auditing of Private Data Use. In *BCS International Academic Research Conference, Visions of Computer Science (In Press)*.

[Aldeco-Pérez and Moreau, 2010] Aldeco-Pérez, R. and Moreau, L. (2010). A provenance-based compliance framework. In Berre, A., Gómez-Pérez, A., Tutschku, K., and Fensel, D., editors, *Future Internet - FIS 2010*, volume 6369 of *Lecture Notes in Computer Science*, pages 128–137. Springer Berlin / Heidelberg.

[Ashley et al., 2003] Ashley, P., Hada, S., Karjoth, G., Powers, C., and Schunter, M. (2003). Enterprise Privacy Authorization Language (EPAL 1.2). Submission to W3C, W3C.

[Awad et al., 2009] Awad, A., Smirnov, S., and Weske, M. (2009). Resolution of compliance violation in business process models: A planning-based approach. In Meersman, R., Dillon, T., and Herrero, P., editors, *On the Move to Meaningful Internet Systems: OTM 2009*, volume 5870 of *Lecture Notes in Computer Science*, pages 6–23. Springer Berlin / Heidelberg.

[Awad et al., 2010] Awad, A., Weidlich, M., and Weske, M. (2010). Consistency checking of compliance rules. In Aalst, W., Mylopoulos, J., Sadeh, N. M., Shaw, M. J., Szyperski, C., Abramowicz, W., and Tolksdorf, R., editors, *Business Information Systems*, volume 47 of *Lecture Notes in Business Information Processing*, pages 106–118. Springer Berlin Heidelberg.

[Barth et al., 2007] Barth, A., Mitchell, J., Datta, A., and Sundaram, S. (2007). Privacy and Utility in Business Processes. In *Proc 2007 Computer Security Foundations Symposium. IEEE.*

[Bauer et al., 2009] Bauer, A., Goré, R., and Tiu, A. (2009). A first-order policy language for history-based transaction monitoring. In Leucker, M. and Morgan, C., editors, *ICTAC*, volume 5684 of *Lecture Notes in Computer Science*, pages 96–111. Springer.

[Becker and Sewell, 2004] Becker, M. Y. and Sewell, P. (2004). Cassandra: Distributed access control policies with tunable expressiveness. In *POLICY '04: Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 159–168, Washington, DC, USA. IEEE Computer Society.

[Bekić, 1984] Bekić, H. (1984). Towards a mathematical theory of processes. In Jones, C., editor, *Programming Languages and Their Definition*, volume 177 of *Lecture Notes in Computer Science*, pages 168–206. Springer Berlin / Heidelberg. 10.1007/BFb0048944.

[Bergstra and Klop, 1984] Bergstra, J. and Klop, J. (1984). Process algebra for synchronous communication. *Information and Control*, 60(1-3):109 – 137.

[Booth et al., 2004] Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D., and Web Services Architecture Working Group (2004). Web services architecture. Technical report, World Wide Web Consortium.

[Bose and Frew, 2005] Bose, R. and Frew, J. (2005). Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37:1–28.

[Bradshaw et al., 2003] Bradshaw, J., Uszok, A., Jeffers, R., Suri, N., Hayes, P., Burstein, M., Acquisti, A., Benyo, B., Breedy, M., Carvalho, M., Diller, D., Johnson, M., Kulkarni, S., Lott, J., Sierhuis, M., and Van Hoof, R. (2003). Representation and reasoning for daml-based policy and domain services in kaos and nomads. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, AAMAS '03, pages 835–842, New York, NY, USA. ACM.

[Byun et al., 2005] Byun, J.-W., Bertino, E., and Li, N. (2005). Purpose based access control of complex data for privacy protection. In *Proceedings of the tenth ACM symposium on Access control models and technologies*, SACMAT '05, pages 102–110, New York, NY, USA. ACM.

[Byun and Li, 2008] Byun, J.-W. and Li, N. (2008). Purpose based access control for privacy protection in relational database systems. *The VLDB Journal*, 17:603–619.

[Büchi, 1966] Büchi, J. R. (1966). Symposium on decision problems: On a decision method in restricted second order arithmetic. In Ernest Nagel, P. S. and Tarski, A., editors, *Logic, Methodology and Philosophy of Science, Proceeding of the 1960 International Congress*, volume 44 of *Studies in Logic and the Foundations of Mathematics*, pages 1 – 11. Elsevier.

[Carroll, 2003] Carroll, J. J. (2003). Signing RDF Graphs. In *The SemanticWeb - International Semantic Web Conference 2003*, pages 369–384, Berlin, Heildelberg. Springer-Verlag.

[Cederquist et al., 2005] Cederquist, J., Conn, R., Dekker, M., Etalle, S., and den Hartog, J. (2005). An Audit Logic for Accountability. In *Policies for Distributed Systems and Networks - Sixth IEEE International Workshop on. June 2005*, pages 34–43.

[Ceri et al., 1989] Ceri, S., Gottlob, G., and Tanca, L. (1989). What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering*, 1:146–166.

[Cheney et al., 2009] Cheney, J., Chiticariu, L., and Tan, W.-C. (2009). Provenance in databases: Why, how, and where. *Found. Trends databases*, 1:379–474.

[Colmerauer and Roussel, 1993] Colmerauer, A. and Roussel, P. (1993). The birth of prolog. *ACM SIGPLAN Notices*, 28:37–52.

[Davidow and Malone, 1992] Davidow, W. H. and Malone, M. S. (1992). *The Virtual Corporation*. HarperCollins, New York.

[Fielding, 2000] Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine.

[Fikes and Nilsson, 1971] Fikes, R. E. and Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189 – 208.

[Føllesdal and Hilpinen, 1971] Føllesdal, D. and Hilpinen, R. (1971). Deontic logic: An introduction. In Hilpinen, R., editor, *Deontic Logic: Introductory and Systematic Readings*, pages 1–35. D. Reidel Publishing Company, Dordrecht.

[Gandon and Sadeh, 2004] Gandon, F. L. and Sadeh, N. M. (2004). Semantic web technologies to reconcile privacy and context awareness. *Journal of Web Semantics*, 1(3):241–260.

[Ghose and Koliadis, 2007] Ghose, A. and Koliadis, G. (2007). Auditing business process compliance. In Krämer, B., Lin, K.-J., and Narasimhan, P., editors, *Service-Oriented Computing – ICSOC 2007*, volume 4749 of *Lecture Notes in Computer Science*, pages 169–180. Springer Berlin / Heidelberg.

[Goedertier and Vanthienen, 2006] Goedertier, S. and Vanthienen, J. (2006). Designing compliant business processes with obligations and permissions. In Eder, J. and Dustdar, S., editors, *Business Process Management Workshops*, volume 4103 of *Lecture Notes in Computer Science*, pages 5–14. Springer Berlin / Heidelberg.

[Gudgin et al., 2007] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., Frystyk Nielsen, H. F., Karmarkar, A., Lafon, Y., and et al. (2007). SOAP Version 1.2 Part 1: Messaging Framework (Second Edition) - W3C Recommendation 27 April 2007. http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[Hallam-Baker and Behlendorf, 1996] Hallam-Baker, P. M. and Behlendorf, B. (1996). Extended Log File Format. http://www.w3.org/TR/WD-logfile-960323.html.

[Hilty et al., 2005] Hilty, M., Basin, D. A., and Pretschner, A. (2005). On obligations. In *ESORICS*, pages 98–117.

[Hinton and Lee, 1994] Hinton, H. M. and Lee, E. S. (1994). The compatibility of policies. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 258–269, New York, NY, USA. ACM.

[Ianella, 2007] Ianella, R. (2007). Open digital rights language (odrl). *Open Content Licensing: Cultivating the Creative Commons*.

[Jensen, 1992] Jensen, K. (1992). *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use - Volume 1*. Springer-Verlag, Berlin.

[Jordan et al., 2007] Jordan, D., Evdemon, J., and the OASIS Web Services Business Process Execution Language (WSBPEL) Technical Committee. (2007). Web Service Business Process Execution Language Version 2.0. OASIS Standard, OASIS.

[Kagal et al., 2003] Kagal, L., Finin, T., and Joshi, A. (2003). A policy language for a pervasive computing environment. In *Policies for Distributed Systems and Networks, IEEE International Workshop on*, pages 63–75, Los Alamitos, CA, USA. IEEE Computer Society.

[Karjoth et al., 2003] Karjoth, G., Schunter, M., and Waidner, M. (2003). Platform for enterprise privacy practices: privacy-enabled management of customer data. In *Proceedings of the 2nd international conference on Privacy enhancing technologies*, PET'02, pages 69–84, Berlin, Heidelberg. Springer-Verlag.

[Klyne et al., 2004] Klyne, G., Carroll, J. J., and et al. (Eds.) (2004). Resource Description Framework (RDF): Concepts and Abstract Syntax. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210.

[Kucera and Mayr, 1999] Kucera, A. and Mayr, R. (1999). Simulation Preorder on Simple Process Algebras. In *ICAL '99: Proceedings of the 26th International Colloquium on Automata, Languages and Programming*, pages 503–512, London, UK. Springer-Verlag.

[Küster et al., 2007] Küster, J., Ryndina, K., and Gall, H. (2007). Generation of business process models for object life cycle compliance. In Alonso, G., Dadam, P., and Rosemann, M., editors, *Business Process*

Management, volume 4714 of *Lecture Notes in Computer Science*, pages 165–181. Springer Berlin / Heidelberg.

[Lansky, 1994] Lansky, A. L. (1994). Action-based planning. In *AL Planning Systems (SIPS'94)*, pages 110–115, Chicago, IL, United States. AAAI.

[Leach et al., 2005] Leach, P., Mealling, M., and Salz, R. (2005). RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace. Internet Engineering Task Force. http://www.ietf.org/rfc/rfc4122.txt.

[Lloyd, 1993] Lloyd, J. W. (1993). *Foundations of Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Lonvick, 2001] Lonvick, C. (2001). RFC 3164: The BSD syslog Protocol. Internet Engineering Task Force. http://www.ietf.org/rfc/rfc3164.txt.

[Ludäscher et al., 2006] Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E. A., Tao, J., and Zhao, Y. (2006). Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065.

[Lupu and Sloman, 1999] Lupu, E. C. and Sloman, M. (1999). Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25:852–869.

[Mayr, 1981] Mayr, E. W. (1981). An algorithm for the general petri net reachability problem. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 238–246, New York, NY, USA. ACM.

[Merriam-Webster, 2004] Merriam-Webster (2004). *Merriam-Webster's Collegiate Dictionary, Eleventh Edition*. Franklin Electronic Publishers.

[Milner, 1980] Milner, R. (1980). A calculus of communicating systems. In *Lecture Notes in Computer Science No. 92*, New York, NY, USA. Springer.

[Milner et al., 1992] Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes, i. *Information and Computation*, 100(1):1 – 40.

[Moreau, 2010] Moreau, L. (2010). The foundations for provenance on the web. *Foundations and Trends in Web Science*, 2(2–3):99–241.

[Moreau et al., 2010] Moreau, L., Clifford, B., Freire, J., Futrelle, J., Gil, Y., Groth, P., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., Plale, B., Simmhan, Y., Stephan, E., and den Bussche, J. V. (2010). The Open Provenance Model core specification (v1.1). *Future Generation Computer Systems*.

[Moreau et al., 2008] Moreau, L., Freire, J., Futrelle, J., Mcgrath, R., Myers, J., and Paulson, P. (2008). The Open Provenance Model: An Overview. *Provenance and Annotation of Data and Processes*, 5272:323–326.

[Moses et al., 2005] Moses, T., Anderson, A., Nadalin, A., Parducci, B., Engovatov, D., Flinn, D., Coyne, E., Damiani, E., Siebenlist, F., Brose, G., Lockhart, H., Kawabe, H., MacLean, J., Merrells, J., Yagen, K., Beznosov, K., Kudo, M., McIntosh, M., Samarati, P., Thiyagarajan, P. V., Humenn, P., Metz, R., Jacobson, R., Hada, S., Vajjhala, S., Proctor, S., Godik, S., Anderson, S., Crocker, S., Damodaran, S., Moses, T., and Welch, V. (2005). eXtensible Access Control Markup Language (XACML) Version 2.0. Oasis standard, OASIS.

[Motik et al., 2009] Motik, B., Patel-Schneider, P. F., Parsia, B., Bock, C., Fokoue, A., Haase, P., Hoekstra, R., Horrocks, I., Ruttenberg, A., Sattler, U., and Smith, M. (2009). OWL 2 Web Ontology Language:

Structural Specification and Functional-Style Syntax, W3C Recommendation 27 October 2009. http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/.

[Naja et al., 2010] Naja, I., Moreau, L., and Rogers, A. (2010). Provenance of decisions in emergency response environments. In McGuinness, D., Michaelis, J., and Moreau, L., editors, *Provenance and Annotation of Data and Processes*, volume 6378 of *Lecture Notes in Computer Science*, pages 221–230. Springer Berlin / Heidelberg.

[Namiri and Stojanovic, 2007] Namiri, K. and Stojanovic, N. (2007). Pattern-based design and validation of business process compliance. In *Proceedings of the 2007 OTM Confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I*, OTM'07, pages 59–76, Berlin, Heidelberg. Springer-Verlag.

[Narayanan and McIlraith, 2002] Narayanan, S. and McIlraith, S. A. (2002). Simulation, Verification and Automated Composition of Web Services. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 77–88, New York, NY, USA. ACM.

[Ni et al., 2010] Ni, Q., Bertino, E., Lobo, J., Brodie, C., Karat, C.-M., Karat, J., and Trombeta, A. (2010). Privacy-aware role-based access control. *ACM Trans. Inf. Syst. Secur.*, 13:24:1–24:31.

[Object Management Group, 2010] Object Management Group (2010). OMG Unified Modeling Language (OMG UML). Technical Report ptc/2010-11-16, Object Management Group.

[Object Management Group, 2011] Object Management Group (2011). Business Process Model and Notation (BPMN). Technical Report formal/2011-01-03, Object Management Group.

[Pednault, 1989] Pednault, E. P. D. (1989). ADL: exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, pages 324–332, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Peterson et al., 2006] Peterson, D., Biron, P. V., Malhotra, A., and Sperberg-McQueen, C. M. E. (2006). XML Schema 1.1 Part 2: Datatypes - W3C. Working Draft 17 February 2006. http://www.w3.org/TR/2006/WD-xmlschema11-2-20060217.

[Peterson, 1981] Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[Petri, 1962] Petri, C. (1962). *Kommunikation mit Automaten*. PhD thesis, Schriften des Rheinisch-Westfälischen Institutes für instrumentelle Mathematik an der Universität Bonn.

[Pnueli, 1981] Pnueli, A. (1981). The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1):45 – 60.

[Pucella and Weissman, 2006] Pucella, R. and Weissman, V. (2006). A formal foundation for odrl. *CoRR*, abs/cs/0601085.

[Ringelstein, 2007] Ringelstein, C. (2007). Protokollierung in service-orientierten Architekturen. *Datenschutz und Datensicherheit*, 31(10/2007):736–739.

[Ringelstein et al., 2007a] Ringelstein, C., Pähler, D., and Schwagereit, F. (2007a). Service-orientierte Architekturen in virtuellen Organisationen. *Datenschutz und Datensicherheit*, 31(9/2007):666–670.

[Ringelstein et al., 2007b] Ringelstein, C., Schwagereit, F., and Pa"hler, D. (2007b). Opportunities and Risks for Privacy in Service-oriented Architectures. In *Proceedings of the 5th International Workshop for Techni-*

cal, Economic and Legal Aspects of Business Models for Virtual Goods incorporating the 3rd International ODRL Workshop, pages 197–214, Koblenz. Nova Science Publishers.

[Ringelstein and Staab, 2007] Ringelstein, C. and Staab, S. (2007). Logging in Distributed Workflows. In *Proceedings of the Workshop on Privacy Enforcement and Accountability with Semantics*, Busan, South-Korea.

[Ringelstein and Staab, 2009] Ringelstein, C. and Staab, S. (2009). Dialog: Distributed auditing logs. In *Web Services, IEEE International Conference on*, pages 429–436, Los Angeles, CA, USA. IEEE Computer Society.

[Ringelstein and Staab, 2010a] Ringelstein, C. and Staab, S. (2010a). DiALog: A Distributed Model for Capturing Provenance and Auditing Information. *International Journal of Web Services Research (JWSR)*, 7(2):1–20.

[Ringelstein and Staab, 2010b] Ringelstein, C. and Staab, S. (2010b). PAPEL: A Language and Model for Provenance-Aware Policy Definition and Execution. In *BPM 2010 - International Conference on Business Process Management*, volume 6336, pages 195–210. Lecture Notes in Computer Science.

[Ringelstein and Staab, 2011] Ringelstein, C. and Staab, S. (2011). Papel: Provenance-Aware Policy Definition and Execution. *IEEE Internet Computing*, 15(1):49–58.

[Rivest et al., 1978] Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21:120–126.

[Russell and Norvig, 2003] Russell, S. J. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition.

[Sandhu et al., 1996] Sandhu, R., Coyne, E., Feinstein, H., and Youman, C. (1996). Role-based access control models. *IEEE Computer*, 29:38–47.

[Simmhan et al., 2005] Simmhan, Y. L., Plale, B., and Gannon, D. (2005). A survey of data provenance in e-science. *SIGMOD Rec.*, 34:31–36.

[Sizov, 2007] Sizov, S. (2007). What Makes You Think That? The Semantic Web's Proof Layer. *IEEE Intelligent Systems*, 22(6):94–99.

[Smullyan, 1968] Smullyan, R. M. (1968). *First-order Logic*. Springer, Berlin.

[Speiser and Studer, 2010] Speiser, S. and Studer, R. (2010). A self-policing policy language. In Patel-Schneider, P., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J., Horrocks, I., and Glimm, B., editors, *The Semantic Web – ISWC 2010*, volume 6496 of *Lecture Notes in Computer Science*, pages 730–746. Springer Berlin / Heidelberg.

[The European Parliament and the Council of the European Union, 1995] The European Parliament and the Council of the European Union (1995). Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data. In *Official Journal L 281 of 23/11/1995*, pages 31–50.

[Tschantz et al., 2011] Tschantz, M. C., Datta, A., and Wing, J. M. (2011). On the semantics of purpose requirements in privacy policies. *CoRR*, abs/1102.4326.

[van der Aalst, 1998] van der Aalst, W. (1998). The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66.

[van der Aalst et al., 2004] van der Aalst, W., Weijters, T., and Maruster, L. (2004). Workflow Mining: Discovering Process Models from Event Logs.

IEEE Transactions on Knowledge and Data Engineering, 16(9):1128–1142.

[van der Aalst et al., 2008] van der Aalst, W. M. P., Dumas, M., Ouyang, C., Rozinat, A., and Verbeek, E. (2008). Conformance Checking of Service Behavior. *ACM Trans. Interet Technol.*, 8(3):1–30.

[Vedamuthu et al., 2007] Vedamuthu, A. S., Orchard, D., Hirsch, F., Hondo, M., Yendluri, P., Boubez, T., Ümit Yalçinalp, and et al. (2007). Web Services Policy 1.5 - Framework. W3C Recommendation, W3C.

[von Wright, 1951] von Wright, G. H. (1951). Deontic logic. *Mind*, 60(237):pp. 1–15.

[Wainer et al., 2003] Wainer, J., Barthelmess, P., and Kumar, A. (2003). W-RBAC - A Workflow Security Model Incorporating Controlled Overriding of Constraints. *International Journal of Cooperative Information Systems (IJCIS)*, 12:455–485.

[Wang et al., 2002] Wang, X., Lao, G., DeMartini, T., Reddy, H., Nguyen, M., and Valenzuela, E. (2002). Xrml – extensible rights markup language. In *XMLSEC '02: Proceedings of the 2002 ACM workshop on XML security*, pages 71–79, New York, NY, USA. ACM.

[Weibel et al., 1998] Weibel, S., Kunze, J., Lagoze, C., and Wolf, M. (1998). Rfc 2413: Dublin core metadata for resource discovery. http://www.ietf.org/rfc/rfc2413.txt.

[Weitzner et al., 2008] Weitzner, D. J., Abelson, H., Berners-Lee, T., Feigenbaum, J., Hendler, J. A., and Sussman, G. J. (2008). Information Accountability. *Communications of the ACM*, 51(6):82–87.

[Wenning et al., 2006] Wenning, R., Schunter, M., and et al. (2006). The Platform for Privacy Preferences 1.1 (P3P1.1) Specification. http://www.w3.org/TR/2006/NOTE-P3P11-20061113/.

[White, 1976] White, J. E. (1976). RFC 707: A High-Level Framework for Network-Based Resource Sharing. . http://www.ietf.org/rfc/rfc707.txt.

List of Figures

List of Tables

Curriculum Vitae:

Christoph Ringelstein is born in Koblenz, Germany. He studied computer science with business information systems at the University of Koblenz-Landau from 2000 to 2005. In 2005, he joined the Institute of Web Science and Technologies of Prof. Steffen Staab as researcher. He has been a visiting researcher to the University of Illinois at Urbana-Champaign, USA, in 2008.

Publications:

Journals and Papers

Ringelstein, Christoph; Staab, Steffen (2011): Papel: Provenance-Aware Policy Definition and Execution. In: IEEE Internet Computing. Bd. 15. Nr. 1. S. 49-58.

Ringelstein, Christoph; Staab, Steffen (2010): DiALog: A Distributed Model for Capturing Provenance and Auditing Information. In: International Journal of Web Services Research (JWSR). Bd. 7. Nr. 2. S. 1--20.

Ringelstein, Christoph; Staab, Steffen (2010): PAPEL: A Language and Model for Provenance-Aware Policy Definition and Execution. In: BPM 2010 - International Conference on Business Process Management. Lecture Notes in Computer Science. Bd. 6336. S. 195-210.

Ringelstein, Christoph; Staab, Steffen (2009): DIALOG: Distributed Auditing Logs. In: ICWS-2009 - 7th IEEE Int. Conference on Web Services. Los Angeles, CA, USA.

Ringelstein, Christoph; Staab, Steffen (2007): Logging in Distributed Workflows. In: Proceedings of the Workshop on Privacy Enforcement and Accountability with Semantics. Busan, South-Korea.

Ringelstein, Christoph; Schwagereit, Felix; Pähler, Daniel (2007): Opportunities and Risks for Privacy in Service-oriented Architectures. In: Proceedings of the 5th International Workshop for Technical, Economic and Legal Aspects of Business Models for Virtual Goods incorporating the 3rd International ODRL Workshop. Koblenz.

Bookchapters

Stein, Sebastian; Stamber, Christian; Hering, Thomas; Donath, Steffi; Ringelstein, Christoph (2008): Service Enabling. In: Weske, Mathias; Tröger, Peter; Kuropka, Dominik; Staab, Steffen: Semantic Service Provisioning. Springer-Verlag.

Ringelstein, Christoph; Franz, Thomas; Staab, Steffen (2007): The Process of Semantic Annotation of Web Services. In: Cardoso, J.: Semantic Web Services - Theory, Tools, and Applications. Idea Publishing Group, USA.

Tech Reports

Ringelstein, Christoph; Staab, Steffen (2010): PAPEL: Syntax and Semantics for Provenance-Aware Policy Definition. Institut WeST, Universität Koblenz-Landau. Nr. 04/2010. Arbeitsberichte aus dem Fachbereich Informatik.

Staab, Steffen; Grimm, Rüdiger; Bizer, Johann; Ringelstein, Christoph; Schwagereit, Felix; Pähler, Daniel; Meissner, Sebastian; Rost, Martin; Schallaböck, Jan (2007): SOAinVO - Chancen und Risiken von Service-orientierten Architekturen in Virtuellen Organisationen.

Artikels

Ringelstein, Christoph (2007): Protokollierung in service-orientierten Architekturen. In: Datenschutz und Datensicherheit. Bd. 31. Nr. 10.

Pähler, Daniel; Ringelstein, Christoph; Schwagereit, Felix (2007): Service-orientierte Architekturen in virtuellen Organisationen. In: Datenschutz und Datensicherheit. Bd. 31. Nr. 9/2007. S. 666-670.

Posters and Demos

Franz, Thomas; Saathoff, Carsten; Görlitz, Olaf; Ringelstein, Christoph; Staab, Steffen (2006): SEA: A Lightweight and Extensible Semantic Exchange Architecture. In: Proceedings of the 2nd Workshop on Innovations in Web Infrastructure. 15th International World Wide Web Conference (Edinburgh, Scotland).

Franz, Thomas; Saathoff, Carsten; Görlitz, Olaf; Ringelstein, Christoph; Staab, Steffen (2006): SEA: Introducing the Semantic Exchange Architecture. In: Poster & Demo Session, ESWC 2006.

Ben Amor, Heni; Murray, Jan; Obst, Oliver; Ringelstein, Christoph (2003): RoboLog Koblenz 2003 -- Team Description. In: Bonarini, Andrea; Browning, Brett; Polani, Daniel; Yoshida, Kazuo: Proceedings of the RoboCup 2003. Academic Press.

Obst, Oliver; Ringelstein, Christoph (2002): RoboLog Koblenz 2002 -- Visualization Description. In: Kaminka, Gal A.; Lima, Pedro U.; Rojas, Raul: RoboCup 2002: Robot Soccer World Cup VI. Fukuoka, Japan.