MODELLING ARCHITECTURES OF FEDERATED IDENTITY MANAGEMENT SYSTEMS

Thesis submitted for the degree of

Doctor of Philosophy

at the University of Leicester

by

Hyder Ali Nizamani

Department of Computer Science

University of Leicester

2011

Declaration

The content of this submission was undertaken in the Department of Computer Science, University of Leicester, and supervised by Dr. Emilio Tuosto during the period of registration. I hereby declare that the materials of this submission have not previously been published for a degree or diploma at any other university or institute. All the materials submitted is the result of my own research except as cited in the references.

Preliminary versions of some results related to those presented in this submission have been published in the following papers:

- Qurat ul Ain Nizamani and Hyder A. Nizamani.*Analysis of a Federated Identity Management Protocol in SOC.* In *Proceedings of the 3rd Young Researchers Workshop on Service Oriented Computing (YRSOC'08), 2008.*

- Hyder A. Nizamani and Emilio Tuosto. *Federated Identity Management System Patterns as Architectural Reconfigurations*. In Electronic Communications of the EASST, Volume 31, 2010.

Abstract

Today's dynamic and scalable collaborative systems demand not only to deal with functional but also some non-functional (e.g., security) requirements. For a secure inter-organisational collaboration scenario, *Federated Identity Management* systems (FIMs) provide a suitable mechanism to deal with access control. FIMs enable users of an organisation to access resources (or services) of the other trusted organisations in a secure and seamless way. More precisely, FIMs allow cross-domain user authentication to enable access control across organisations under the concept known as *Circle of Trust* (CoT). Patterns of FIMs emerged as recurring CoT scenarios due to the fact that each of these patterns has different security requirements. More importantly, organisations may join up or leave the CoT during the development life-cycle. Such a change in a FIM system may have an impact on its security requirements. Therefore, it is important to formally describe architectural and reconfiguration aspects of FIMs by considering their patterns.

To this purpose, we propose

- two UML models for FIMs where one model uses the standard UML notations to describe architectural aspects of FIMs while the other uses the UML profile in [33] to describe those aspects of FIMs together with their reconfigurations
- a formal model for FIMs in ADR (Architectural Design Rewriting) to characterise their patterns by describing an architectural style together with style-preserving reconfigurations.

We also study the adequacy of UML to describe architectural aspects of systems and compare it with ADR. Our comparison develops through the modelling of architectural and reconfiguration aspects of FIMs. In ADR, these aspects of FIMs are suitably represented through style-consistent (graphical) designs in terms of ADR productions. On the other hand, UML has limitations in expressing constraints over complex associations; also, UML seems to provide unsatisfactory support for presenting architectural styles in a general way. Overall, our investigation shows that UML has some drawbacks due to the complexity of diagrams, their proliferation, and the lack of a precise semantics that consistently relates them. ADR gives precise and simpler specifications for architectural design.

Acknowledgements

First and foremost I would like to thank my supervisor Dr. Emilio Tuosto, it was not possible to complete this dissertation without his help and support. He was always there to provide me best advice and guidance whenever I needed it. There were quite a few moments when I lost track of my research but he managed to put me back on the right direction. I would like to express my gratitude to my co-supervisor Dr. Stephan Reiff-Marganiec whose timely and valuable suggestions were very helpful during my Ph.D. studies. I would also like to thank Dr. Fer-Jan de Vries and Professor Thomas Erlebach for the help and support they provided during my stay at the University of Leicester.

I am lucky to have friends who always extended their help whenever I needed it. Specially I am indebted to Fida Hussain Chandio, Amar Jalil Metlo, Aslam Shahani, Abdul Fattah Soomro, Niaz Arijo, Imtiaz Korejo, Tariq Umrani, and Muzammil Chaudhary. I am also grateful to Intazar Lashari for being extremely helpful in dealing with official matters at University of Sindh on my behalf.

And most importantly I thank my wife Qurat ul Ain for giving me all the love and care and always standing by me through thick and thin and our son Talha Nizamani for being a source of love and inspiration. Talha's innocent acts and loving smile always cheered me up after long hours of studies at the university. Thanks to my brothers Junaid Nizamani, Kamran Nizamani and Abdul Rahman Nizamani for taking care of my loving Baba and Amma so that I could do my research with peace of mind.

I dedicate this work to my parents who made me realise the importance of quality education and provided the opportunities to achieve the same. I also dedicate this work to my teachers who endowed me with knowledge and skills that enabled me to reach this level of qualification.

Glossary

AAI Authentication and Authorisation Infrastructure.

ADL Architectural Description Language.

ADR Architectural Design Rewriting.

AF Arbitrary Federation.

BF Bilateral Federation.

CoT Circle of Trust.

DG Dimension Graph.

DTMS Dynamic Trust Management System.

F Federation.

FIM Federated Identity Management.

FIMs Federated Identity Management systems.

GC Global Constraint.

GG4SA Graph Grammars for SAs.

GT4SA Graph Transformation for SAs.

HR4SA Hyperedge Replacement for SAs.

IDP Identity Provider.

LHS Left Hand Side.

MIF Multiple Identity Providers Federation.

MSF Multiple Service Providers Federation.

OCL Object Constraint Language.

PG Pattern Graph.

RHS Right Hand Side.

RIC Regional Information Centre.

SA Software Architecture.

SAML Security Assertion Markup Language.

SSO Single Sign-on.

SOA Service Oriented Architecture.

SOC Service Oriented Computing.

SP Service Provider.

UML Unified Modelling Language.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Today's dynamic and scalable collaborative systems (e.g., Cloud, Web 2.0, etc.) demand not only to deal with functional but also with non-functional (e.g., security) requirements. For instance, access to protected resources in an organisation is typically governed by an access control system.

Organisations need some form of authentication by using usernames/passwords, Kerberos [1] tokens or X.509 [62] certificates before allowing users to access security enabled resources. The users provide verified identities in order to access these resources shared by the trusted organisation in inter-organisation collaboration scenarios. *Federated Identity Management* systems (FIMs) provide a suitable mechanism to deal with access control for such scenarios. More precisely, a FIM system allows subscribers from different organisations to use their internal identification information in order to obtain access to the network of all enterprises in the group called the *Circle of Trust* (CoT) [30, 79]. Consequently, FIMs are becoming more and more appealing as they enable organisations to smoothly join up and share distributed resources (e.g., services, applications, etc.) in a secure way.

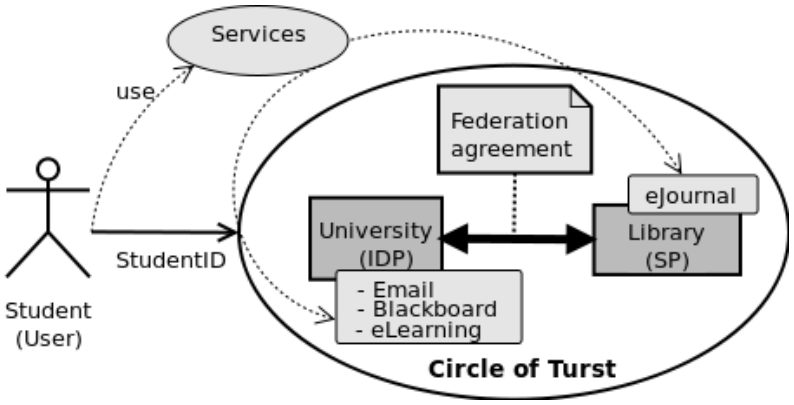


Figure 1.1: A circle of trust in a FIM system

The notion of CoT in FIMs allows a group of organisations to be federated by establishing a trusted relation and agreeing on certain rules of the cooperation. There are two distinguished kinds of organisations in a FIM system, one called *identity provider* (IDP) and the other called *service provider* (SP). Where an IDP and SP are parts of the same CoT, the user registered with the IDP can access the resources/services offered by the SP. For instance, a university may federate with a digital library to grant students access to some electronic publications. Figure 1.1 shows such a FIM scenario where the university (i.e., IDP) may provide students' authentication related information (i.e., security token) to the library (i.e., SP) on request to access the publications. In this case, the IDP is responsible to ensure the security of the tokens issued to the SP through the requester by digitally signing them to provide token authentication and integrity [42].

Today, various FIM protocols (e.g., in [82]) and standard specifications (e.g., in [5, 43, 6]) exist and we discuss a few of them which are widely referred in the literature relevant to FIM [66, 76, 82, 42, 79, 87, 44]. For instance, the *Security Assertion Markup Language* (SAML) [5] is a de-facto standard for exchanging security related information between the collaborating organisations. SAML is an XML-based framework developed by the *OASIS* (Organization for the Advancement of Structured

Information Standards) *Security Services Technical Committee*[1]. SAML is an open FIM standard [82]. It allows organisations to communicate security related information for the purpose of enabling cross-domain user authentication. To this purpose, SAML conveys such an information about the users in the from of *assertions* [87], which are the statements issued by the IDP while the users make requests to access the resources. Further, SAML can be considered as a flexible and extensible protocol specification which can be used and (if necessary) customized by the other FIM standards [20]. In this connection, various standards bodies and organisations have developed specific SAML-based FIM protocols, software packages, and standards. For instance, the *Liberty Alliance* [22], the *Internet2 Shibboleth* project[2], and the OASIS *Web Services Security (WS-Security) Technical Committee*[3] have respectively adopted SAML to develop a FIM protocol, software product, and standard-based specification.

The Liberty Alliance is an industrial consortium of more than 150 companies. It has developed specifications and guidelines for governments, businesses, and individuals to establish legally binding CoTs [29]. Such specifications and guidelines have been adopted in various domains (e.g., eGovernment, Finance, Healthcare, Education, Telecom, etc.)[4]. One of the Liberty Alliance's specifications called the *Liberty Identity Federation Framework* (ID-FF) [43] provides an approach to implement a FIM system. In particular, ID-FF describes a federation protocol which can be used to enable *Single Sign-on* (SSO) namely, allowing users to authenticate only at their IDP without re-authenticating themselves to access services offered by the SPs [44].

Internet2 is an advanced networking consortium comprising a large research and education community. It has developed a standard-based open-source software package known as *Shibboleth*, which provides an effective FIM solution [76]. Specifi-

[1] http://www.oasis-open.org/committees/security/
[2] http://shibboleth.internet2.edu/
[3] http://www.oasis-open.org/committees/wss/
[4] The relevant case studies can be found at http://www.projectliberty.org/liberty/adoption

cally, Shibboleth offers multi-protocol support to implement FIM systems. For instance, currently Shibboleth supports standard SAML-based protocols. In addition to this, Shibboleth supports a FIM protocol which is based on an other open standard called *OpenID* [16]. Such an approach allows a common internet user to federate their OpenID account (e.g., at VeriSign's Personal Identity Provider) across several OpenID-enabled web-sites (e.g., Google, Yahoo, Flickr, PayPal, MySpace, VeriSign, etc). Also, support for an additional identity management system (Microsoft's InfoCard [45]) is also being considered in Shibboleth's future release [21].

OASIS is an international consortium which produces open standards for *Service-oriented architectures* (SOA), Web services, security, Cloud computing, eGovernment, and several other areas. The WS-Federation [6] specification was initially proposed by IBM and Microsoft to support the development of secure web-services in a larger context [82]. WS-Federation was adopted by the OASIS Web Services Security Technical Committee for its standardisation efforts. The main goal of such a specification is to provide a mechanism to simplify the development of security related services (e.g., authentication and authorisation) to enable cross-domain communication and management of web-services within the federation [59].

Recently, a few FIM patterns have been proposed in [66]. Note that term "FIM pattern" is used in this dissertation to represent a particular constellation of trust relationships between IDP and SP organisations in a FIM system (cf. Section 2.2 for details). More precisely, the FIM patterns in [66] can be differentiated according to the configuration of IDPs and SPs participating in a FIM system. We remark that this may have an impact on the security requirements for the FIMs. In particular, FIM patterns can be differentiated according to the security threats they are exposed to. These threats (cf. Section 2.2 for details) include

- unauthorised access,

- compromise of users' privacy,
- disclosure of users' credentials,
- consolidate usage statistics,
- unauthorised delegated authentication, and
- accumulating data on individual users.

Typically, an IDP in a FIM system uses certain procedures to register its users (i.e., by creating their credentials) and map them to their designated roles in the organisation. For instance, consider an IDP that uses inadequate procedures to accomplish these tasks; a possible threat is that the IDP may grant unauthorised access to users.

In a FIM system, a provider (i.e., an IDP and an SP) can typically acquire and keep information of users' activities. For instance, an IDP knows how often a user communicates with an SP. This is regarded as a threat to users' privacy; in fact, such kind of information is exploited by the provider in order to acquire knowledge on users' behaviour. For instance, an IDP may be interested in monitoring usage data of users and/or the history of their interactions with SPs (sometimes providers are offered incentives, like financial benefits, for sharing the usage data of the users with third parties). Similarly, an SP in a FIM system might disclose user credentials or a portion of those credentials (e.g., selected user attributes) received from the IDP.

In FIMs, a user can be registered with multiple IDPs. In this case, the user may (frequently) change his/her IDP and, as a result, a threat may happen when some or all of the IDPs may collude and consolidate his/her usage statistics (i.e., accesses to the SPs) that are distributed across the providers. In FIMs, multiple SPs can be federated and they might allow direct access to each other via delegated user authentication. For instance, an SP may provide a complex service that combines (or orchestrates) other services of the federated SPs. While allowing users to access such services, a threat

may happen when the SPs allow delegation of authentication information without obtaining consent of the user and his/her IDP.

In FIMs, IDPs typically provide the necessary information on users' credentials to the SPs for authentication purposes. For instance, in the FIM system shown in Figure 1.1 the university (i.e., IDP) may provide selected attributes of student (i.e., StudentID, TypeOfStudent, etc.) to the library (i.e., SP) so as to allow access to the articles. In a FIM scenario where multiple SPs are federated, another threat may happen when they may collude and gather such information into a comprehensive user file (e.g., a centralised database/file containing details about the user credentials). As a result, those SPs may correlate their respective portions of the authentication information in order to accumulate data on individual users.

Nowadays, FIM implementations may be found in various domains (e.g., finance, education, healthcare, eGovernment, etc.) and such systems are considered relatively static. However, a dynamic approach may be realised that enables organisations leaving or joining the CoTs to take some economic benefits in today's emerging dynamic and scalable systems (e.g., Clouds). In this way, one can also take advantage of *service-oriented computing*. For instance, a FIM system can be used to deal with access control in a cloud (e.g., in [50]). Architectural modelling of these systems becomes necessary, because, in current distributed systems, the possibility to tackle (dynamic) changes is paramount. We remark that this reflects at the architectural level and requires what is known as architectural *reconfiguration*. Architectural styles may provide a suitable mechanism for guiding the reconfigurations in a way that any change in the architecture does not violate the style. Therefore, it is important to formally represent architectural aspects of FIMs by characterising their configurations (and reconfigurations) according to the FIM patterns.

During design time, such an architectural description of FIMs can potentially be exploited to deal with the risks associated with their configurations. For instance, de-

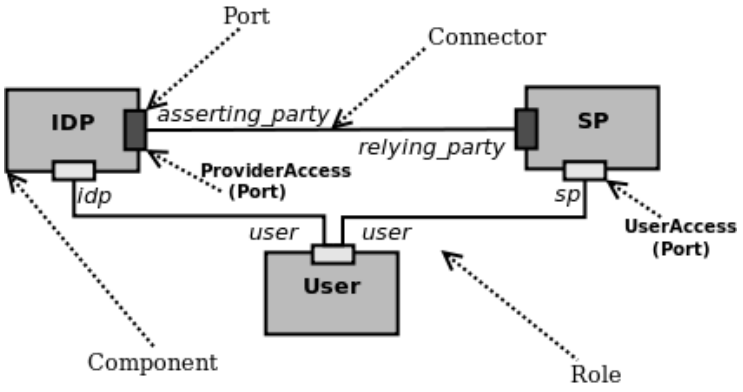


Figure 1.2: Architecture of a software system

signers (e.g., architects, developers, etc.) may enforce the required security mechanism beforehand while they generate such configurations (or possibly reconfigure them).

The architecture of a software system basically consists of the structure of components and the way they are interconnected. For instance, Figure 1.2 shows the architecture of a software system that describes how components are connected to each other by attaching their respective interaction elements (i.e., ports). Often, the notion of architectural style [86] of a system is used to specify how its elements should be "legally" interconnected in a given configuration. An important aspect to consider when modelling architectural aspects is that, typically, the architectural style of a system should be preserved. For instance, the essential architectural property of FIMs specifies that at least one IDP and one SP should be federated in order to form a legal CoT. In this way, each of the FIM patterns may have distinct architectural properties (e.g., in terms of the number and kind of providers involved in a given configuration). Due to the security critical nature of FIMs, therefore, it is desirable that the modelling approach should provide a suitable mechanism to generate FIM configurations with respect to their underlying FIM patterns.

Architectural Description Languages (ADLs) formally represent architectures of software systems. Unlike programming languages that target machine executables, architectural descriptions in an ADL target a kind of analysis so as to reason about a

given configuration with respect to certain architectural properties (i.e., conformance, performance, deadlock, etc..). Specifically, ADLs allow designers to consider the high-level structure of the overall software systems rather than implementation details of a specific element (e.g., a component or a source module) [72]. In this way, ADLs promote architectural development at the desired level of abstraction where irrelevant details are not considered. For instance, one may abstract away the implementation level details of an IDP in a FIM system that may be considered later.

1.2 Problem Statement

In this dissertation, the research question we address is:

> *How can architectural description of FIM systems be exploited to model and analyse threats associated with structural configurations?*

Recently, a few structural patterns [66] of FIMs have been informally considered in relation to security and trust requirements of FIM systems. Such patterns will be detailed in Section 2.2 and are summarised below:

- The *Bilateral Federation* (**BF**) consisting a single IDP and a single SP.
- The *Multiple IDPs Federation* (**MIF**) consisting multiple IDPs and a single SP.
- The *Multiple SPs Federation* (**MSF**) consisting a single IDP and multiple SPs.
- The *Arbitrary Federation* (**AF**) consisting multiple IDPs and multiple SPs.

Those patterns are based on *direct* trust relations, that is relationships between the organisations participating in a CoT that do not involve third parties. Moreover, they can be differentiated in terms of the security threats that they are exposed to. In Table 1.1 we summarise the relationships between the threats and FIM patterns. In this

Threats	FIM patterns			
	BF	**MIF**	**MSF**	**AF**
Unauthorized access	✓	✓	✓	✓
Compromise users' privacy	✓	✓	✓	✓
Disclosure of credentials	✓	✓	✓	✓
Consolidate usage statistics	-	✓	-	✓
Unauthorised delegation	-	-	✓	✓
Accumulate identity information	-	-	✓	✓

Table 1.1: Threats to FIM patterns

table, a tick (symbol ✓) represents that the pattern is exposed to the threat while a dash (symbol -) represents that the pattern is not exposed to the threat. Notably, it is crucial to characterise configurations of these patterns which may allow designers to consider the known threats to the configurations and the mechanism (e.g., executing the required security and trust policies) to deal with them.

ADLs formally describe architectures of software systems. Instead of considering just general aspects of ADLs, we focus on particular architectural issues arising in the context of FIMs. Architectural aspects of FIMs impact on the security threats they are exposed to; therefore, it is crucial to precisely describe architectural views of FIMs. We contend that the architectural modelling of FIMs offers a relatively complete range of challenging issues (described below) that are common to many other realistic scenarios.

- **Architectural styles:** For FIMs, styles are paramount as they allow the designer to precisely characterise their configurations so as to control their security threats.

- **Style checking:** To fully exploit the feature of architectural styles, the designer should be supported with features allowing him/her to validate a FIM configuration against its style. This corresponds to being able to classify FIM configurations as well as decide which productions (or reconfigurations) to apply.

- **Refinement:** Architectural refinement is a convenient way to handle complex systems. The intuition is that the designer considers high-level descriptions of architectural elements and *refines* abstract architectures into actual configurations in several steps. In FIM, refinement allows concepts like CoT, Federation, etc., to be clearly separated and related with each other.

- **Reconfigurations:** Architectures of software systems may change during the development life-cycle. Typically, dynamic software systems allow such changes to be described by different kinds of reconfigurations. For FIM, style preserving reconfigurations are particularly relevant.

- **Generating FIM patterns:** For FIMs, it is paramount to have a precise mechanism for generating configurations when specific FIM patterns are chosen. In fact, one may need to enforce specific security measures for FIM configurations according to the underlying pattern. When such mechanisms are available, specific guidelines about the composition of architectural configurations can be given in order to respect a given style. This allows designers to consider the required security mechanism beforehand for FIM configurations.

- **Identifying FIM patterns:** A crucial aspect of FIMs is their high degree of dynamicity. More precisely, CoTs can add/remove new IDPs or SPs at run time and this would require a reconfiguration of the system, but also at design time it might be necessary to reconfigure architectural views of a FIM system. In the latter case, one needs to identify the actual pattern of the system of interest in order to (*a*) assess its security threats and (*b*) decide when a given reconfiguration could be applied. For instance, adding an SP to a system of one FIM pattern may reconfigure it into another FIM pattern. Notice that such a change in the configuration has a direct effect on the security requirements. Hence, pattern

identification techniques are paramount for FIM and calls for suitable mechanisms to support designers.

1.3 Our Approach

We mainly focus on structural and reconfiguration aspects of FIMs as those aspects impact on the security requirements. We use

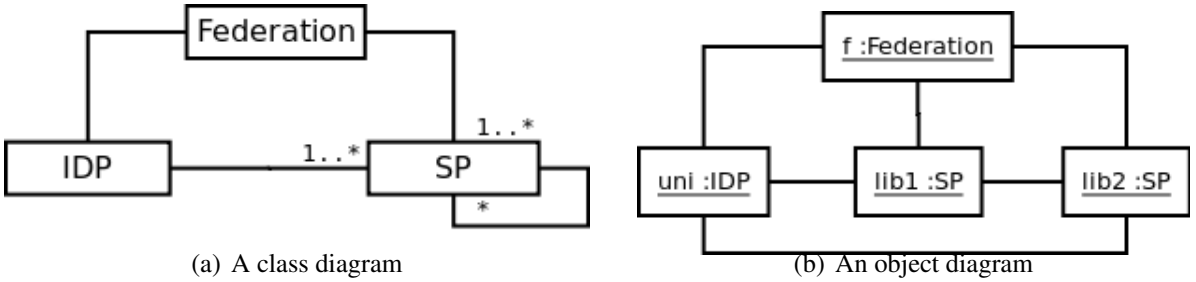
- UML [80] (and the profile introduced in [33]) as an ADL and
- ADR (after Architectural Design Rewriting) [37, 38] as an ad-hoc ADL

to describe those aspects of FIMs.

Notice that we do not model behavioural aspects of FIM systems. For instance, to support the execution of FIM systems in ADR we may consider the FIM protocol in [89] to formally represent the interactions between providers. Such aspects of FIM systems can be considered in future work. In fact, a particular FIM implementation model (e.g., SAML [5], Liberty Alliance [22], WS-Federation [6], etc.) influences the possible reconfigurations to be represented.

Using UML as an ADL to Model FIMs UML is considered a de-facto standard to model software systems. UML has been promoted as an ADL with the introduction of structured classifiers (e.g., classes, components) and ports concepts. We use standard UML notations to describe architectural aspects of FIMs. The main advantage of using such an approach is that UML designers can conveniently manipulate the diagrams.

In UML, we use classes to represent types of architectural elements (i.e., components and ports). To represent configurations, we use two different kinds of instance level diagrams, namely object diagrams and composite structure diagrams (instance level). Object diagrams model a flat view of the system and they suitably represent



(a) A class diagram (b) An object diagram

Figure 1.3: The structural diagrams in UML

configurations of simple scenarios. Instead, composite structure diagrams impose a hierarchy over complex configurations. Also, these diagrams simplify architectural configurations by attaching ports of the components via UML connectors.

To model a style, class diagrams are used to represent types of components and their possible relationships. Complex relations among components can be modelled by using multiplicities and OCL constraints. For instance, Figure 1.3(b) shows an object diagram which describes such a configuration with respect to its class diagram given in Figure 1.3(a). Consider the multiplicities over the associations among the classes in Figure 1.3(a); a `Federation` can be connected to one `IDP` and one or more `SP`s, an `IDP` is connected to one or more `SP`s, and an `SP` is connected to zero or more `SP`s. In addition to these multiplicities, the class diagram in Figure 1.3 may require a few OCL constraints, for instance, a constraint which specifies that an `IDP` should be attached to all `SP`s attached to a `Federation`. The OCL constraint

context IDP
inv: *self*.sps.size()>=1 `and` *self*.sps.size()=*self*.fed.sps.size()

describes such condition and a similar constraint is also needed for `SP`s.

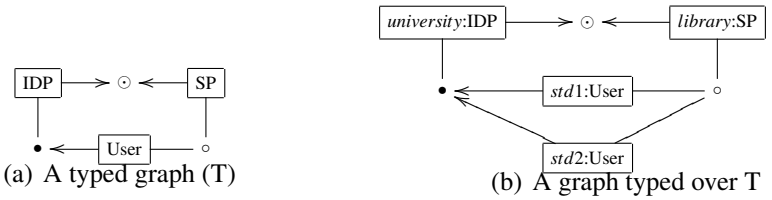
Since standard UML does not support abstract architectural components, we do not address refinement for such components in FIMs. Similarly, UML does not provide any mechanism which can be used to describe reconfigurations. Therefore, we apply changes directly in the configurations.

To generate configurations in UML, one has to use rules of thumb to create object diagrams and composite structure diagrams (instance level). Notice that UML lacks certain desired features (i.e., architectural refinement, reconfiguration, and pattern generation) that may be required by designers during the design of software systems.

To overcome the limitation of standard UML, we use a profile. The profile is based on the recent proposal in [33] inspired by the formal framework introduced in [38, 37] and specifically designed to support architectural modelling. To support architectural refinement, the profile allows one to describe abstract architectural components whose associated productions describe their configurations via composite structure diagrams. Also, such productions may enable one to effectively generate configurations of simple scenarios. However, this approach also suffers from some limitations for generating complex configurations. In order to describe the changes in software architectures, the profile uses a graph transformation approach in UML to model reconfigurations.

Using ADR as an ad-hoc ADL to Model FIMs ADR [37, 38] is a graphical and formal approach to describe style-preserving reconfigurable architectures. It allows us to appropriately address the complete range of issues related to the modelling of architectural and reconfiguration aspects of FIMs.

In ADR, hyperedges model types of components, non-terminals hyperedges model abstract (or refineable) components, and terminal hyperedges model basic (or non-refineable) components. A node models interaction between the components and tentacles leaving hyperedges and joining a common node represents how the components may interact. A typed hypergraph (i.e., a graph typed over a type graph) represents a configuration where components are composed together via their associated ports. For instance, Figure 1.4(b) shows a graph typed over the type graph in Figure 1.4(a). Type hypergraphs suitably represent architectural styles, i.e., the type of architectural elements (components and ports) and a set of productions (or design rules) define legal



(a) A typed graph (T) (b) A graph typed over T

Figure 1.4: A type graph and a typed graph

connection between those elements.

To generate valid configurations of interest (e.g., FIM patterns), ADR's style-based refinement approach is used that allows abstract components to be refined by applying the corresponding productions. As a result, configurations in ADR have an associated term-like representation that describes how such configurations are built. Also, those algebraic formulations provide witnesses of their construction. In this way, productions can be given an algebraic formulation where a term describes a particular style-proof.

In ADR, rewrite rules define style-preserving reconfigurations. Such reconfigurations are operated at the level of style-proofs by exploiting term rewriting over style-proof terms. We define the basic reconfiguration rules (without variables) in ADR that add a single component of each type (e.g., IDP or SP). Also, we define the complex reconfiguration rules (with variables) that add a collection of components of each such type in a given configuration.

To identify a pattern in a configuration, the term representation of the configuration can effectively be parsed by considering the occurrences of the productions that generate a particular type of components (i.e., IDP and SP).

1.4 Main Contributions

In this thesis, we address the issues of modelling styles and reconfigurations for FIMs. The main contributions of this thesis are:

- Two UML models for FIMs where one of the models uses the standard UML notations to describe structural aspects while the other uses a UML profile in [33] to describe both the structural and reconfiguration aspects.

- A formal model of FIMs in ADR that captures both the structural and reconfiguration aspects of FIM architectures.

- A comparison of the modelling approaches; the comparison considers criteria relevant to the modelling of FIMs.

Modelling FIMs in UML The first approach adopts UML "as-is", namely it exploits only the basic linguistic features of UML to express architectural aspects of systems.

Class diagrams together with the constraints (i.e., related multiplicities and OCL) describe FIM architectural styles in UML. Classes model types of FIM components and associations model possible relationships among those components. In UML, one may require the use of tools and techniques to validate a configuration against its style. Typically, object diagrams suitably model simple scenarios. To represent such scenarios of FIMs, we consider a few configurations.

As anticipated in Section 1.2, UML "as-is" approach does not support abstract architectural components, therefore we do not address refinement of abstract FIM architectures. Also, there is no support for reconfigurations, therefore, we demonstrate how to introduce changes (e.g., to add FIM components in a CoT) directly in a FIM configuration.

Due to the limitations of the standard UML, the above UML model does not address specific architectural issues, namely refinement and reconfigurations. In addition to capture the details necessary to address these issues, we use the profile in [33] to model the same FIM details as specified by the above UML model.

While using the profile, composite structure diagrams (instance level) are used to

represent configurations where UML connectors attach ports of the components. To represent a FIM style, a class diagram represents an abstract (or refineable) CoT component and two production components whose internal structures are described by their corresponding composite structure diagrams (type level). These production components model different architectural views of FIMs. For instance, a production describes how CoTs can be legally connected in a chain. Similarly, another production describes federations of providers. Since UML has limited support for representing complex relationships in terms of classes and associations, global constraints are provided for the later production and they can be defined in OCL.

In this model, we describe different reconfiguration rules to support the changes within and across the FIM patterns. More precisely, reconfigurations that change the FIM patterns are suitably described. On the other hand, the reconfiguration rules that preserve the patterns may raise consistency problems. For instance, designers have to consistently update the UML diagrams after each application of these rules.

To generate FIM configurations, an abstract FIM component can be refined using the productions that respectively describe a federation of providers and a complex chain of CoTs (recursively). Also, the production which describes the federation can suitably be used to generate configurations of simple FIM scenarios.

To identify patterns in FIM configurations, corresponding composite structure diagrams (instance level) may effectively be analysed as they impose structure over a complex system (e.g., chain of CoTs).

A Formal Model of FIMs in ADR An architectural style of FIMs is formally described in ADR. A type hypergraph represents a set of architectural elements (i.e., FIM components and their ports). ADR productions define the legal connections between the FIM components.

Non-terminal hyperedges model abstract (or refineable) components and terminal

hyperedges model basic components. Nodes model interactions between the components. Typed hypegraphs represent FIM configurations whose associated ADR terms describe the way those configurations were actually generated. Also, these terms give proofs of construction of the corresponding configurations.

Abstract FIM components (e.g., CoT) are refined using corresponding productions in order to generate their configurations. A few basic reconfiguration rules are defined to add the FIM components into the configurations. Similarly, complex reconfiguration rules are defined to add the collection of the FIM components to the configurations.

ADR's refinement mechanism is exploited to generate specific configurations of FIM patterns. A term in ADR which describes an existing FIM configuration can effectively be used to generate the same configuration again. Also, an extended (or updated) FIM configuration can be generated by applying the reconfiguration rules. Since ADR terms formalise configurations, they can be parsed to identify the underlying FIM patterns of the configurations.

A Comparison of the Modelling Approaches Since various other approaches (such as other ADLs or graph-based formalisms) could possibly be used to model FIMs, setting up the criteria for comparing ADR with UML and the other approaches became necessary so as to assess them with regard to describing architectures of software systems. Among the other approaches, we considered a few graph-based formalisms including Baresi et al. [27], Hirsch et al. [61], and Le Métayer [73, 74] and, two ADLs including C2SADEL [69] and Acme [56]. Consequently, a comparison between the architectural modelling approaches is developed against those criteria.

More precisely, we analyse ADR, UML, and other ADLs for describing architectural aspects of FIMs according to the criteria by distinguishing them between "general" and "pattern-specific" criteria , which will be detailed in Section 6.1. The former pertain to aspects of software architectures common to all sufficiently complex sys-

tems. The latter can be considered as relevant to FIMs. However, they could also be germane to other classes of systems as they mainly concern generation and identification of actual systems' configurations required to satisfy an architectural scheme. We assess the support provided by the modelling approaches against those criteria. This enables us to develop a comparison of ADR with UML and the other approaches.

The summary of the comparison against the general criteria is given below:

- **Core architectural concepts.** For components, C2SADEL supports only a particular kind of components (i.e., C2 style) having a fixed number and kind of ports. For connectors, UML provides unsatisfactory support and it lacks connector semantics (i.e., connector types). However, one may choose from other UML notations (e.g., classes, component, etc.) to describe connectors. The rest of approaches suitably describe core architectural concepts including components, connectors, and configurations.

- **Architectural styles.** All of the approaches suitably describe vocabulary (i.e., type of architectural elements) of the style. However, the approaches including UML, the profile in [33], and Baresi et al. [27] may require OCL constraints to model complex associations. Such constraints are often not easy to express.

- **Style checking.** For style checking configurations in UML, the profile in [33] and Baresi et al. [27] require the use of validation tools and techniques. However, style checking of a configuration whose style uses OCL constraints is still problematic. In ADLs (Acme and C2SADEL), their run-time systems perform type-checking of configurations. In addition to this, one may further require the use of certain analysis tools in order to validate configurations against their styles. On the other hand, the graph grammar based approaches including ADR, Hirsch et al. [61], and Le Métayer [73, 74] use a formal mechanism that ensures construction of valid configurations.

- **Reconfigurations.** UML does not provide any mechanism to describe reconfigurations. Also, the profile in [33] provides unsatisfactory support to describe reconfigurations in general way as they may raise consistency problems. C2SADEL supports basic reconfiguration operations where a single component/connector can be added/removed at a time. On the other hand, the rest of the approaches suitably describe reconfigurations and, except in the approach of Hirsch et al. [61], also provide the means to check whether the changes preserve the style or not.

- **Refinement.** UML supports refinement through its generalisation concept which is not sufficient for architectural refinement. More precisely, UML does not support the notion of abstract architectural components. Baresi et al. [27] describe one-to-one structural mapping between the elements of two styles. They mainly focus on behavioural refinement. As observed in [70], C2SADEL provides limited support to deal with architectural refinement. On the other hand, the profile in [33], the graph grammar based approaches including ADR, Hirsch et al. [61], Le Métayer's [73, 74], and Acme suitably describe architectural refinement.

Now we give a summary of the comparison developed against pattern specific criteria:

- **Generating patterns.** To instantiate configurations (i.e., the FIM patterns as instances of the style), developers use rules of thumb in UML. Also, the profile in [33] and the approach of Baresi et al. [27] have the same limitations while creating initial configurations. On the other hand, the refinement rules in graph grammar based approaches including ADR, Hirsch et al. [61], and Le Métayer [73, 74] can be used to generate such configurations in a precise way by applying the rules in a specific order. The ADLs (Acme and C2SADEL) lack such a formal mechanism that can be used to precisely guide the developer to instantiate configurations.

- **Identifying patterns.** For identifying FIM patterns in configurations, object diagrams and instance level composite structure diagrams in UML can be analysed in an automated way (i.e., parsing the XMI representations) by enumerating instances of particular type. However, such an approach can be problematic while analysing the UML diagrams which represent complex configurations. Also, the profile in [33] and Baresi et al. [27] have the same limitations as UML. In graph grammar based approaches including ADR, Hirsch et al. [61], and Le Métayer [73, 74], the formal description of the configurations can effectively be parsed to identify patterns. In ADLs (Acme and C2SADEL), one may need to incorporate specific techniques in the executables in order to identify the underlying patterns of FIM configurations. To the best of our knowledge, these ADLs lack a mechanism for realising such techniques.

1.5 Related Work

A crucial work for our research is presented in [66], where various federation patterns are described in terms of security and trust requirements. We formalise the patterns that are based on direct trust relationships, namely relationships not relying on third parties (cf. [66]).

Delessy et al. [48] informally describe CoT in terms of a structural pattern. This pattern models a federation consisting a single IDP and multiple SPs. They use UML class diagram (with an OCL constraint) to model the structure of the pattern. Moreover, their focus is on how to centralise users' authentication related information within a single IDP by considering lower level (i.e., implementation) details in order to realise a given security mechanism. We give a formal representation and UML models that capture various patterns of FIMs in terms of their structural and reconfiguration aspects. We model FIM components (e.g, IDP) at a higher level of abstraction and we do

not consider implementation details of such components.

In [91] two new types of representation models are introduced; such models are called *dimension graph* (DG) and *pattern graph* (PG). The former shows the relationship (pattern-to-dimension) of a pattern with respect to various "dimensions" (i.e., life cycle stage, architectural level, security concern, business domain, type of pattern, and regulations/policies) of classification of security patterns. Instead, PG shows the relationship (pattern-to-pattern) of a pattern to other patterns. In [91], the focus is on representing properties (e.g., what pattern can be used for certain purposes) of the security patterns and relationships (e.g., what kind of patterns can be used at the next stage to realise a given pattern) between the security patterns using a metamodel in UML class diagram. The metamodel is then used to create DG and PG as its instances represented in UML object diagrams so as to introduce an improved classification of security patterns that helps the designers in analysing, finding, and understanding security patterns at each level of the development process. We propose a few generic architectural models that represent a class of FIM patterns in UML and ADR, which is a formal and a graphical approach. Our goal is the modelling of (direct security and trust) relationships between the collaborating organisations at an abstract architectural level. While using the formal approach, we represent FIM patterns as instances (i.e., typed hypergraphs) of the model whose corresponding terms precisely show their construction. Similarly, we use object diagrams and composite structure diagrams to instantiate FIM patterns in UML. Moreover, reconfiguration rules and their relationships have been defined in terms of their effects on the architectures of given FIM patterns.

An informal pattern system for authentication and authorisation infrastructures (AAI) has been described in [49] by showing the possible interactions between the patterns given in [48, 78]. In [49], the focus is on security aspects at implementation level and can be used directly in the software development process such as to deal

with security in web services. The purpose of our work is twofold. On the one hand, we aim to formally model FIMs as an architectural style and, on the other hand, to deal with changes in their architectures while respecting the FIM style. We provide a mechanism to formally model FIMs at an abstract level that may be used for concrete implementation for detailed analysis of the FIM properties (i.e., privacy) while allowing reconfiguration in the FIMs.

Finally, in [37] ADR has been promoted to model some aspects of SOA by proposing an architectural style for a modelling language featuring module composition. FIM patterns could be modelled following the approach described in [33] where ADR has been used as a formal support of style-based designs and reconfiguration of a UML profile for SOA. However, such approach would require OCL constraints to represent FIMs which are complex to deal with in the FIM context.

1.6 Structure of the Thesis

Chapter 2 briefly describes FIMs, FIM patterns, basics of software architectures, ADR, and UML. In this chapter, an overview of FIMs is given together with their security aspects and also existing FIM patterns are described. The basics of software architectures are briefly described by considering core architectural concepts and design vocabulary of an architectural style. This chapter also gives basic definitions of ADR and describes UML's concepts that are used to represent architectural aspects of software systems.

Chapter 3 describes a style using UML "as-is" to model architectural aspects of FIM systems. These aspects of FIM systems are modelled in Chapter 4 which uses the UML profile in [33] to describe a FIMs style.

Architectural aspects of FIMs are formalised in ADR in Chapter 5. In this chapter, a formal model of FIMs is given which characterises FIM configurations together with reconfigurations rules.

Chapter 6 gives a comparison of the modelling approaches used in this dissertation to model FIM systems. A few other modelling approaches are also considered for the comparison which could possibly be used to model such systems. The comparison is developed by fixing the criteria related to general and pattern specific architectural aspects. Further, we also evaluate the approaches used in Chapter 3, Chapter 4, and Chapter 5 by applying them to a case study. The available tool support for the modelling approaches is discussed also in this chapter.

Finally, concluding remarks and future work are given in Chapter 7.

Published work The formal model of FIM systems in ADR was first presented in [26]. The paper also presents a few reconfiguration rules to add identity providers in a FIM configuration. The main goal of the paper was to represent the style-preserving reconfigurations of FIM systems.

Chapter 2

Background

In this chapter, an overview of FIMs is given together with their security aspects. Also, the basic concepts of software architectures and the basic definitions of ADR are given. Finally, we describe the concepts in UML to model the structural aspects of software systems.

2.1 Federated Identity Management

FIMs are becoming ubiquitous and can be found in many different application contexts, including finance, education, eHealth, eGovernment. FIMs form an interesting class of distributed systems that allow group of organisations to "federate" in order to share services (or resources). FIMs allow cross-organisation authentication [79]. Typically, access to resources is governed by an access control system that requires user authentication. FIMs make users' authentication information available in a global context so that an organisation can have more business relationships with different organisations and it can be part of different federations.

We present the roles of FIMs by considering the UML use case diagram in Figure 2.1. The roles involved in FIMs are *users* (whose identity is to be federated),

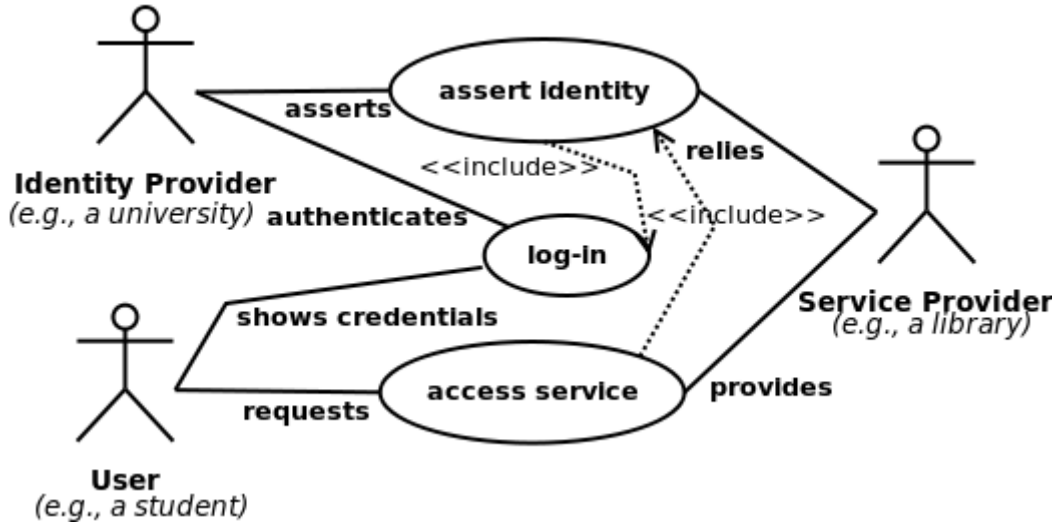


Figure 2.1: Roles in a FIM system

identity providers which vouch authentication statements for users (e.g., by issuing certificate), and *service providers* dispensing actual services. Hereafter, we abbreviate identity provider with IDP and service provider with SP. In Figure 2.1, a university (the IDP) where students (the users) access on line articles at one of the partner digital libraries (SP) form a FIM system.

The notion of *Circle of Trust* (CoT) is key to FIMs and permits to establish complex policies and obligations. In [30], a CoT is defined as a framework that specifies a common set of cooperation policies together with collaboration interfaces within a certain group of organisations having trusted relationships. Users provide verified identities in order to access resources shared by member organisations of the CoT. In FIMs, a CoT can be described as a federation of identity and service provider organisations.

Example 2.1 (cf. [60]) *A financial institution needs its users (employees, customers, etc.) to access services offered by a third party provider. The financial institution is the IDP managing the authentication information of its users to the third party SP.* ◇

In FIMs, an SP relies on the authentication information sent by the IDP when users request access to services so as to support *Single Sign-on* (SSO), namely allowing

users to authenticate only at their IDP without re-authenticating themselves to access services offered by SPs in the CoT.

2.2 The FIM Patterns

For FIMs, several recurring CoT scenarios exist which specify how IDPs and SPs are associated into federations. Kylau et al. [66] have examined such scenarios in terms of their trust requirements. Furthermore, they referred those scenarios as *patterns* by using term pattern in its literal meaning (i.e., a repeating theme). Specifically, their definition of a pattern[1] indicates a particular constellation of trust relationships between IDP and SP organisations to enable FIM. Furthermore, they differentiate their definition of a pattern from the usual notion of a *design pattern* which describes a solution to a recurring problem in a particular context.

In [48], structural (and behavioural) aspects of FIMs are informally represented as a pattern for a CoT where a single IDP is federated to multiple SPs. We consider some patterns based on direct trust relationships, namely relationships not relying on third parties. More precisely, following [66], we focus on (i) *Bilateral Federation* (BF), (ii) *Multiple IDPs Federation* (MIF), (iii) *Multiple SPs Federation* (MSF), and (iv) *Arbitrary Federation* (AF); also we introduce and model an additional pattern called *Chain of CoTs*.

The differences among those patterns mainly lie on how they manage trust and what security threats they are exposed to (e.g., privacy of user identities, business data, access control, authentication). More precisely, patterns **BF**, **MIF**, **MSF**, and **AF** are ordered according to the security threats they are subject to, pattern **BF** being the "most robust". We now briefly comment on the security threats hindering each pattern; the reader is referred to [66] for details.

[1] We use this term for FIM patterns in this dissertation.

Bilateral Federation (BF) In **BF**, a single IDP is federated to a single SP. The IDP is supposed to follow adequate procedure to register the users (e.g., by creating their credentials). Also, the IDP is required to adapt suitable identity mapping (i.e., mapping individual users to designated roles in the organisation) and authentication procedures so as to avoid unauthorised access to the SP. In addition to these requirements, the IDP and the SP agree to deal with the private data according to common policies. Since the access to services is mediated by the IDP, the latter is aware of users' activities (e.g., how often users communicate with the SP). The IDP may exploit this kind of information to acquire knowledge on users' behaviour and this is regarded as a threat to users' privacy. Also, SP receives information related to users' identity from the IDP hence the SP might disclose (i.e., to other SPs) such information without the consent of the IDP or the user.

Example 2.2 *An airline is federated to a hotel to allow a traveller to book a room after booking a flight. The airline acts as an IDP while the hotel is the SP. According to FIM pattern **BF**, a traveller can book a room after booking a flight.*

Example 2.2 describes a scenario where an airline is federated to a hotel.

Multiple IDPs Federation (MIF) In **MIF**, a single SP is federated to multiple IDPs; users may be registered at several IDPs and they notify the SP about which IDPs will be used for authentication. The additional threat with respect to pattern **BF** is that some or all IDPs might decide to cross-check the information about the accesses to the SP for accumulating data on individual users. Example 2.3 describes a scenario where multiple IDPs are federated to a single SP.

Example 2.3 *Suppose that the scenario described by Example 2.2 has to be modified so that travellers are authenticated either by the airline company or via another account on a train company. In other words the airline and the train companies act as*

*IDPs while the hotel is the SP. According to FIM pattern **MIF**, a traveller can book a room after booking a flight or a train.*

Multiple SPs Federation (MSF) In **MSF**, a single IDP is federated to multiple SPs; a typical situation in this case is that delegation of user authentication is necessary. For instance, a service in the federation may be delegated (by an IDP or by another SP) to provide users' credentials if it needs to invoke other services in the federation. The additional threats with respect to pattern **MIF** include (*i*) unauthorised delegation of authentication and (*ii*) "collusion" of SPs to accumulate identity information. The first threat may happen when users invoke a complex service making further invocations to other SPs in the federation. The second threat allows SPs to correlate their information and accumulate data on users. Example 2.4 describes a scenario where a single IDP is federated to multiple SPs.

Example 2.4 *While extending the scenario described by Example 2.2 to allow travellers to be authenticated by the airline company to access the hotel and a car rental company. The airline acts as an IDP while the hotel and the car rental company are SPs. According to FIM pattern **MSF**, a traveller can book a room and a car after booking a flight.*

Arbitrary Federation (AF) Pattern **AF** is the most vulnerable as it allows the free combination of patterns **BF**, **MIF**, and **MSF** and is exposed to all their threats. Example 2.5 describes a scenario where multiple IDPs are federated to multiple SPs.

Example 2.5 *While combining the scenarios described by Example 2.3 and Example 2.4, travellers can either be authenticated by the airline company or the train company to access the hotel and the car rental company. The airline and the train companies act as IDPs while the hotel and the car rental company are SPs. According*

*to FIM pattern **AF**, a traveller can book a room and a car after either booking a flight or a train.*

Chain of CoTs In this pattern, two or more CoTs are connected in a chain to enable FIM where users from an IDP of one CoT may access services in the other CoTs that are participating in the chain. This pattern is exposed to the same threats as with the pattern **AF**. Example 2.6 describes a scenario where two CoTs can be connected in a chain.

Example 2.6 *In a chain of CoTs pattern, a CoT described by Example 2.3 is attached to another CoT described by Example 2.4. In this chain, travellers are authenticated by either the airline or the train companies (i.e., IDPs) in order to access the hotel and the car rental companies (i.e. SPs). Consequently, this pattern enable travellers to book a room and a car after either booking a flight or a train.*

2.3 A Few Real Examples of FIM Systems

In this section, we describe a few real examples of the FIM systems taken from [11] where several FIM implementations can be found in various application contexts (e.g., education, eGovernment, Healthcare, eCommerce, eEducation, etc.). The main criterion we fix to choose such examples is to find the FIM implementations that establish legally binding CoTs [29] based on the existing FIM specifications (e.g., the Liberty Alliance [22]).

Example 2.7 (cf. [2]) *In* New York State *(NYS), 12 Regional Information Centres (RICs) are responsible to provide technology services to its 697 school districts associated to the* Boards of Cooperative Educational Services *(BOCES). One of these RICs called EduTech directly supports 47 schools districts. Another RIC called SCT provides technology services to 9 school districts. DataMentor application run by EduTech shows*

district-wide performance that enables teachers and administrators to create their own assessments for evaluating progress and identifying topics that need to be focused. SCT runs an application called QuizMaker to help teachers in targeting and building strength in their weak areas. After deploying a FIM system, teachers in all of the districts associated to the RICs are able to seamlessly access these applications.

Example 2.7 describes a complex scenario of an existing large scale FIM system which will be modelled in Section 6.5.3. Also, we describe a few more FIM examples for the sake of demonstrating the adoption of FIM systems by organisations in the real world. For instance, Example 2.8 is to illustrate how FIMs can undergo reconfigurations and Example 2.9 is to show that FIMs are used in industrial scenarios. The FIM scenarios described by those examples could be modelled in the same way as the FIM scenario described by Example 2.7.

Example 2.8 (cf. [3]) *The government of Denmark deploys a FIM system to enable its citizens to conveniently access various public digital services (e.g., TAX self Service, Student Loans, State Education Grant, etc.) operated by the designated agencies/departments. In 2008, the Danish government introduced a portal called* MyPage *(or* www.borger.dk*) which acts as an IDP and 12 SPs which provide access to 30 applications in the first wave of realising the FIM system. To further extend the FIM system later in the same year, the government introduced 3 portals and 25 SPs which provide access to 75 applications.*

Example 2.9 (cf. [4]) *A FIM system has been in production at General Motors (GM) which is one of the world's largest automaker. In the FIM system, the IDP called* MySocrates *provides authentication information on request to services offered by a third party SP. This allows GM's employees to access many of the outsourced HR services (i.e., health benefits, expense reporting, 401(k), etc.) provided by the SP. Further,*

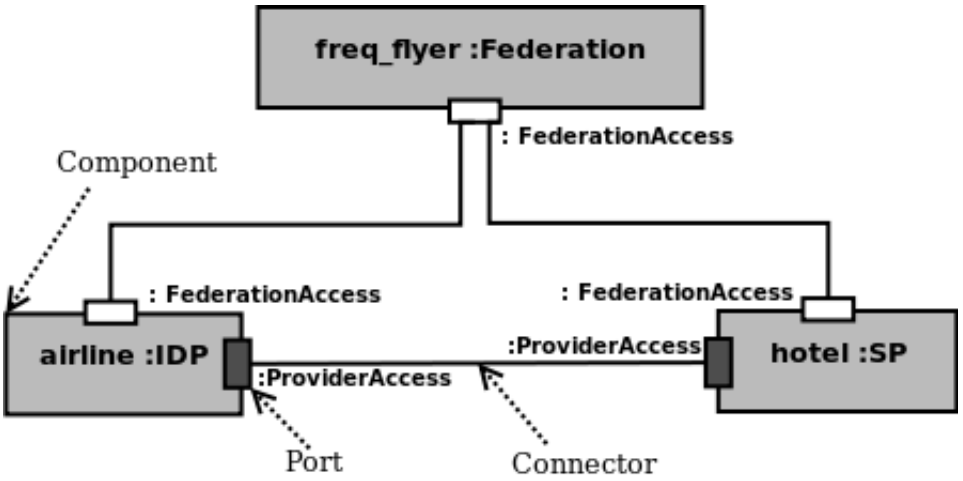


Figure 2.2: Architecture of a FIM system

GM has also been exploring the ways to extend the federation via adding two more SPs namely, DealerWeb *and* OnStar. *DealerWeb would enable GM's dealers worldwide to access variety of services (e.g., online auctions). OnStar provides variety of services including in-vehicle safety, communication, and security services.*

2.4 Basics of Software Architectures

Software architectures (SAs) are considered as the high-level descriptions that specify core architectural concepts including *components*, *connectors*, and *configurations*. In an architectural description, components are central computational elements of a system and connectors model a kind of interaction between them. Example 2.2 describes a scenario where airline and hotel entities are the FIM components and their interactions (i.e., service calls) can be modelled as connectors.

A configuration describes an architecture where components and connectors are composed together. For instance, in Example 2.2 an airline (i.e., IDP) is connected to a hotel (i.e., SP) in a FIM system that may allow travellers to book a room after booking a flight. This configuration represents FIM pattern **BF** where a single IDP is federated to

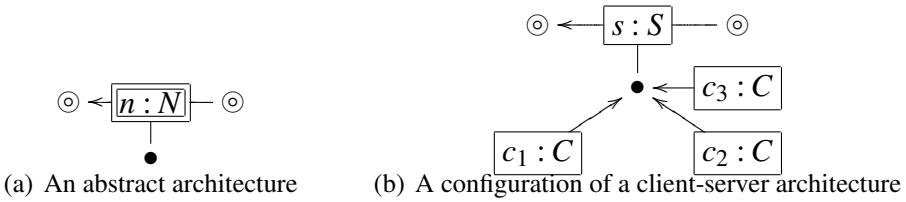
a single SP and Figure 2.2 shows such a configuration. Moreover, a configuration may be used for architectural analysis (i.e., conformance) and its primary role is to provide means of communication among the different stakeholders to understand the system at the desired level of abstraction. For instance, a valid configuration of the CoT adheres to the essential architectural properties specified in the FIM patterns. In this case, one has to check conformance of the FIM configurations against these patterns.

The *vocabulary* of an SA describes a set of architectural elements (e.g., component and port) types. These element types are instantiated in order to create architectural configurations. For example, a component of type IDP in the FIMs encapsulates the functionality of an identity provider. One may need to create multiple instances of component of type IDP while generating configurations of FIM patterns **MIF** and **AF**.

2.5 Architectural Design Rewriting

The *Architectural Design Rewriting* (ADR) approach [37, 38] permits to design hierarchical, style conformant and reconfigurable software architectures. The main features of ADR include a rule-based approach, hierarchical design (or structured graph), and an algebraic presentation. Bruni and Lluch-Lafuente in [34] raised some drawbacks related to using flat, unstructured graph for the design and analysis of software systems. Based on their experience of using ADR to model such systems, they argue that such drawbacks can be alleviated by using hierarchical, structured graphs. In order to tackle the complexity of software systems, they propose the use of structured graph instead of unstructured graphs to provide efficient formal reasoning on the architectures of such systems by superimposing some structure on the graphs describing these architectures.

We borrow the technical definitions in this section from [37] (where more details can be found).



(a) An abstract architecture (b) A configuration of a client-server architecture

Figure 2.3: A client-server architecture

Example 2.10 *We illustrate the approach through a simple scenario of a client-server application where* "clients" *can connect to a* "server" *within a* "network" *and Figure 2.3 represents such a scenario. Figure 2.3(a) shows an abstract architecture which represents the black-box view of a system. The abstract architecture is replaced with a configuration given in Figure 2.3(b) where several clients are connected to a server. In such a configuration, each client of type C communicate with the server of type S via a* "client port" *(node •) that models interconnection between the clients and the server. Similarly, a "*chaining port*" (node ◎) connects two servers to extend the chain of servers. Since clients can connect to the server, we model reconfigurations that describe such kind of change into the configurations. To demonstrate the changes in the client-server architecture, we also show how such reconfigurations can be applied.*

Software architectures are modelled in ADR as hypergraphs whose edges represent components and nodes (vertices) represent interconnections between components.

Definition 1 *An ADR* (hyper)graph *is a tuple* $G = \langle V, E, t \rangle$ *where* V *is the set of nodes,* E *is the set of edges, and* $t : E \to V^*$ *is the tentacle function.*

The vocabulary of an architecture is given by a distinguished graph, the *type graph*, over which graphs are typed. Moreover, ADR edges are partitioned in *terminal* (not refinable) and *non-terminal* (refinable) edges.

Example 2.11 *An ADR graph on the sets of nodes* $\{◎, •\}$ *and edges* $\{N, S, Cs, C\}$ *can be graphically represented by a graph (*H*) in Figure 2.4. The tentacle function is represented by the lines connecting edges ordered clockwise starting from the arrow-*

*headed tentacle; the tentacle function of the type graph (*H*) maps edges N and S to* $[\circledcirc, \circledcirc, \bullet]$ *and edges Cs and C to* $[\bullet]$*. The doubly-lined boxes N and Cs represent non-terminal edges while single-lined boxes S and C are terminal edges.* $\diamond$

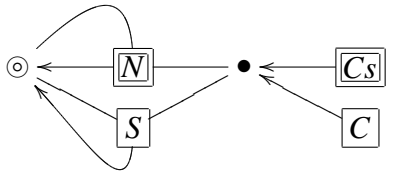


Figure 2.4: A type graph (H) for the client-server architectural style

In ADR, a type graph on which the graphs are typed describes vocabulary of an architectural style. The type graph in Figure 2.4 can be formally defined as

$$V_H = \{\circledcirc, \bullet\} \qquad t_H : \begin{cases} N, S \longmapsto [\circledcirc, \circledcirc, \bullet] \\ Cs, C \longmapsto [\bullet] \end{cases}$$
$$E_H = \{N, S, Cs, C\}$$

In a type graph, an edge models the type of components and a node models an interaction point between the components. In the type graph (H) given in Figure 2.4, the non-terminal edges N and Cs represent a network and a collection of clients, respectively. The terminal edges S and C represent a server and a client, respectively. More precisely, non-terminal edges model abstract (refinable) components which can be replaced by a complex graph. For instance, the non-terminal N models a network at an abstract level. In this case, the non-terminal N could be replaced with a graph where several clients can be attached to a server. On the other hand, terminal edges model basic components which do not require further refinement.

Definition 2 *A graph G is* typed *over a graph H when G is homomorphic to H, namely when there are the functions* $f_V : V_G \to V_H$ *and* $f_E : E_G \to E_H$ *preserving the tentacle functions such that* $f_V^* \circ t_G = t_H \circ f_E$*, where* f_V^* *is the homomorphic extension of* f_V *to* V_G^**.*

In ADR, typed graphs are defined as graphs equipped with typing morphism. Ar-

chitectures are modelled using *designs* in ADR which represent architectural components with their interconnections.

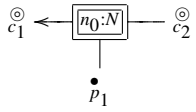
Definition 3 *A* design *is a graph with interface, i.e. a triple* $d = \langle L_d, R_d, i_d \rangle$*, where*

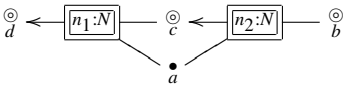
- L_d *is a (typed) graph consisting only of a non-terminal and distinct nodes attached to its tentacles,*
- R_d *is a (typed) graph without non-terminal edges,*
- *and* $i_d : V_{L_d} \to V_{R_d}$ *is a total function.*

In ADR, architectures are built using a set of composition operations known as *design productions*.

Definition 4 *A* (design) production *p is a tuple* $\langle L_p, R_p, i_p, l \rangle$ *where* L_p *is a (typed) graph consisting only of a non-terminal edge and by distinct nodes attached to its tentacles;* R_p *is a (typed) graph containing both terminal and non-terminal edges;* $i_p : V_{L_p} \to V_{R_p}$ *is a type preserving function; and* l *is a bijection mapping the non-terminal edges of* R_p *on an initial segment* $[1, 2, \ldots, n_p]$ *of positive numbers.*

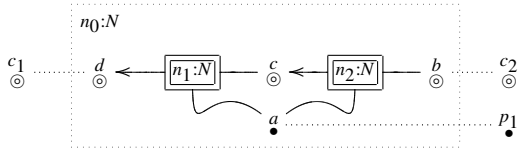
Example 2.12 *[Like graphs, ADR productions have a convenient graphical representation which we illustrate with an example.] Given*

graph G_1 

graph G_2 

which can be typed over the graph (H) from Figure 2.4 in the obvious way, and $l : n_i \longrightarrow i \ \in \{1, 2\}$ *the trivial function, the production having* G_1 *as left-hand-side (LHS) and* G_2 *as right-hand-side (RHS) (together with the homomorphisms (cf. Definition 2)) can be drawn as*

$$\mathbf{nets} : N \times N \to N$$



(2.1)

where the outermost dotted box corresponds to G_1, the inner graph is G_2 (with the explicit typing given by the homomorphism (cf. Definition 2)), and the dotted lines represent the mapping from the nodes of G_1 to those of G_2. ◇

Productions allow top-down design by refinement, bottom-up typing of the actual architecture and well-formed composition of the architectures. The set of design productions together with the type graph represent the architectural style.

If non-terminal edges are considered as 'types' (of architectures), ADR productions have a convenient "functional" reading. For instance, the production of Example 2.12 can be abstractly thought of as a function that takes two architectures of 'type' N (namely obtained by refining N) and returns a new architecture of type N. This allows one to consider productions as constructors of a sorted algebra of architectures where the terms of such an algebra yield architectural configurations. For instance, if x, y, and z are architectures of type N, the term

$$\mathbf{nets}(\mathbf{nets}(x,y),z)$$

is an architecture of type N.

The formal definition of the production **nets** in (2.1) can be given as under:

The morphism between the LHS typed graph G_1 of the production **nets** and the type graph H (Figure 2.4) is given below

$$f_{V_{L_{\mathbf{nets}}}} : \begin{cases} c_1 \longmapsto \circledcirc \\ c_2 \longmapsto \circledcirc \\ p_1 \longmapsto \bullet \end{cases} \qquad f_{E_{L_{\mathbf{nets}}}} : n_0 \longmapsto N$$

Similarly, the morphism between the RHS typed graph G_2 of the production **nets** and the type graph H (Figure 2.4) is given below

$$f_{V_{R_{\mathbf{nets}}}} : \begin{cases} b \longmapsto \circledcirc \\ c \longmapsto \circledcirc \\ d \longmapsto \circledcirc \\ a \longmapsto \bullet \end{cases} \qquad f_{E_{R_{\mathbf{net}}}} : \begin{cases} n_1 \longmapsto N \\ n_2 \longmapsto N \end{cases}$$

In production **nets**, function $i_{\mathbf{nets}}$ maps the interface nodes of $L_{\mathbf{nets}}$ with the interface nodes of $R_{\mathbf{nets}}$ and function $l_{\mathbf{nets}}$ is the bijective mapping of the non-terminals in the $R_{\mathbf{nets}}$ on an initial segment $[1,2,...,n_{\mathbf{nets}}]$

$$i_{\mathbf{nets}} : \begin{cases} p_1 \longmapsto a \\ c_1 \longmapsto d \\ c_2 \longmapsto b \end{cases} \qquad l_{\mathbf{nets}} : \begin{cases} n_1 \longmapsto 1 \\ n_2 \longmapsto 2 \end{cases}$$

Figure 2.5 describes a few productions which can be used to generate configurations of architectures of type N. The production **server** will be used to generate architectural configuration of N consisting of a S and a Cs. The RHS of the production **server** takes a non-terminal edge Cs attached to a node of type $\bullet$ which is then exported in the interface of the production. Further, the non-terminal edge of type Cs in production **server** will be refined using the corresponding productions (i.e., **clients**, **client**, and **noclient**).

The production **clients** generates an architectural configuration of the Cs consisting of two Cs. The RHS of the production **clients** combines two non-terminal edges of type Cs to the same $\bullet$ which is then exported in the interface of the production. In this way, the Cs will all share the $\bullet$ node with the same S edge.

The production **client** is meant to generate architectural configuration of the Cs

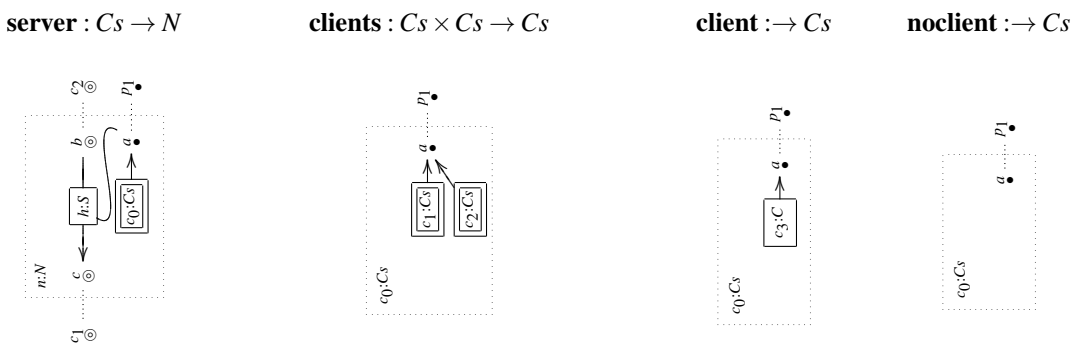


Figure 2.5: The productions for the network

consisting of a single C. The RHS of the production **client** takes nothing and attaches the C to a $\bullet$ which is then exported in the interface of the production. In this way, the Cs shares a node $\bullet$ with the the same S edge.

The production **noclient** is meant to generate an architectural configuration of a Cs for empty design. The RHS of the production **noclient** takes nothing and uses node $\bullet$ to share with the server and then this node is exported in the interface of the production.

The formal definitions of the productions described in Figure 2.5 are given below:

(a) The typed graph $L_{\mathbf{net}}$ (b) The typed graph $R_{\mathbf{net}}$

Figure 2.6: LHS and RHS graphs of production **server** typed over the graph H

The morphism between the LHS typed graph of the production **server** in Figure 2.6(a) and the type graph H (Figure 2.4) is given below

$$f_{V_{L_{\mathbf{server}}}} : \begin{cases} c_1 \longmapsto \circledcirc \\ c_2 \longmapsto \circledcirc \\ p_1 \longmapsto \bullet \end{cases} \qquad f_{E_{L_{\mathbf{server}}}} : n \longmapsto N$$

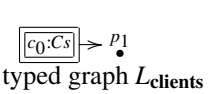
Similarly, the morphism between the RHS typed graph of the production **server**

and the type graph H (Figure 2.4) is given below

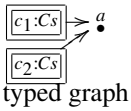
$$f_{V_{R_{\textbf{server}}}} : \begin{cases} c \longmapsto \circledcirc \\ b \longmapsto \circledcirc \\ a \longmapsto \bullet \end{cases} \qquad f_{E_{R_{\textbf{server}}}} : \begin{cases} h \longmapsto S \\ c_0 \longmapsto Cs \end{cases}$$

In production **server**, function $i_{\textbf{server}}$ maps the interface nodes of $L_{\textbf{server}}$ with the interface nodes of $R_{\textbf{server}}$ and function $l_{\textbf{server}}$ is the bijective mapping of the non-terminals in the $R_{\textbf{server}}$ on an initial segment $[1,2,..,n_{\textbf{server}}]$

$$i_{\textbf{server}} : \begin{cases} c_1 \longmapsto c \\ c_2 \longmapsto b \\ p_1 \longmapsto a \end{cases} \qquad l_{\textbf{server}} : c_0 \longmapsto 1$$



(a) The typed graph $L_{\textbf{clients}}$



(b) The typed graph $R_{\textbf{clients}}$

Figure 2.7: LHS and RHS graphs of production **clients** typed over the graph H

The morphism between the LHS typed graph of the production **clients** in Figure 2.7(a) and the type graph H (Figure 2.4) is given below

$$f_{V_{L_{\textbf{clients}}}} : p_1 \longmapsto \bullet \qquad f_{E_{L_{\textbf{clients}}}} : c_0 \longmapsto Cs$$

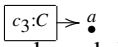
Similarly, the morphism between the RHS typed graph of the production **clients** and the type graph H (Figure 2.4) is given below

$$f_{V_{R_{\textbf{clients}}}} : a \longmapsto \bullet \qquad f_{E_{R_{\textbf{clients}}}} : \begin{cases} c_1 \longmapsto Cs \\ c_2 \longmapsto Cs \end{cases}$$

In production **clients**, function $i_{\textbf{clients}}$ maps the interface nodes of $L_{\textbf{clients}}$ with the interface nodes of $R_{\textbf{clients}}$ and function $l_{\textbf{clients}}$ is the bijective mapping of the non-terminals in the $R_{\textbf{clients}}$ on an initial segment $[1,2,..,n_{\textbf{clients}}]$

$$i_{\textbf{clients}} : p_1 \longmapsto a \qquad\qquad l_{\textbf{clients}} : \begin{cases} c_1 \longmapsto 1 \\ c_2 \longmapsto 2 \end{cases}$$

(a) The typed graph $L_{\textbf{client}}$



(b) The typed graph $R_{\textbf{client}}$

Figure 2.8: LHS and RHS graphs of production **client** typed over the graph H

The morphism between the LHS typed graph of the production **client** in Figure 2.8(a) and the type graph H (Figure 2.4) is given below

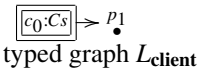
$$f_{V_{L_{\textbf{client}}}} : p_1 \longmapsto \bullet \qquad\qquad f_{E_{L_{\textbf{client}}}} : c_0 \longmapsto Cs$$

Similarly, the morphism between the RHS typed graph of the production **client** and the type graph H (Figure 2.4) is given below

$$f_{V_{R_{\textbf{client}}}} : a \longmapsto \bullet \qquad\qquad f_{E_{R_{\textbf{client}}}} : c_3 \longmapsto C$$

In production **client**, function $i_{\textbf{client}}$ maps the interface nodes of $L_{\textbf{client}}$ with the interface nodes of $R_{\textbf{client}}$. Since there is no non-terminal in the production **client**, function $l_{\textbf{client}}$ does not represent the mapping.

$$i_{\textbf{client}} : p_1 \longmapsto a$$



(a) The typed graph $L_{\textbf{client}}$

(b) The typed graph $R_{\textbf{client}}$

Figure 2.9: LHS and RHS graphs of production **noclient** typed over the graph H

The morphism between the LHS typed graph of the production **noclient** in Figure 2.9(a) and the type graph H (Figure 2.4) is given below

$$f_{V_{L_{\textbf{noclient}}}} : p_1 \longmapsto \bullet \qquad\qquad f_{E_{L_{\textbf{noclient}}}} : c_0 \longmapsto Cs$$

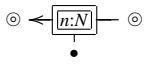
Similarly, the morphism between the RHS typed graph of the production **noclient** and the type graph H (Figure 2.4) is given below

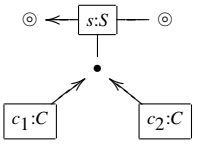
$$f_{V_{R_{\textbf{noclient}}}} : a \longmapsto \bullet$$

In production **noclient**, function $i_{\textbf{noclient}}$ maps the interface nodes of $L_{\textbf{noclient}}$ with the interface nodes of $R_{\textbf{noclient}}$. Since there is no non-terminal in the $R_{\textbf{noclient}}$, function $l_{\textbf{noclient}}$ does not represent the mapping.

$$i_{\textbf{noclient}} : p_1 \longmapsto a$$

Generating configurations To illustrate how legal configurations can be generated using the productions given in Figure 2.5 we consider the graphs[2] G_1 and G_2 in (2.2)

$G_1 =$ 

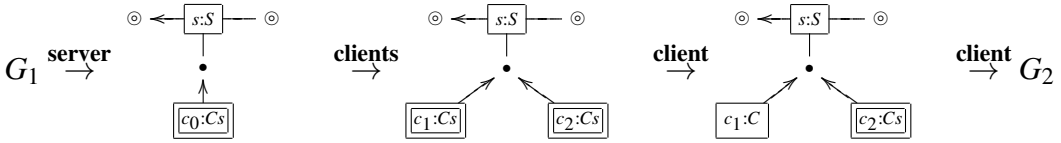
$G_2 =$ 

(2.2)

and show how G_1 can be refined into the configuration G_2.

[2]Graphs G_1 and G_2 are typed over the graph depicted in Figure 2.4 in the obvious way.

The initial sequence of reductions is



In the first step, **server** is applied to generate the server s; then the edge c_0 is refined by applying **clients** yielding two clients of type Cs. In the next step, the edge c_1 is refined by applying **client** yielding a client of type C. Finally, the edge c_2 is refined by applying **client** yielding another client of type C. As a result, configuration G_2 is obtained which represents the two clients of type C that are attached to the server of type S. The term-like representation of G_2 is

$$\mathbf{server}(\mathbf{clients}(\mathbf{client},\mathbf{client})) \tag{2.3}$$

which highlights the hierarchical structure of the configuration G_2. In this way, configurations of the client-server architecture can be generated.

Describing reconfigurations In ADR, the design productions can be given an algebraic formulation where a term describes a particular style-proof. For instance, term of (2.3) describes how a legal configuration of a client-server architecture is built by using the productions given in Figure 2.5. Style-preserving reconfigurations are operated at the level of style-proofs by exploiting term rewriting over style-proof terms. A graph transformation rule can be represented as a rewrite rule $L \longrightarrow R$ where L and R are terms of the same type (interface graph). This condition enforces style preservation by constraining both the L and the R of each reconfiguration rule to have the same type of the interface graph [38].

For instance, let us define a reconfiguration rule to add components (at abstract level) to the architectures of type N in Example 2.12 and let $\mathbf{net} : N \to N$ be a produc-

tion that takes a configuration of type N and returns another configuration of type N. To illustrate this, assume x and y are architectures of type N and consider the reconfiguration rule

$$add(y) : \mathbf{net}(x) \longrightarrow \mathbf{nets}(\mathbf{net}(x), y) \tag{2.4}$$

where a sub-term $\mathbf{net}(x)$ of type N on the LHS of the rule is replaced by term $\mathbf{nets}(\mathbf{net}(x), y)$ of the same type on the RHS. For instance, the reconfiguration

$$\mathbf{nets}(\mathbf{net}(x_1), y_1) \longrightarrow \mathbf{nets}(\mathbf{nets}(\mathbf{net}(x_1), y_2), y_1)$$

is obtained by applying the rule $add(y_2)$ in (2.4) to the subterm $\mathbf{net}(x_1)$ on the LHS so as to yield the term on the RHS where y_2 is attached in the new configuration by means of the constructor **nets**.

To further illustrate how ADR reconfiguration rules can describe changes in the client-server architecture (cf. Example 2.10) we consider the following rules

$$addClient_1 : \mathbf{client} \longrightarrow \mathbf{clients}(\mathbf{client}, \mathbf{client}) \tag{2.5}$$

$$addClient_2 : \mathbf{noclient} \longrightarrow \mathbf{clients}(\mathbf{client}, \mathbf{noclient}) \tag{2.6}$$

$$addClients(X) : \mathbf{client} \longrightarrow \mathbf{clients}(\mathbf{X}, \mathbf{noclient}) \tag{2.7}$$

These rules allow one to add a single client and a collection of clients into a configuration of the client-server system. More precisely, the rule $addClient_1$ in (2.5) and the rule $addClient_2$ in (2.6) add a single client of type C while the rule $addClients(X)$ in (2.7) adds a collection of clients via an edge of type Cs. Notice that in these rules the LHSs and the RHSs terms have the same type. In this way, ADR guarantees by construction that if all reconfiguration rules preserve the types then any derivation will not change the style.

Figure 2.10 illustrates the reconfiguration of a client-server architecture by applying

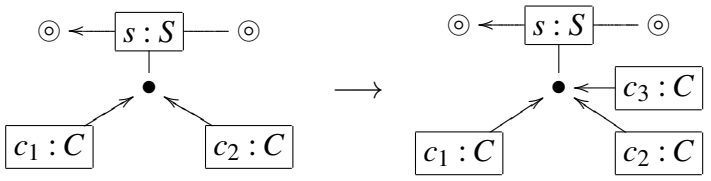


Figure 2.10: Rule to add a client (from left to right)

the rule (2.5). The LHS graph shows configuration consisting two clients of type C and a server of type S. This architecture is reconfigured by connecting an additional client to the server. To illustrate such a change in the configuration obtained by applying the rule (2.5), transition

$$\textbf{server}(\textbf{clients}(\textbf{client},\textbf{client})) \longrightarrow \textbf{server}(\textbf{clients}(\textbf{clients}(\textbf{client},\textbf{client}),\textbf{client}))$$

describes the reconfiguration where the LHS term defines configuration having two clients c_1 and c_2 attached to the server s while the RHS term defines the new configuration that attaches an additional client c_3. In this reconfiguration, the subterm **client** of type Cs on the LHS is replaced with a new term $\textbf{clients}(\textbf{client},\textbf{client})$ of same type on the RHS and such a transition preserves the style.

2.6 UML for Structural Modelling

This section describes UML diagrams used in the thesis to model FIMs. Specifically, we discuss

- class diagram
- object digram
- composite structure diagram
- package diagram

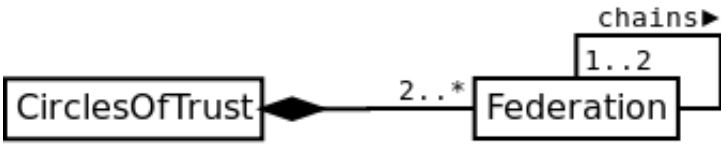


Figure 2.11: A class diagram showing a FIM system

Also, we give a brief description of OCL *constraints* and UML *profiles*.

2.6.1 Class diagram

A class diagram represents classes (or domain concepts) and their relationships. Figure 2.11 shows a UML class diagram that represents relationships between these classes of the FIMs (attributes/properties and methods are immaterial).

The relationships between two classes A and B of interest are association, composition, and inheritance.

- An association establishes that objects in A contains a reference to objects in B. *Multiplicity* value shown on each end of the association describes how many objects of the class will participate in the association. The default multiplicity is 1. An association can have an optional name and is graphically represented by lines. For instance, Figure 2.11 shows an association named `chains` that relates objects of type `Federation`.

- An association which represents that objects in A own or contain objects in B is called composition. A link with filled diamond arrow-head represents composition. For instance, in Figure 2.11 an object of type `CirclesOfTrust` may consist of two or more `Federations`.

- An association which represents that a class (i.e., subclass) is sub-type of another class (i.e., superclass) is called inheritance. The generalisation arrow (with

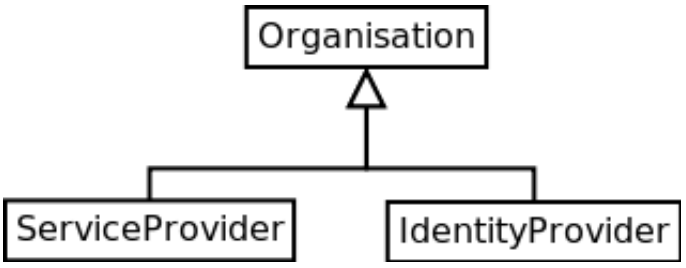


Figure 2.12: A class diagram showing inheritance relationship

filled diamond) describes such a relationship. For instance, in Figure 2.12 the classes `IdentityProvider` and `ServiceProvider` are specialisations of the class `Organisation`.

2.6.2 Object diagram

Typically, object diagrams are used to represent snapshots of the objects in systems during their execution. Since connections among objects are complicated, object diagram usually consider simple scenarios [52].

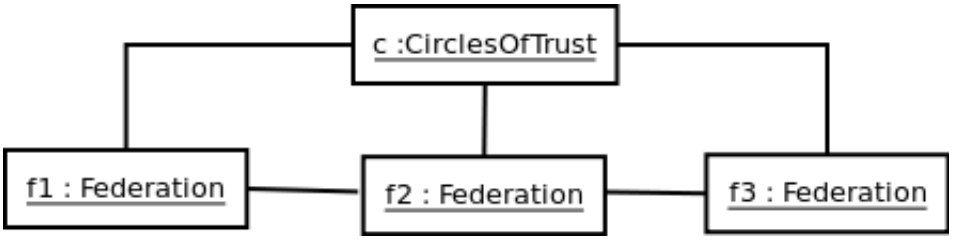


Figure 2.13: An object diagram showing a FIM configuration

Object diagrams use a simple notation; an object is drawn as a rectangle tagged with the (optional) object name followed by its class; a link between two objects represents a possible interaction between them (a link can only connect objects corresponding to the association between their classes). Figure 2.13 shows an object diagram of a FIM system corresponding to the class diagram in Figure 2.11 where federations `f1, f2,` and `f3` are associated to each other in a circle of trust `c`.

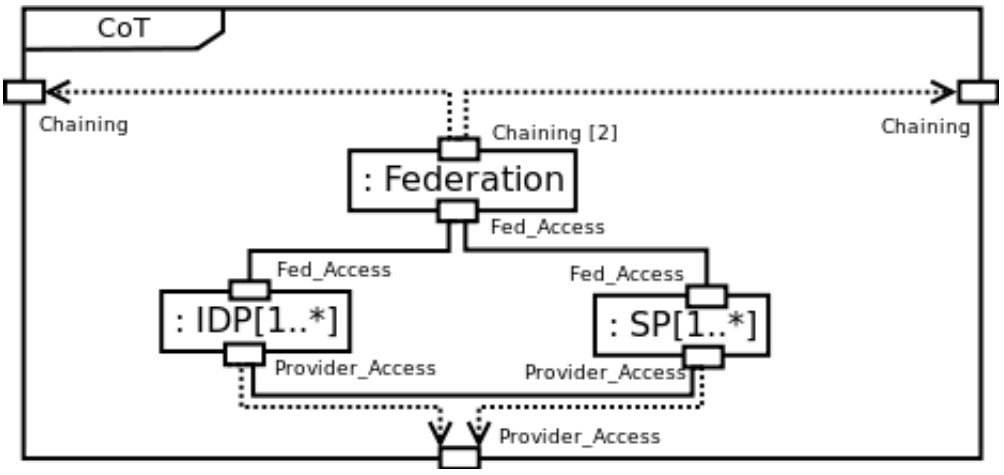


Figure 2.14: A composite structure diagram (type level)

2.6.3 Composite structure diagram

In UML, *composite structure diagrams* (hereafter referred as structure diagrams) may represent the internal structure of a classifier (e.g., class or components) in terms of interconnected instances collaborating over communication links. In these diagrams, collections of instances are called *parts*. For instance, the internal structure of the CoT in Figure 2.14 consists of three different parts represented by their respective classes, namely a `Federation`, one or more `IDP`s, and one or more `SP`s.

Parts in structure diagram can be associated to other parts; an association may have multiplicity defined on its each end. If multiplicity of an association end is not explicitly defined then multiplicity of the part it attaches will be used. Links relate parts with each other or with their enclosing classifier (e.g., class). UML links are graphically represented by *connectors*, namely lines ending into *ports*. A port represents a distinct interaction point of the instances of a classifier and its behaviour can be defined in terms of *provided interfaces* and *required interfaces*. Note that we do not model behaviour of FIMs. Therefore, ports are used in this dissertation without interfaces.

There are two types of connectors in UML called *assembly* and *delegation* connectors, respectively. Assembly connectors are represented by solid lines and specify

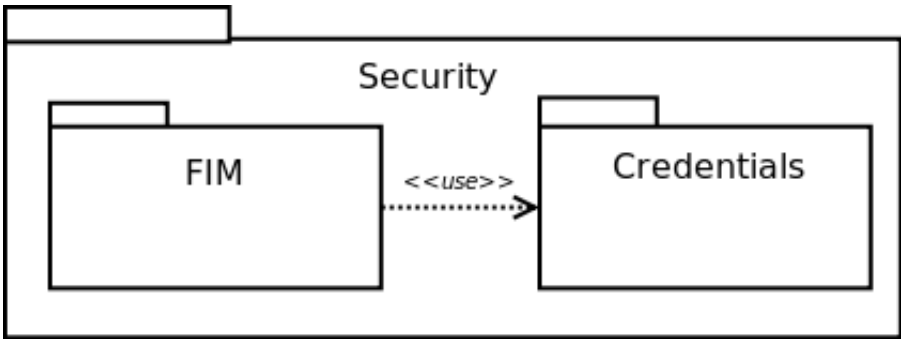


Figure 2.15: A package diagram

connections between the ports of two or more instances. Delegation connectors are represented by dashed lines and connect instances to their enclosing class.

A structure diagram yields a context specific view of a classifier. For instance, Figure 2.14 yields the context of a CoT having a federation of `IDPs` and `SPs`; the outermost ports permit to chain CoTs. Similarly, another context specific view of a CoT in FIMs may be described as a chain of CoTs.

2.6.4 Package diagram

In UML, a package is used to organise the model elements (e.g., classes, packages, etc.) into logically related groups. For example, a credential package may contain classes related to different types of security tokens (e.g., Kerberos, X.509, Smartcard, etc.) that can be used in a FIM system.

As shown in Figure 2.15 a package is graphically represented as a folder (The name of a package can be shown either inside the body of the folder or on its tab).

A package may contain other (sub-)packages and package diagrams suitably model the dependencies among the packages. A link (i.e., a dashed line with an arrowhead) between two packages represents the dependency relationship. Figure 2.15 shows a package diagram where package `Security` contains (sub-)packages `FIM` and `Credentials`. In this case, a change in package `Credentials` may affect the de-

pendent package FIM. Also, links connecting packages in a package diagram can be decorated with additional information specifying the kinds of relationships that may exist between the involved packages (i.e., *import*, *merge*, *use*, etc.). In Figure 2.15, the link between packages FIM and Credentials specifies a usage dependency that is decorated with the *stereotype* «use» (described later).

2.6.5 Constraints

Often the essential aspects of the system are restricted using constraints. In UML, constraints are defined using *Object Constraint Language* (OCL) [7]. Typically, OCL constraints describe invariant conditions. Every OCL expression relies on the types (e.g., classes) that are defined in the UML diagrams to precisely specify systems [90]. In OCL, an *invariant* is a constraint that must be true for an object during its whole life time. For example, the invariant "a valid CoT in the FIMs must have at least one IDP and SP" can be expressed in OCL as

context Federation
inv: *self*.idps.size()>=1 and *self*.sps.size()>=1

where identifiers idps and sps are the variables (or attributes) in the class Federation. Note that we do not represent attribute of the classes in UML class diagrams. For instance, consider the class diagram of Figure 1.3 (page 12) where a single IDP and multiple SPs are associated with the Federation. In this case, the identifiers (e.g., *idp* and *sps*) followed by key word *self* in the OCL constraints will represent the attributes which are derived from such associations.

2.6.6 UML profiles

The notion of profiles describes the lightweight extension mechanism that allows one to adapt UML to a particular domain (e.g., finance, healthcare, etc.) or a platform (e.g.,

SOA) without changing the UML metamodel. A UML profile is a kind of package which may consist of stereotypes, tagged values, and a set of constraints.

In a UML profile, stereotypes give specific meanings to the use of UML model elements. Also, stereotypes may have one or more associated (optional) tagged values that provide extra information. Constraints may typically define well-formedness rules that are specific to the profile. For instance, one may restrict the way the metamodel and its constructs need to be used while using the profile.

The UML4SOA Reconfigurations Profile In Chapter 4, we will use the UML profile in [33] to model architectural and reconfigurations aspects of FIMs. The profile aims to provide the support for describing such aspects under style. It extends UML4SOA profile in [51, 92] to allow the modelling of inherent dynamic topologies of service oriented applications where components may join or leave the systems and connections between the components are rearranged. In order to avoid ill-formed configurations, a suitable mechanism is required in the modelling approaches to constraint allowable configurations. To express such conditions, architectural style provides a mechanism where a set of rules specify such configurations.

As observed in [33], UML provides limited support for describing architectural styles. The profile in [33] provides the additional support in UML to model styles. In addition to this, it also provides a methodology that can be used to model dynamic changes in configurations under style. To this purpose, the profile uses UML diagrams which may conveniently be manipulated by UML designers to represent style and reconfigurations. In this profile, SA components (e.g., a service provider) are represented via UML components[3] with ports attached. UML connectors model interactions (e.g., service references) between the components by attaching ports.

Modelling Architectural Style The profile in [33] uses two different kinds of struc-

[3]For simplicity we do not decorate UML components with the stereotype «component»

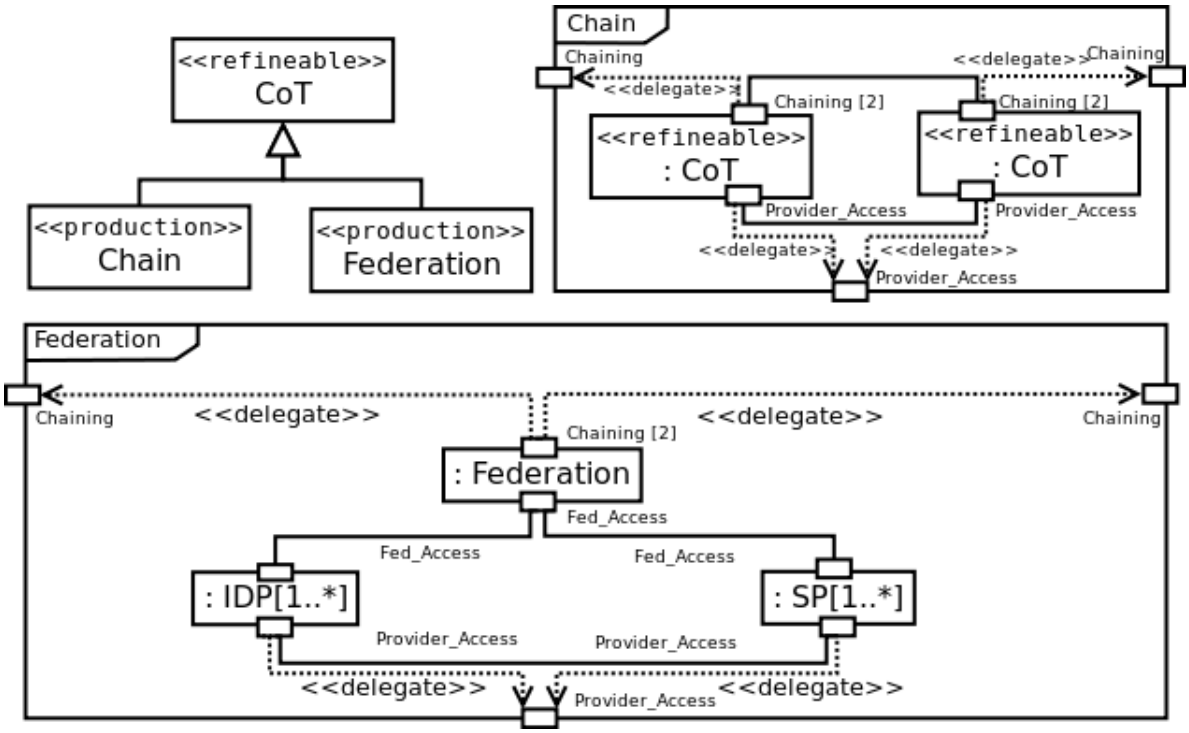


Figure 2.16: Describing an architectural style using the profile

tural diagrams (see Figure 2.16) namely, class diagrams and type level composite structure diagrams to represent style. More precisely, the profile allows one to describe abstract architectural components which are stereotyped as «`refineable`». Such components can be replaced with a given configuration by applying one of the productions (or design rules) defined within the style. The components stereotyped as «`production`» represent such design rules. The internal structure of these components is described in their corresponding composite structure diagrams. In this way, «`production`» components define the composeable patterns in the style. As a result, styles are defined in an inductive manner with composeable patterns.

In this profile, a class diagram is used to represent «`refineable`» components and their associated «`production`» components whose corresponding structure diagrams (type level) model their internal structure. Figure 2.16 describes an architectural style which will be described later in the corresponding chapter. We use this diagram to

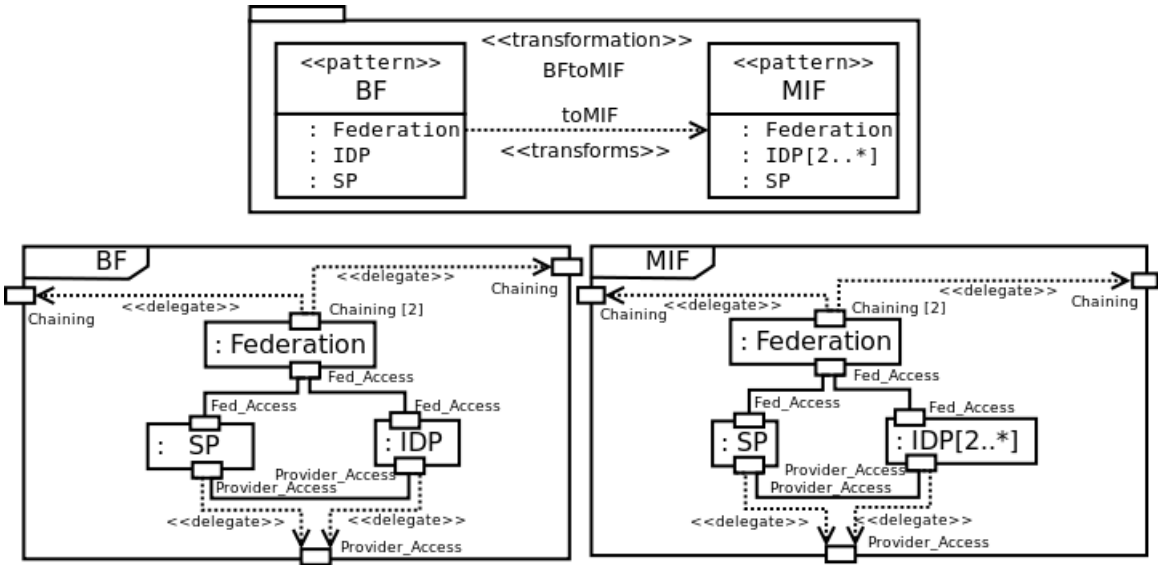


Figure 2.17: Describing a reconfiguration rule using the profile

briefly highlight the concepts in the profile. For instance, consider the class diagram (on the top-left) in Figure 2.16 where component `CoT` is stereotyped as «`refineable`» and it models an abstract architectural component. Such a component is amenable to be replaced using one of the components stereotyped as «`production`» whose corresponding structure diagrams (on the top-right and the bottom) in Figure 2.16 describe the internal structures of the CoT. In this way, the legal connections between the components are defined using the «`production`» components. Moreover, the «`production`» components can be considered as basic building blocks of the style where a set of these components actually determine the style.

While using the profile in [33], a valid configuration can be produced by replacing «`refineable`» components with the composeable patterns given by the «`production`» components. Such a process of refining abstract components can be achieved by applying the desired production patterns of the style. Furthermore, configurations created using such a mechanism can be analysed with the help of composeable patterns. Also, one may determine whether they adhere to the style described by the applied productions.

Modelling reconfigurations The profile uses package diagrams to model reconfigurations. In a style, each such diagram represents a reconfiguration rule and consists of two (the left hand side and the right hand side) components. The package diagram is stereotyped as «transformation» and the two packaged components are stereotyped as «pattern». These components in the package diagram are linked via an edge stereotyped as «transforms» and the corresponding structure diagrams describe their internal structure. Figure 2.17 describes a simple reconfigurations rule BFtoMIF which adds one or more components of type IDP in the configuration of the left hand side «pattern» BF. We will describe similar rules in Chapter 4 to deal with style-preserving changes in FIM systems.

To preserve the style while allowing reconfigurations, the profile requires both the left hand side and right side components should be having the same type. However, the profile does not explicitly specify typing of these components in the rule. Observe that both of these components in rule BFtoMIF represent configuration described by the production Federation given in Figure 2.16. In other words, these components in the rule share the same number and type of properties which are common to the production Federation with the change introduced in the cardinality of the properties of the right hand side «pattern» MIF. Moreover, a rule may have an associated precondition attached to the transformation edge, which points to a package containing such a condition. Note that we do not model preconditions of the rules in this dissertation.

Chapter 3

Modelling FIMs in UML

This chapter describes a semi-formal model in UML to represent structural and reconfiguration aspects of FIMs. Specifically, we model the FIM patterns namely, **BF**, **MIF**, **MSF**, **AF**, and chain of CoTs that are described in Section 2.2. In this chapter, we use UML in a way that allows us to represent FIMs at the desired level of detail. For instance, we discard attributes of classes that represent the FIM components. Since it is out of scope of this thesis to model behaviour of the FIMs, we also do not consider interfaces of the ports attached to the FIM components.

3.1 Identifying Constraints of FIM Systems

In this section, we revisit the main concepts in FIM systems and their relationships for the purpose of identifying the constraints for valid configurations of interest (cf. Section 2.1 and Section 2.2). We identify the constraints over those relationships in order to restrict the valid models of FIM systems, that is those that conform to the FIM patterns. Table 3.1 shows multiplicities of the FIM components in those patterns. In this table, a dash (symbol -) shows that the multiplicity of the component is irrelevant to the pattern. In addition to these multiplicity constraints, the structural constraints

FIM pattern	Multiplicities of FIM components		
	IDP	SP	CoT
BF	1	1	-
MIF	2..*	1	-
MSF	1	2..*	-
AF	2..*	2..*	-
Chain of CoTs	-	-	2..*

Table 3.1: Multiplicities of the components in the FIM patterns.

listed below further describe architectural composition of a CoT in UML:

***GC0**: a unique instance of a port is attached to a single component.*

***GC1**: a `CoT` may consist of either a `Federation` together with its associated `IDP`s and `SP`s, or `CoT`s attached to each other in a chain.*

***GC2**: each instance of type `IDP` and `SP` associated to a `Federation` should be within the `CoT`.*

***GC3**: each instance of type `IDP` and `SP` should be associated to the rest of instances of type `IDP` and `SP` associated to the `Federation`.*

***GC4**: all instances of type Dynamic Trust Management System (`DTMS`) attached to `IDP`s and `SP`s should be within the `CoT`. The purpose of a `DTMS` is to manage the dynamic security and trust requirements of an `IDP` and an `SP` in a **CoT** where `IDP`s and `SP`s can potentially join or leave the CoT at run time.*

***GC5**: a unique instance of type `DTMS` is attached to a single `IDP` (and `SP`).*

***GC6**: each instance of type `IDP` and `SP` should be associated to a `Federation`.*

Constraints **GC3** and **GC6** are obtained from the FIM patterns described in Section 2.2. Also, constraint **GC2** is a variation of constraint **GC6**. On the other hand, constraints **GC0**, **GC1**, **GC4**, and **GC5** are introduced in this dissertation. Furthermore, constraints **GC0**, **GC1**, **GC2**, **GC3**, **GC4**, and **GC5** will be defined over the UML model described in Section 3.2 while constraints **GC3** and **GC6** will be defined over the UML model described in Section 4.1. Notice that constraint **GC3** is common to both those UML models. Now, we provide the comments on those constraints:

To restrict the use of ports in UML model described in Section 3.2, we describe a general condition which specifies that a unique instance of a port should be referenced by a single component. This will enable designers to avoid mess up (or visual clutter) in the UML diagrams. Constraint **GC0** describes such a condition which will be applied to every port type described in the UML model. Note that the multiplicity constraints over the associations between the components and their ports further restrict the valid models.

In this dissertation, we intend to model two architectural views of a CoT where one describes the CoT as a federation of providers (i.e., FIM patterns **BF**, **MIF**, **MSF**, and **AF**) while the other describes the CoT as a chain of CoTs. To this purpose, constraint **GC1** describes a mutually exclusive condition which specifies that the CoT can either consist of a federation of providers or a chain of CoTs.

Since a CoT may consist of a federation together with its associated IDPs and SPs, constraint **GC2** describes such a condition which specifies that the IDPs and SPs attached to a federation should also be attached to the enclosing CoT.

In a FIM system, the providers (i.e., IDPs and SPs) associated to a federation may interact with each other. In this case, constraints **GC3** describes a condition which specifies that all providers associated to a common federation should be connected to each other. More precisely, the condition specifies that each IDP and SP should be attached to the rest of IDPs and SPs attached to the common federation.

In this dissertation, we introduce a component which models a dynamic trust management system (DTMS). In the FIM system, each IDP and SP may have a separate DTMS attached in order to manage its security and trust requirements. Note that it is out of scope of this dissertation to give details of this component. Constraint **GC4** describes a condition which specifies that all DTMSs associated to the providers in the federation should also be associated to the enclosing CoT.

Constraint **GC5** describes a condition which is similar to the one described by constraint **GC0**. Since each IDP and SP in a federation has a separate DTMS attached, constraint **GC5** describes such a condition which specifies that a unique instance of DTMS should be attached to a single provider of type IDP or SP.

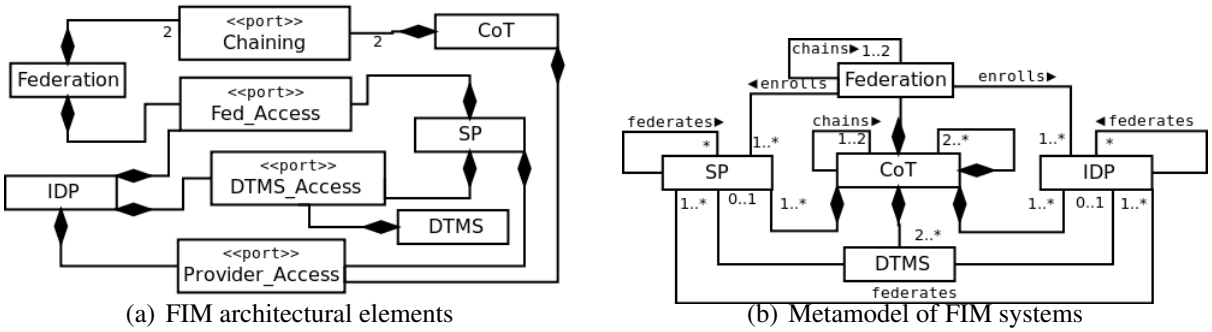
The UML model described in Section 4.1 does not require the use of constraints to restrict the associations between the FIM components and their enclosing CoT. Therefore, constraints **GC1**, **GC2**, **GC4**, and **GC5** are irrelevant to this model. However, this model still requires the use of constraints **GC3** and **GC6** which restrict the associations that relate components enclosed in the CoT. Notice that constraint **GC6** is quite similar to constraint **GC2** with the difference that the former eliminates CoT.

3.2 Architectural Style of FIMs

We model structural aspects of FIM patterns described in Section 2.2 by describing their style. We consider the UML "as-is" approach where the standard UML notations are used to model such aspects.

3.2.1 Vocabulary of the style

The class diagram in Figure 3.1(a) models FIM components, their ports, and their associations. The components of FIM (cf. Section 2.2) are captured by the classes `CoT`, `Federation`, `SP`, `IDP`, and `DTMS`. (The latter class deals with dynamic secu-



(a) FIM architectural elements (b) Metamodel of FIM systems

Figure 3.1: A logical model of the FIMs

rity and trust management and it is not relevant in this dissertation.) The stereotype «port» singles out classes representing ports, namely `Chaining`, `Fed_Access`, `DTMS_Access`, and `Provider_Access`. For instance, an `IDP` may interact with `SP`s using a `Provider_Access` port. Likewise, `IDP`s and `SP`s communicate with a `Federation` through `Fed_Access` ports. Also, a `Federation` and a `CoT` may respectively interact with other `Federation`s and `CoT`s using `Chaining` ports.

3.2.2 Constraints of the style

We remark that SA components have a composition association with the attached ports and the constraint (constraints are repeated for the readers convenience)

***GC0**: a unique instance of a port is attached to a single component.*

is needed on Figure 3.1(a). Notice that this constraint is defined over the ports in a general way. However, it can be applied to any of the ports including `Chaining`, `Fed_Access`, `DTMS_Access`, and `Provider_Access`. For instance, to restrict the port of type `Chaining` OCL constraint

context Chainning
inv: *self*.cot->size()=1 `xor` *self*.federation->size()=1

defines a condition which describes that an instance of port type `Chaining` can have a reference to either a components of type `CoT` or a component of type `Federation`.

In other words, a `Chaining` port can be attached to either a `CoT` or a `Federation`. Similarly, we define below a separate OCL constraint for ports of types `Fed_Access`, `DTMS_Access`, and `Provider_Access`:

context Federation_Access
inv: *self*.idp->size()=1 `xor` *self*.sp->size()=1 `xor` *self*.federation->size()=1

context DTMS_Access
inv: *self*.idp->size()=1 `xor` *self*.sp->size()=1 `xor` *self*.dtms->size()=1

context Provider_Access
inv: *self*.idp->size()=1 `xor` *self*.sp->size()=1 `xor` *self*.cot->size()=1

Also, associations may partially define constraints of FIM styles like in Figure 3.1(b) which describes[1] two different architectural views of a CoT. More precisely, a view represents a CoT that models a federation of providers by considering FIM patterns **BF**, **MIF**, **MSF**, and **AF** while the other describes a CoT as a chain of CoTs. Note that, it is difficult to clearly describe the constraint between two mutually exclusive sets of associations for modelling such a CoT in Figure 3.1(b). This can be accommodated by the constraint

> ***GC1****: a* `CoT` *may consist of either a* `Federation` *together with its associated* `IDPs` *and* `SPs`*, or* `CoTs` *attached to each other in a chain.*

whose OCL representation is given below:

context CoT
inv: *self*.cots->size()>=2 `xor` (*self*.federation->size()=1 `and`
self.dtmss->size() = *self*.idps->size() + *self*.sps->size() `and`
self.idps->size()>=1 `and` *self*.sps->size()>=1)

In Figure 3.1(b), a `Federation` is associated with at least one `IDP` and `SP`, and several DTMSs. An `IDP` is associated to one `DTMS`, zero or more other `IDP`s (and at

[1]For simplicity, ports are not represented in Figure 3.1(b).

Table 3.2: OCL constraints

Constraint	OCL representation
GC2	**context** CoT **inv:** *self*.idps->includesAll(self.federation.idps->asSet()) `and` *self*.sps->includesAll(*self*.federation.sps->asSet())
GC3	**context** IDP **inv:** s*elf*.idps->includesAll(*self*.federation.idps->asSet(excludes(*self*))) `and` *self*.sps->includesAll(*self*.federation.sps->asSet()) **context** SP **inv:** *self*.idps->includesAll(*self*.federation.idps->asSet()) `and` *self*.sps->includesAll(*self*.federation.sps->asSet(excludes(*self*)))
GC4	**context** CoT **inv:** *self*.dtms->includesAll(*self*.idps.dtms->asSet()) `and` *self*.dtms->includesAll(*self*.sps.dtms->asSet())
GC5	**context** DTMS **inv:** *self*.idp->size()=1 `xor` *self*.sp->size()=1

least one `SP`). Similarly, an `SP` is associated to one `DTMS`, zero or more other `SP`s (and at least one `IDP`). Besides those multiplicity constraints common to FIMs, one may need to specify the constraints given below to restrict valid models of interest:

***GC2**: each instance of type `IDP` and `SP` associated to a `Federation` should be within the `CoT`.*

***GC3**: each instance of type `IDP` and `SP` should be associated to the rest of instances of type `IDP` and `SP` associated to the `Federation`.*

***GC4**: all instances of type `DTMS` attached to `IDP`s and `SP`s should be within the `CoT`.*

***GC5**: a unique instance of type `DTMS` is attached to a single `IDP` (and `SP`).*

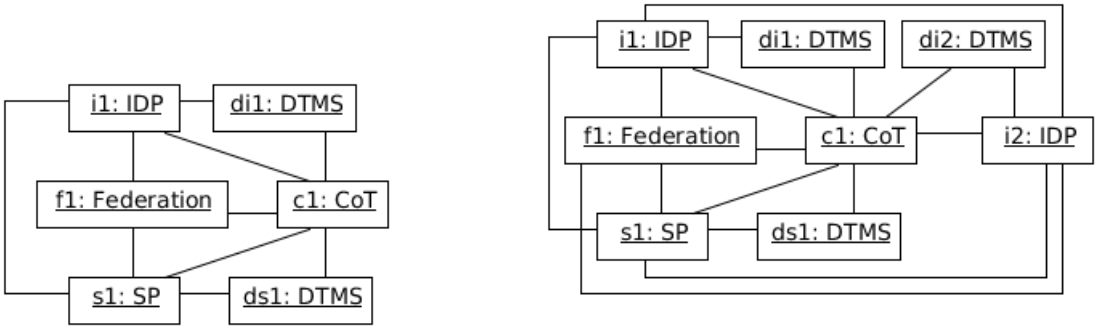
In Table 3.2, OCL constraints are defined corresponding to the constraints which are given above.

Finally, observe that Figure 3.1(b) uses two different kinds of associations over a `CoT`. In fact, a composition association is used to describe that a `CoT` may consist of

two or more CoTs while a (self-)association chains is used to describe that CoTs are attached to each other within the CoT to form a chain. (A similar association is defined over a Federation.) As a result, Figure 3.1(b) yields a model that enables one to create a complex CoT.

3.3 Generating Configurations of FIMs

In this section, we discuss how FIM configurations can be obtained using UML "as-is" approach. We create a few FIM configurations using object diagrams where objects model components and links between objects specify potential communication capabilities of components. For simplicity, ports of components are not considered in these diagrams. Figure 3.2 represents a few FIM configurations of the model given in Sec-



(a) **BF** configuration (b) **MIF** configuration

Figure 3.2: Object diagrams showing a few FIM configurations

tion 3.2. Specifically, Figure 3.2(a) shows a configuration of pattern **BF** where a CoT consists of a Federation having a single IDP and a single SP attached together with their DTMSs. Similarly, Figure 3.2(b) describes a configuration of pattern **MIF**.

3.4 Modelling Reconfigurations of FIMs

The UML "as-is" approach does not provide any mechanism to describe architectural changes. Consequently, one has to operate changes in actual configurations.

Example 3.1 *The diagram in Figure 3.2(a) of pattern **BF** can be reconfigured into the one in Figure 3.2(b) of pattern **MIF** introducing an* `IDP i2` *and its associated* `DTMS di2`*.*

We remark that this may raise consistency problems. Also, the configuration produced by such reconfigurations may violate the style.

3.5 Evaluating the FIMs model

We evaluate the FIMs model given in this chapter with respect to describing architectural and reconfiguration aspects. Note that these aspects and their relevance to FIMs will be detailed in Section 6.1 (page 94).

Figure 3.1 (page 58) represents two different but related class diagrams and they model the same type of FIM components. Class diagram in Figure 3.1(a) models FIM components and their associated ports. This diagram provides typing of the ports when we use structure diagrams (instance level) to describe configurations. Class diagram in Figure 3.1(b) models FIM components and their possible relationships. This diagram allows us to generate FIM configurations (e.g., via object diagrams) without explicitly considering the interaction elements (i.e., ports). For instance, consider object diagrams to represent FIM configurations. Indeed, objects may potentially represent component and port instances (respectively typed over the classes in Figure 3.1(a) and in Figure 3.1(b)) in the configurations. But it would be very difficult to make visual distinction between components and ports in a given configuration while using the same notation to represent both those kinds of elements. Therefore, for simplicity, we

use object diagrams (cf. Figure 3.2 on page 61) to describe FIM configurations where components are represented without ports. Notice that we do not distinguish FIM components of the same type in the class diagrams shown in Figure 3.1 (page 58). For instance, a CoT in Figure 3.1(a) and Figure 3.1(b) describes the same FIM component. Consequently, we generate FIM configurations (i.e., via structure diagrams) according to these class diagrams where components may have ports attached.

In Figure 3.1(b), we use composition associations between the CoT and its associated FIM components. This is due to the fact that, from a security point of view, if a CoT is destroyed (or made non-functional) then the involved providers need to stop issuing (or consuming) the authentication related information of the users. In UML, such an architectural composition may require the use of global OCL constraints which typically makes the UML models cumbersome to use. In this way, we define a few global constraints for the architectural composition of the CoT. To some extent, such constraints could be avoided by using the simple UML association between the CoT and its associated components. However, this would result into having a flat view of the CoT and loosing hierarchical structure over a complex FIMs.

Moreover, the two architectural views (i.e., a federation of providers and a chain of CoTs) of the CoT in Figure 3.1(b) (page 58) are loosely defined with the help of an OCL constraint whose textual description is given by **GC1** (page 59). Since it is difficult to clearly specify such a condition (i.e., via the two mutually exclusive sets of associations) within the diagram, we specify the constraint in such a way that allows us to conveniently represent the two architectural views of the CoT. In UML, one may use inheritance to simplify these architectural views of the CoT. For instance, two sub-classes of the CoT can be created where each sub-class models one of these views. However, such an approach not only introduces two additional (or artificial) FIM components but still it requires the complex association ”chains” over Federation Figure 3.1(b) together with the OCL constraint need to be described so as to allow

federations can be connected in a chain of CoTs.

This model is developed by following a simple strategy where existing UML notations are used to represent structural aspects of FIMs. As a result, a UML designer can immediately understand the models and can use existing tools to manipulate them. In this way, one may conveniently apply changes to the simple configurations described by either object diagrams or structured diagrams (at instance level) while respecting the underlying style. However, we argue that manipulating a complex configuration in UML can be problematic.

To illustrate this, consider Example 2.7 (page 29) which describes a rather complex scenario involving many IDPs (i.e., school districts) and SPs (i.e., regional information centres). While applying this UML model to such a scenario, designers use rules of thumb to generate the required FIM configurations for the scenario.

Due to security critical nature of FIMs and its patterns, designers will heavily rely on OCL compliant UML tools to perform style checking of FIM configurations. However, we argue that such an approach does not apply well to the situations where designer may find it very difficult to deal with subsequent changes in a large complex FIM configuration. For instance, a large school district in Example 2.7 is divided into multiple school districts (i.e., IDPs). Similarly, such a change may apply to the regional information centres (i.e., SPs) in Example 2.7. In this case, the designated constraints need to be checked against every single IDP and SP instance to validate the updated FIM configuration.

We consider a simple topology (i.e., chain) for a CoT in Figure 3.1(b) (page 58). In order to make changes in this diagram to support more complex topologies (e.g., tree, star, mesh, etc.) for the CoT, one may conveniently introduce new classes and their associations together with the OCL constraints. However, the obvious limitation of such an approach is that it may make the FIMs model cumbersome to use.

Chapter 4

Modelling FIMs in UML Profile

Chapter 3 describes structural aspects of FIMs. However, it lacks in modelling certain desired aspects (i.e., refinement and reconfigurations) of such systems. Therefore, we use the UML profile in [33] in this chapter to model those aspects of FIMs. In addition to this, we capture the same details of the FIM patterns as described by Chapter 3.

4.1 Architectural Style of FIMs

We model structural aspects of FIM patterns described in Section 2.2 by describing their style. We use the profile in [33] to model FIMs. In particular, the profile allows us to describe the architectural style for FIMs. To support the architectural refinement, an abstract (or refineable) architectural component is modelled which can be replaced with the complex configurations using the corresponding productions. Also, the profile is used to describe reconfigurations for FIMs.

Figure 4.1 describes the architectural style for FIMs. Notice that this figure is similar to Figure 2.16 (page 51) with the difference that it represents an additional component (i.e., `DTMS`) and a self-association over the providers (i.e., components of type `IDP` and `SP`) in production `Fed` (bottom class diagram). Such a production will be

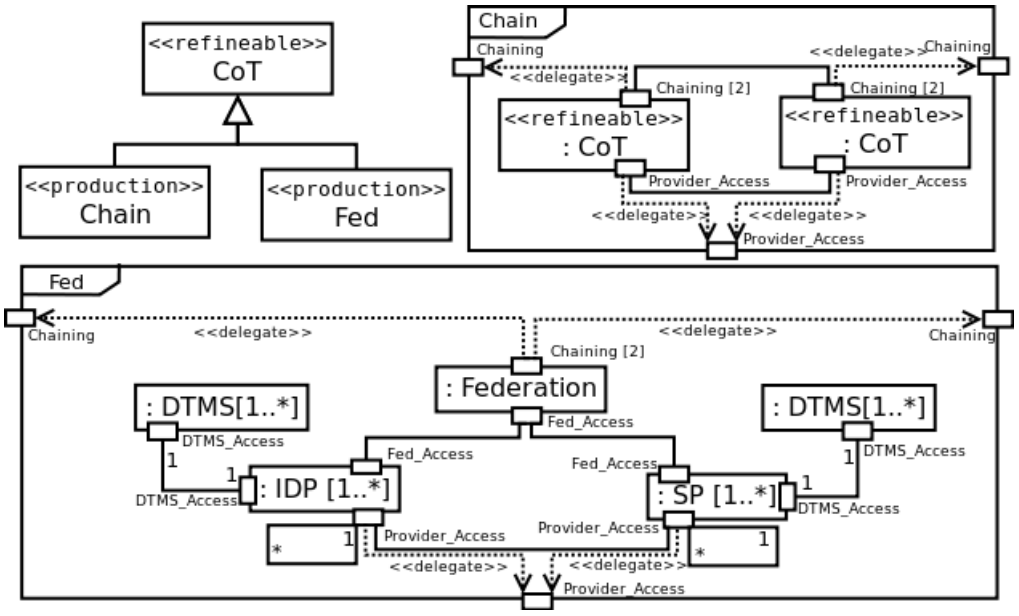


Figure 4.1: Architectural style productions for FIMs

described later in this section. Consider Figure 4.1 where the top-left class diagram has two «`production`» components `Chain` and `Fed` associated to a «`refineable`» component `CoT`. A `CoT` can be replaced by configurations corresponding to the productions `Chain` and `Fed`. A production in the profile represents a composeable pattern in the corresponding structure diagram (which gives the internal structure of the `CoT`). More precisely, a structure diagram (at type level) describes the legal connections between the components of a «`refineable`» component.

In our case, the configurations of a `CoT` can be generated using the productions `Chain` and `Fed` given in Figure 4.1. To replace a `CoT` in production `Chain` with the configuration, one may freely use

- production `Chain` to generate a complex chain of `CoT`s or
- production `Fed` to generate the other configurations.

The former production is the top-right structure diagram in Figure 4.1; it states that each `CoT` component has two `Chaining` ports and a `Provider_Access` one. The

`Chaining` ports enable `CoT`s to chain up while the latter ports connect providers. The production `Fed` is the bottom diagram in Figure 4.1; it describes a `CoT` as a federation of providers in a general way capturing FIM patterns **BF**, **MIF**, **MSF**, and **AF**.

As described earlier, a legal FIMs configuration is subject to some constraints. This is rendered by decorating (parts of) structure diagrams with multiplicities. For instance, in production `Fed` of Figure 4.1, a `Federation` is attached to at least one `IDP` and one `SP` through a `Fed_Access` port and each provider uses a different `DTMS` via a `DTMS_Access` port. Each `IDP` connects to a (possibly empty) set of `IDP`s through a `Provider_Access` port by a connector rendering a (self-)association on `IDP`. Similarly, each `SP` is attached to zero or more `SP`s. Each `IDP` is attached to at least one `SP` and each `SP` is attached to at least one `IDP` through `Provider_Access` ports.

We remark that, for FIM architectures, it is not always convenient to infer the multiplicity of connectors from the multiplicity of the types of instances (as typical in UML designs). The reason being that the same structure diagram is used to specify constraints pertaining to different architectural levels. In fact, the multiplicities on connectors refer to instances of actual configurations while those in classes pertain to architectural constraints of enclosing classes.

Example 4.1 *Production `Fed` in Figure 4.1 imposes a 1-to-1 association between `DTMS`s and `IDP`s components. Accordingly, `CoT`s may be composed by many `DTMS`s and many `IDP`s.*

In Example 4.1 the constraints of `DTMS` and `IDP` refer to the architectural aspects of the enclosing class `CoT`, while the connector between the two classes specifies that there is a 1-to-1 association among `DTMS` and `IDP` components in a legal configuration. Notice that this implies that UML forces the designer to explicitly consider the multiplicities of connectors as they may 'conflict' with those of the classes when the associations need special conditions (e.g., injectivity).

An advantage of structure diagrams to describe the style is that they may reduce the use of OCL constraints. In fact, it may not be required to give OCL constraints over composition associations (e.g., **GC4** on page 60) and the associations between a structured classifier (e.g., `CoT`) and its parts (e.g., `DTMS`, `SP`, `Federation`, and `IDP`) may be implicitly represented. OCL constraints may still be required on other associations; for instance, the structure diagram `Fed` in Figure 4.1 requires **GC3** (page 60) and the additional constraint

***GC6**: each instance of type `IDP` and `SP` should be associated to a `Federation`.*

that is very similar to **GC2** (page 60) but eliminates `CoT` and its OCL representation is given below:

context IDP
inv: *self*.federation->size()=1

context SP
inv: *self*.federation->size()=1

4.2 Generating Configurations of FIMs

We discuss, the profile-based approach, how FIM configurations can be obtained.

Object diagrams are typically used to model simple scenarios (e.g., pattern **BF**) and are less suitable for complex configurations (e.g., chain of CoTs). To model such a configuration, we use a structure diagram (at instance level) that may impose a hierarchy over a complex system. These diagrams are essentially object diagrams for the classes (e.g., CoT) with internal structure [75].

The structure diagram in Figure 4.2 (names are omitted for readability) represents a configuration of a chain of two CoTs of pattern **BF** generated by using the corresponding productions in Figure 4.1 (page 66). In fact, a «`refineable`» component `CoT` is

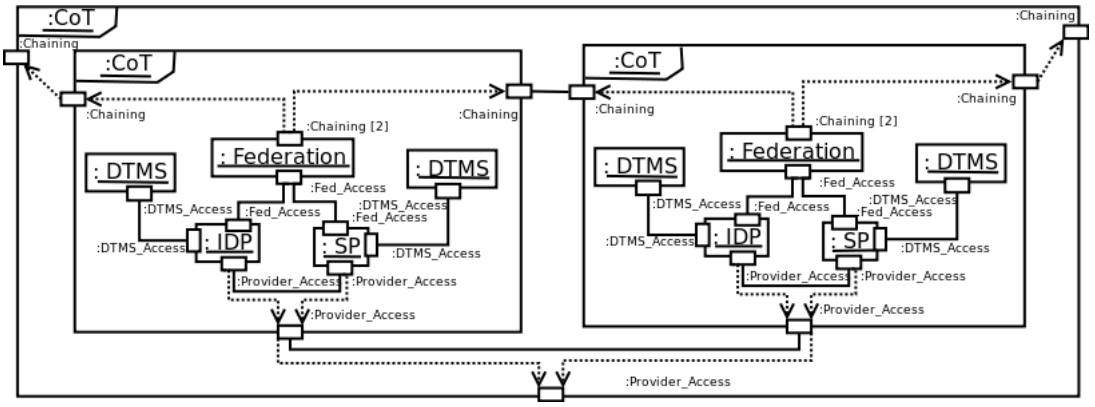


Figure 4.2: Structure diagram (instance level) showing a configuration

replaced with two `CoT`s attached to each other in a chain. Each `CoT` in this chain is further refined with configuration of pattern **BF** by applying the production `Fed`. Observe that this refinement allows us to deal with the design (at type level) of a complex system (e.g., chain of CoTs). However, designers have to use rules of thumb to generate UML configurations (at instance level) of the FIM patterns.

4.3 Modelling Reconfigurations of FIMs

The profile-based approach allows one to describe architectural changes at the level of design by using structure diagrams (at type-level). The profile uses a package stereotyped as «`transformation`» linked with a «`transforms`» edge. The scope of the named properties in «`pattern`» components spans over the whole package.

Example 4.2 *The package in Figure 4.3, consists of two «`pattern`» components*[1]*; the properties of the left component are of type `Federation`, `IDP`, `SP`, and `DTMS`. Such properties are common to all the components of the rules.*

Each component has an associated structure diagram modelling its internal structure. To model reconfiguration rules, the «`pattern`» components represent architectural

[1]The stereotype «`pattern`» does not represent FIM pattern.

configurations according to the production `Fed` in Figure 4.1 (page 66). Also, structure diagrams associated to these components are created according to this production. In this way, the «`pattern`» components in the rules can be considered as valid architectural instances (i.e., configurations of patterns **BF**, **MIF**, **MSF**, and **AF**) of production `Fed`.

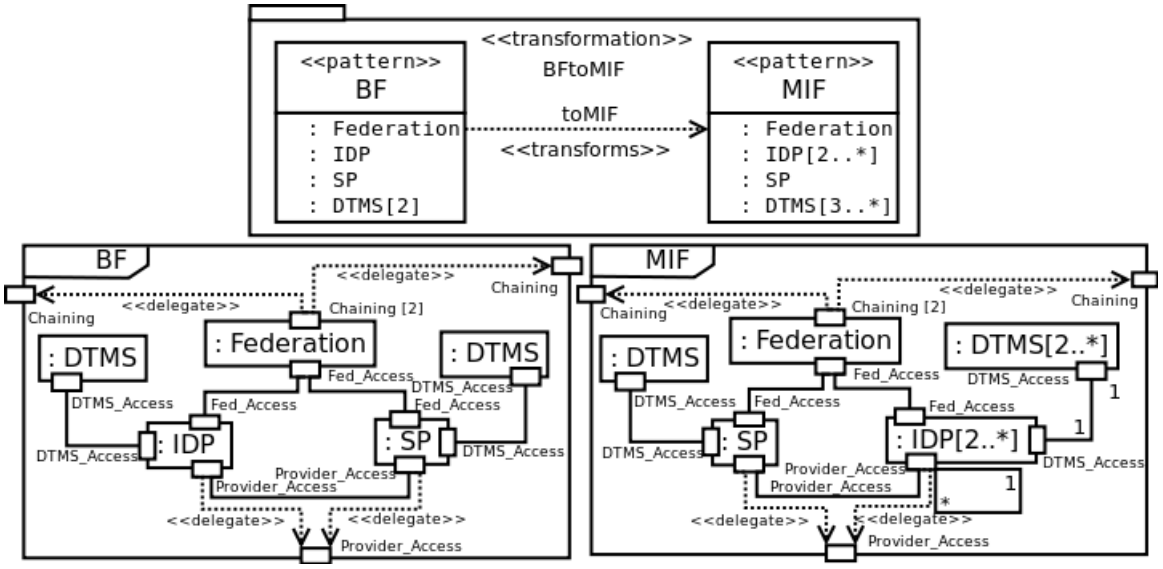


Figure 4.3: Reconfiguring pattern **BF** to pattern **MIF**

Figure 4.3 describes the reconfiguration rule `BFtoMIF` that reconfigures FIM pattern **BF** (left) into FIM pattern **MIF** (right). There, the component `BF` represents federation with one `IDP`, one `SP`, and two `DTMS`s. The component `MIF` represents federation obtained by adding one or more `IDP`s and their associated `DTMS`s to the `BF` federation. Note that the internal structures of `BF` and `MIF` differ only for the associations on `IDP`s and `DTMS`s. Indeed, the reconfiguration introduces a new (self-)associations relating `IDP`s and updates the multiplicity of the association between `IDP`s and their `DTMS`s. Correspondingly, the rule updates the ranges of the properties `IDP` and `DTMS`. Other reconfiguration rules of FIM patterns can be defined in a simple way; we illustrate a few more reconfiguration rules in the profile-based approach.

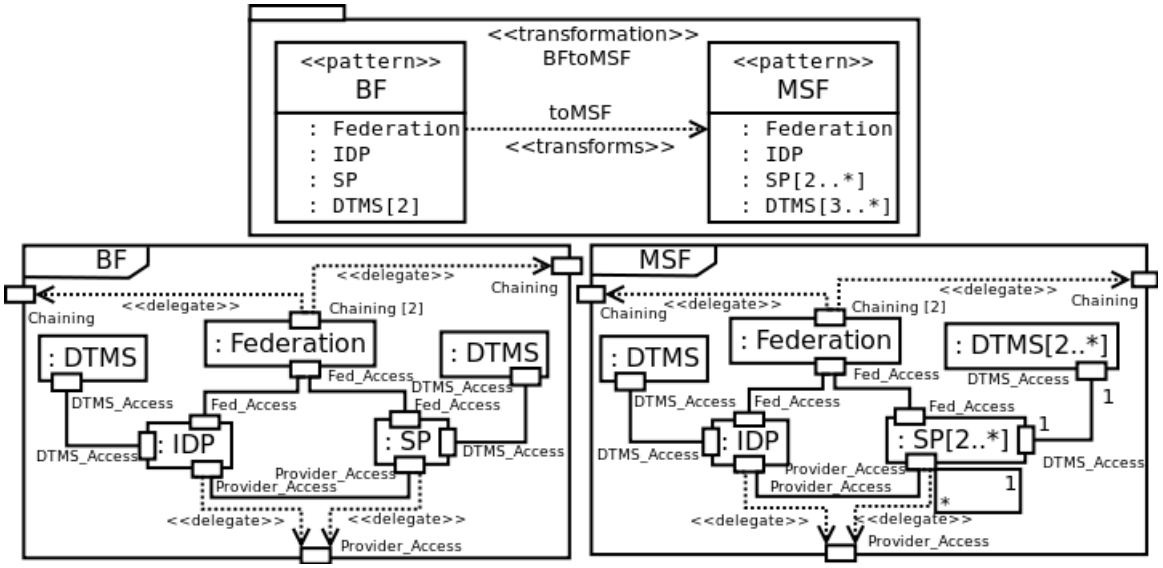


Figure 4.4: Reconfiguring pattern **BF** to pattern **MSF**

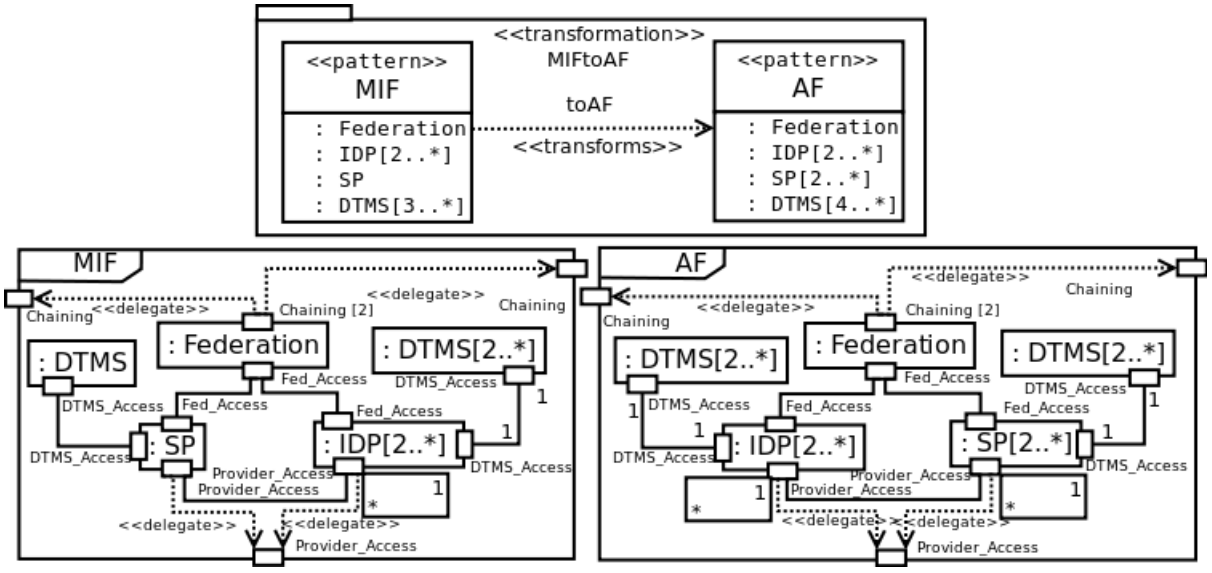


Figure 4.5: Reconfiguring pattern **MIF** to pattern **AF**

A rule can be defined for reconfiguring FIM pattern **BF** into FIM pattern **MSF** by introducing one or more `SP`s and their associated `DTMS`s in the new configuration. Rule `BFtoMSF` in Figure 4.4 describes such a change in FIM pattern **BF**.

Rule `MIFtoAF` in Figure 4.5 reconfigures FIM pattern **MIF** into FIM pattern **AF** by introducing one or more `SP`s and their associated `DTMS` into the new configuration.

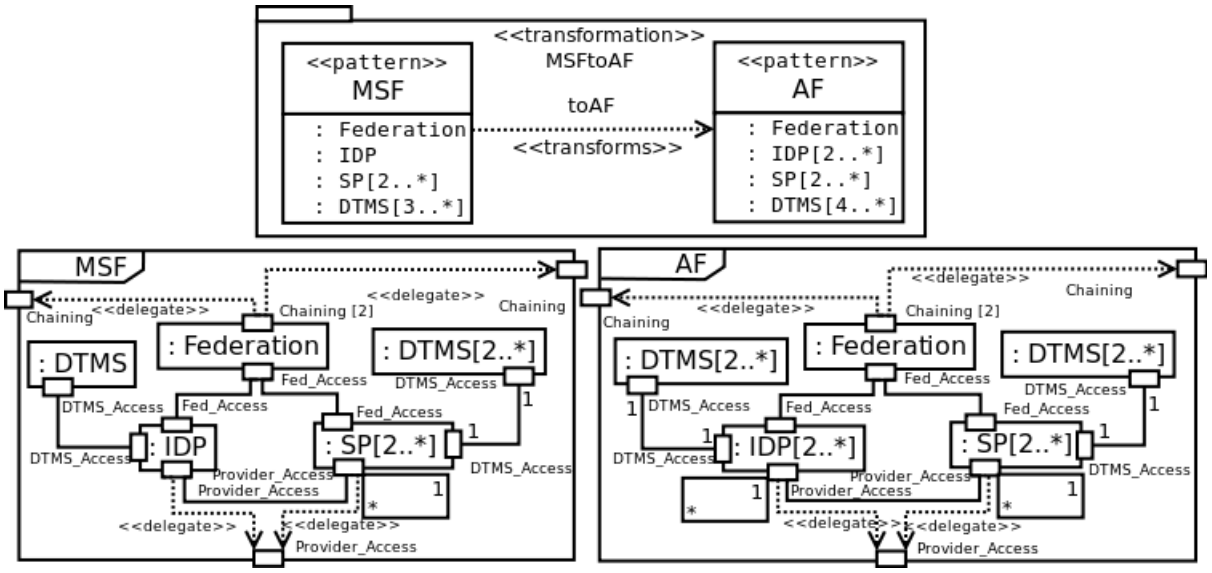


Figure 4.6: Reconfiguring pattern **MSF** to pattern **AF**

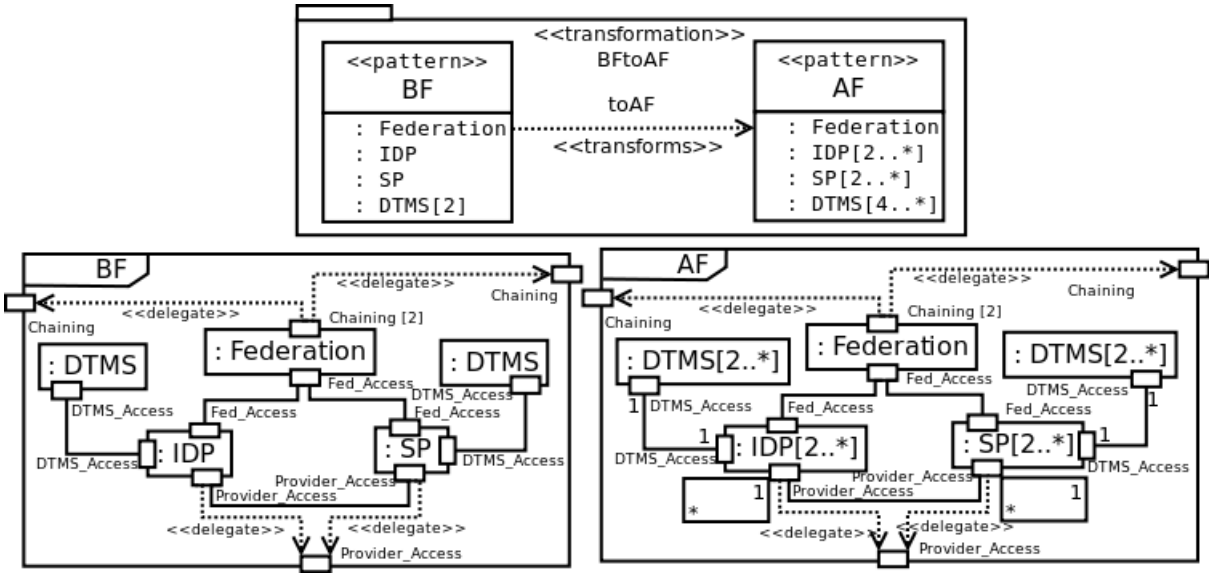


Figure 4.7: Reconfiguring pattern **BF** to pattern **AF**

Figure 4.6 describes the reconfiguration rule `MSFtoAF` that adds one or more `IDPs` and their associated `DTMS` in FIM pattern **MSF** to reconfigure it into FIM pattern **AF**.

Figure 4.7 describes the rule `BFtoAF` that adds; one or more `IDPs` and `SPs`, and two or more `DTMSs` attached to the providers. In this way, the rule `BFtoAF` changes FIM pattern **BF** into FIM pattern **AF**.

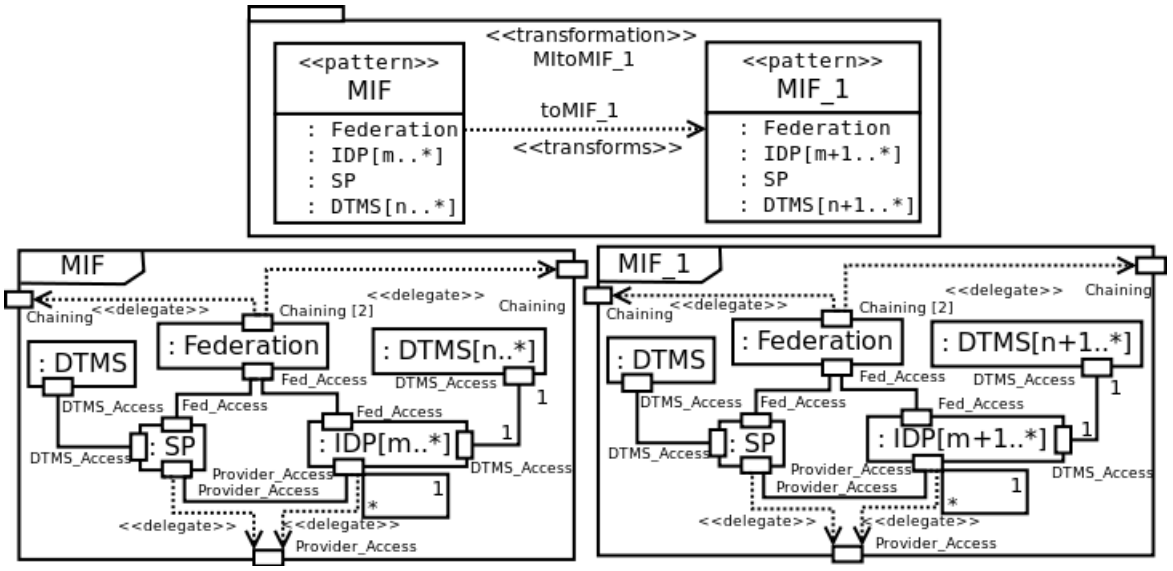


Figure 4.8: Reconfiguring pattern **MIF**

The advantage of this approach is that style preserving reconfiguration can be simply obtained by imposing that the components in the rules are of the same type. In [33] it is not clearly specified how to provide typing of those components; if we interpret the «`refineable`» component `CoT` as a common type for the «`pattern`» components, then we have a style-preserving reconfigurations for FIMs.

We remark that the profile-based approach does not satisfactorily supports reconfigurations within a same pattern like the ones that add `IDP` into configurations of FIM pattern **MIF**. To illustrate this, we discuss the rule `MIFtoMIF_1` in Figure 4.8. Such rule adds one or more `IDPs` in FIM pattern **MIF** to obtain a configuration `MIF_1` (right).

Technically, the reconfiguration is obtained by increasing the lower bounds of multiplicities of the corresponding properties. Since UML uses OCL to specify constraints, lower and upper bounds for the multiplicities may be specified by (side-effect free constant) expressions (see [80, page 112]). In Figure 4.8, OCL expressions specify the lower bounds of the properties `IDP` and `DTMS` belonging to the «`pattern`» components. These expressions may potentially be evaluated by UML tools. In a parametrised context, before evaluating these expressions the tools assign actual integer values to the

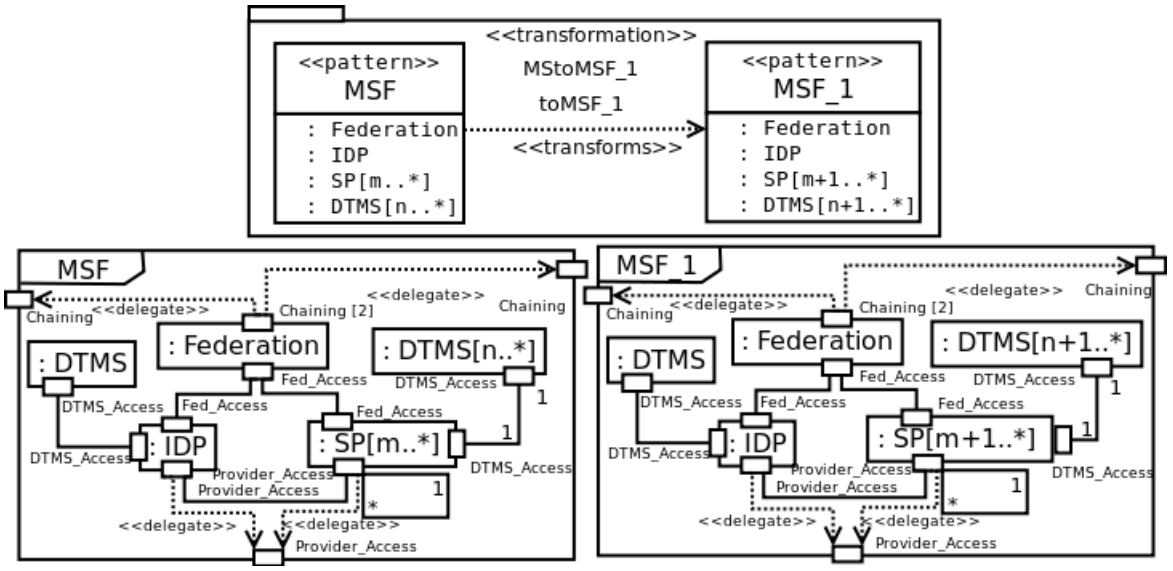


Figure 4.9: Reconfiguring pattern **MSF**

parameters (i.e., initially $m = 2$ and $n = 3$ for `MIFtoMIF_1`). While relying on such a mechanism, designers have to consistently update the actual values of the parameters whenever configurations of the FIM patterns are changed. Otherwise, the rule cannot be applied if the multiplicities may become inconsistent.

Similarly, rule `MSFtoMSF_1` in Figure 4.9 adds one or more `SP`s in FIM pattern **MSF**. The left-hand-side `MSF` «`pattern`» component represents **MSF** configurations that are transformed into the ones on the right-hand-side (`MSF_1`) by introducing one or more `SP`s (and corresponding `DTMS`s).

4.4 Evaluating the FIMs model

Due to the limitations of UML ”as-is” approach, the FIM model developed using this approach does not address specific architectural issues, namely refinement and reconfigurations. Consequently, we use the UML profile in [33] to address these issues in particular. Apart from addressing these issues, the purpose of this model is to capture the same structural aspects of FIMs as described by the FIMs model developed using

UML ”as-is” approach.

Consider Figure 4.1 (page 66). The productions `Chain` and `Fed` via their corresponding structured diagrams clearly represent the two separate architectural views of an abstract CoT. As a result, designers may conveniently use these diagrams to generate FIM configurations. For instance, the production `Chain` supports successive refinement of an abstract CoT. This production can recursively be applied to generate complex FIM configurations. Similarly, the production `Fed` can effectively be used to generate simple FIM configurations (e.g., FIM pattern **BF**). However, the designer may face the challenges (e.g., style checking) that are very similar to the UML ”as-is” FIM model while generating a complex FIM configuration (e.g., FIM pattern **AF**).

To support the changes in the style, the model in Figure 4.1 (page 66) is flexible to adopt more productions. For instance, a production can be described to support a tree like structure of CoTs in a FIM system. In this case, one need to introduce a production and its corresponding structured diagram where one may use the existing classes and/or introduce new classes together with their associations. However, the limitation of this approach is that the vocabulary of the style be loosely defined and scattered across different diagrams. As a result, the productions that use common types of components may become invalid if the required changes in the updated components are not applied consistently.

In Section 4.3, we characterise FIM reconfigurations in UML according to their effects (i.e., pattern changing or pattern preserving) on the FIM patterns. In this way, five different rules are defined to support pattern changing configurations. Recall that each FIM pattern has different security needs. The benefit of such an approach to the reconfigurations is that it may allow designers to enforce the required security mechanism while applying the rules. On the other hand, reconfigurations that preserve the FIM patterns do not affect their security requirements. A rule is also defined for each of the FIM patterns (i.e., **MIF**, **MSF**, and **AF**) to deal with such a reconfiguration. While ap-

plying such a rule, designers need to consistently update both the actual configuration and the rule. Therefore, such an approach to reconfigurations may face consistency problems in the UML model.

Moreover, the UML profile in [33] allows us to describe reconfigurations for FIMs (cf. Section 4.3) at the level of design. More precisely, type level structure diagrams are used to describe FIM reconfigurations. As a result, the problem of making changes in the actual configurations (i.e., instance level structured diagrams) still needs to be addressed. Also, the run-time FIM reconfigurations (e.g., system initiated reconfigurations in a Cloud [50]) need to be modelled by considering a dynamic security and trust management system. This requires modelling of the behavioural aspects of FIMs that may have direct effect on the application of the reconfigurations (e.g., providers joining or leaving the CoT).

Chapter 5

Modelling FIMs in ADR

Since UML is considered a de facto standard for modelling various aspects of software systems, we used UML in Chapter 3 and 4 to model architectural (and reconfiguration) aspects of FIMs. However, the UML models suffer a few draw-backs that are highlighted later in Chapter 6. In this chapter, a formal model of FIMs is designed using ADR, which is an ad-hoc ADL that features a mathematically rigorous style preserving modelling approach. Such a model captures architectural and reconfiguration aspects of FIMs. Specifically, we model FIM patterns including **BF**, **MIF**, **MSF**, **AF**, and chain of CoTs, which are described in Section 2.2. More precisely, an architectural style for FIMs is given in terms of ADR productions while the architectural changes (i.e., style-preserving) in FIMs are described using rewrite rules.

5.1 Architectural Style of FIMs

5.1.1 The type graph

The type graph yielding the vocabulary for the architectural elements of FIMs is depicted in Figure (5.1); edges yield the types of components and nodes $\circledcirc$ (federation chain), $\circ$ (federation access), and $\bullet$ (provider access) represent the kind of ports used

to connect them.

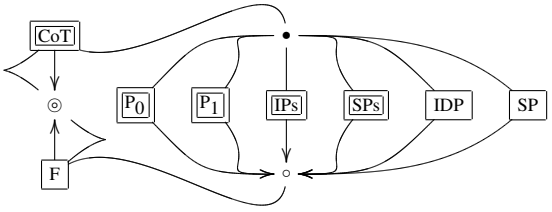
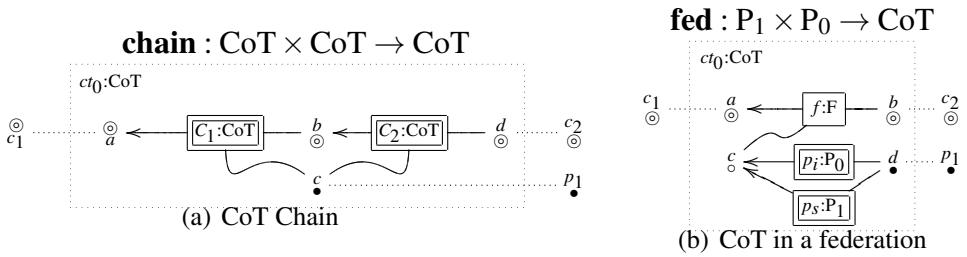


Figure 5.1: Type graph (H)

More precisely, federation chain nodes ◎ are used to form chains of edges of type F or CoT; federation access nodes ○ connect IDP and SP providers with a federation F; provider access nodes • connect providers. The vocabulary of terminal and non-terminal edges for architectural style of FIMs is given by the type graph Figure 5.1 and formally defined as

$$V_H = \{◎, ○, •\}$$
$$E_H = \{\mathrm{CoT}, \mathrm{F}, \mathrm{P}_0, \mathrm{P}_1, \mathrm{IPs}, \mathrm{SPs}, \mathrm{IDP}, \mathrm{SP}\}$$
$$t_H : \begin{cases} \mathrm{CoT} \longmapsto [◎, ◎, •] \\ \mathrm{F} \longmapsto [◎, ◎, ○] \\ \mathrm{P}_0, \mathrm{P}_1, \mathrm{IPs}, \mathrm{SPs}, \mathrm{IDP}, \mathrm{SP} \longmapsto [○, •] \end{cases}$$

The non-terminal edges shown in Figure 5.1 can be refined into complex graphs using the corresponding design productions (described later) that define legal configurations of FIMs. The non-terminal edge CoT will be refined into providers (i.e., IDP and SP) connected to each other and to their federation in FIMs. These providers can be obtained by refining non-terminal edges; for example, by refining non-terminal edges of type P_0 and IPs configurations with identity providers can be obtained, while configurations with service providers are obtained by refining non-terminal edges of type P_1 and SPs. Notice that, in this case, the role of non-terminal edges of type P_0 and P_1 is to represent a non empty set of identity providers and service providers, respectively. The terminal edges F, IDP and SP are the types for a federation, an identity provider, and a service provider respectively.



(a) CoT Chain (b) CoT in a federation

Figure 5.2: A chain of CoTs and a federation of providers

5.1.2 The productions

Productions for CoT Figure 5.2 shows productions **chain** and **fed** that respectively refine non-terminal edge CoT into a federation of providers and chain of CoTs. Notice that LHS and RHS of productions are typed over the graph depicted in Figure 5.1. The productions for CoT (See Appendix A.1 for the formal definitions) are further described as follows :

Production **chain** catenates CoTs C_1 and C_2 by connecting them on node b as illustrated in Figure 5.2(a). Also, node c is used to connect C_1 and C_2 to providers and exported together with a and d to possibly extend the chain.

Production **fed** generates configurations of CoT by connecting several providers (obtained by refining p_i and p_s) to each other and to a federation f as illustrated in Figure 5.2(b). Providers p_i and p_s interact with federation f through node c of type $\circ$; nodes c_1 and c_2 allow f to connect to other CoT and node p_1 to connect to other providers. Observe that nodes c_1, c_2, and p_1 are the nodes of the LHS of **fed** corresponding to the nodes of the RHS as specified by the dotted lines and 'exported' in the interface of **fed**; also, p_1 allows providers generated by p_s and p_i to connect to providers in other CoTs.

Productions for Identity Providers Figure 5.3 shows the productions to generate configurations of identity providers for P_0 in production **fed**. Production **pips** generates

pips : IPs $\rightarrow$ P_0 **ips** : IPs $\times$ IPs $\rightarrow$ IPs **ip** :$\rightarrow$ IPs **noip** :$\rightarrow$ IPs

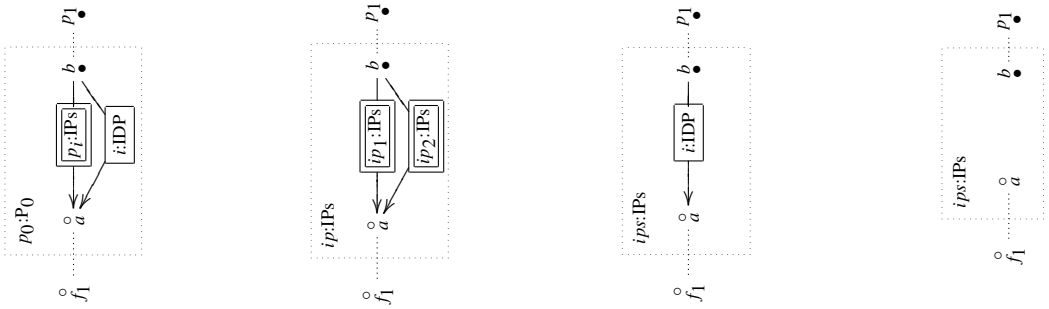


Figure 5.3: The productions for identity providers

a configuration consisting of an IPs and an IDP. Production **ips** generates configuration with many IPs (obtained by refining ip_1 and ip_2). Finally, productions **ip** and **noip** yield non refineable configurations; **ip** generates a single identity provider *i* while **noip** generates an empty configuration. These productions (See Appendix A.2 for the formal definitions) are further described as follows:

The production **pips** given in Figure 5.3 will be used to generate architectural configuration of P_0 consists of an IDP and an IPs. The RHS of the production **pips** takes a non-terminal edge IPs attached to a node of type $\circ$ and one of type $\bullet$ which are shared with the IDP and then both nodes are exported in the interface of the production. In this way, the IPs and the IDP share the $\circ$ node with the same federation edge and share the node $\bullet$ with rest of the providers within or outside the CoT.

The production **ips** given in Figure 5.3 generates an architectural configuration of the IPs consisting of two IPs. The RHS of the production **ips** combines two non-terminal edges of type IPs to a $\circ$ and a $\bullet$ nodes which are then both exported in the interface of the production. In this way, the IPs will all share the $\circ$ node with the same federation edge and use the same node $\bullet$ with rest of the providers within or outside the CoT.

The production **ip** given in Figure 5.3 is meant to generate architectural configuration of the IPs consisting of a single IDP. The RHS of the production **ip** takes nothing

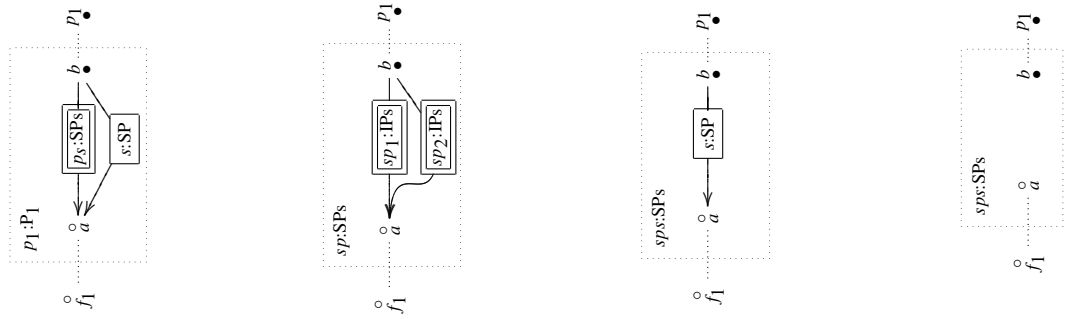


Figure 5.4: The productions for service providers

and attaches the IDP to a ○ and a ● nodes which are then both exported in the interface of the production. In this way, the IPs shares a node ○ with the federation and use a node ● to connect with rest of the providers within or outside the CoT.

The production **noip** given in Figure 5.3 is meant to generate an architectural configuration of an IPs for empty design. The RHS of the production **noip** takes nothing and uses nodes ○ and ● to share with the federation and providers within or outside the CoT are then exported in the interface of the production.

Productions for Service Providers The productions for generating service providers are very similar to those in Figure 5.3. The productions **psps**, **sps**, **sp**, and **nosp** in Figure 5.4 generate the configurations for service providers by refining non-terminal p_s in the production **fed** (Figure 5.2). More detailed descriptions of these productions (See Appendix A.3 for the formal definitions) are given as follows:

The production **psps** will be used to generate architectural configuration of P_1 consists of an SP and an SPs. The RHS of the production **psps** takes a non-terminal edge SPs attached to a node of type ○ and one of type ● which are shared with the SP and then both nodes are exported in the interface of the production. In this way, the SPs and the SP share the ○ node with the same federation edge and share the node ● with rest of the providers within or outside the CoT.

The production **sps** generates an architectural configuration of the SPs consisting of two SPs. The RHS of the production **sps** combines two non-terminal edges of type SPs to a ○ and a • nodes which are then both exported in the interface of the production. In this way, the SPs will all share the ○ node with the same federation edge and use the same node • with rest of the providers within or outside the CoT.

The production **sp** is meant to generate architectural configuration of the SPs consisting of a single SP. The RHS of the production **sp** takes nothing and attaches the SP to a ○ and a • nodes which are then both exported in the interface of the production. In this way, the SPs shares a node ○ with the federation and use a node • to connect with rest of the providers within or outside the CoT.

The production **nosp** is meant to generate an architectural configuration of an SPs for empty design. The RHS of the production **nosp** takes nothing and uses nodes ○ and • to share with the federation and providers within or outside the CoT are then exported in the interface of the production.

Matching constraints with the productions Since ADR productions represent constraints of the style, we discuss how the constraints described in Section 3.1 are matched with the productions described above. To this purpose, we consider the constraints **GC1**, **GC3** and **GC6** (page 55) that restrict FIM configurations with respect to the FIM patterns. In particular, we do not consider the constraints that specify conditions which were required to represent architectural composition of the FIM components in UML.

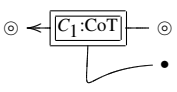
For matching the constraint **GC1**, consider the non-terminal of type CoT in Figure 5.1 which can be replaced with the configurations generated using the corresponding productions in Figure 5.2 namely, production **chain** and production **fed**. The production **chain** generates a configuration of a chain of CoTs while the production **fed** generates a federation of providers attached to each other. In this way, a CoT can be

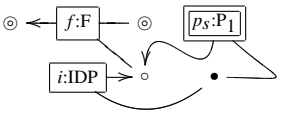
replaced with the configuration describing either a chain of CoTs or a federation of providers.

For matching the constraints **GC3** and **GC6**, consider the non-terminals in production **fed** in Figure 5.2 which can be used to generate configurations of a federation of providers. In production **fed**, the non-terminal of type P_0 represents a set of IDPs while the non-terminal of type P_1 represents a set of SPs in a federation. The productions in Figure 5.3 and Figure 5.4 generate configurations of P_0 and P_1, respectively. Notice that the these non-terminals in the production **fed** are connected to a federation F through node $\circ$ and to each other through node $\bullet$. Observe that the IDPs and SPs are connected to the federation and to each other in the configurations generated by refining the non-terminals in **fed** using their corresponding productions. In this way, constraints **GC3** and **GC6** are satisfied as both the terminal hyperedges of type IDP and SP are connected to the terminal hyperedge of type F through node $\circ$ and to each other by sharing a node $\bullet$.

5.2 Generating Configurations of FIMs

To illustrate how legal configurations of FIMs can be derived using the productions given in Section 5.1.2 we consider the graphs[1] G_1 and G_2 in (5.1)

$G_1 =$ 

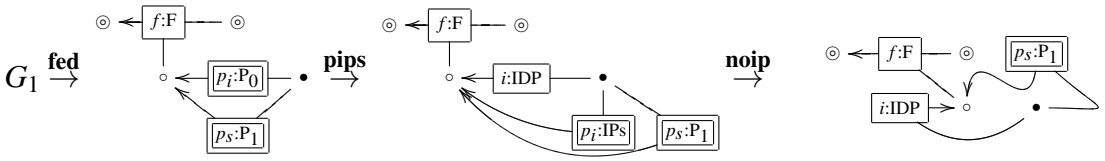
$G_2 =$ 

(5.1)

and show how G_1 can be refined into the configuration G_2.

[1] Graphs G_1 and G_2 are typed over the graph depicted in Figure 5.1 in the obvious way.

The initial sequence of reductions is



Namely, in the first step, **fed** (cf. Figure 5.2) is applied to generate the federation f; then the edge p_i is refined by applying **pips** (cf. Figure 5.3) yielding a provider p_i of type IPs and a provider i of type IDP. Finally, configuration G_2 is obtained by applying **noip** which cancels the non-terminal p_i. Any configuration x refining P_1 yields a term-like representation of G_2 as $\mathbf{fed}(\mathbf{pips}(\mathbf{noip}),x)$ which highlights the hierarchical structure of the FIMs configuration G_2. In this way, configurations of the FIM patterns can be generated and are illustrated in the next section.

With the help of some simple examples, we show how to generate the architectural configurations of FIM patterns **BF**, **MIF**, **MSF**, **AF**, and Chain of CoTs. Notice that graph G_1 in (5.1) represents an initial (abstract) architecture that can be further refined into these configurations. To illustrate this, we use the productions for FIMs and the approach given in Section 5.1.

Generating configuration of FIM pattern BF The configuration with a single IDP and a single SP is generated by applying production **fed** (cf. Figure 5.2) first then followed by a refinement of the non-terminal edges P_0 and P_1 (introduced by **fed**).

In the second step, production **pips** (cf. Figure 5.3) generates the configuration for P_0 consisting of a terminal IDP and a non-terminal IPs. Similarly, productions **psps** (cf. Figure 5.4) generates the configuration for P_1 that consists of a terminal SP and a non-terminal SPs.

Observe that the obtained graph contains non-terminal edges of type IPs and SPs (introduced by applying **pips** and **psps**, respectively); these (spurious) non-terminal

(a) A config. of **BF** (b) A config. of **MIF**

Figure 5.5: A few configurations of the CoT in FIMs

edges are cancelled using the productions **noip** and **nosp**. As a result, we obtain the configuration of pattern **BF** which is graphically represented in Figure 5.5(a) (where edge names are omitted as immaterial). In ADR, such configurations can be given an algebraic formulation; for instance, the configuration in Figure 5.5(a) is given by term $\mathbf{fed}(\mathbf{pips}(\mathbf{noip}), \mathbf{psps}(\mathbf{nosp}))$. Example 2.2 (on page 27) describes such a scenario where a single IDP (i.e., an airline) is federated to a single SP (i.e., a hotel).

Generating configuration of FIM pattern MIF A configuration of pattern **MIF** can be generated in a similar way as done for the configuration of pattern **BF**. For instance, consider the case where two IDPs are federated to a single SP.

Initially, the same sequence of productions can be followed that is used above for generating configuration of pattern **BF** where a single IDP is federated to a single SP, with the difference that before applying the **noip** production, the non-terminal p_i in the production **pips** is further refined using the production **ips**. This introduces an additional IDP where either ip_1 or ip_2 in the production **ips** is on turn refined by applying production **ip**, while the other non-terminal is cancelled using productions **noip**. Notice that the production **ips** can recursively be applied to generate a complex configuration of IDPs. In this way, a configuration of pattern **MIF** is obtained; its term representation is

$$\mathbf{fed}(\mathbf{pips}(\mathbf{ips}(\mathbf{ip}, \mathbf{noip})), \mathbf{psps}(\mathbf{nosp}))$$

and graphical representation is shown by Figure 5.5(b). Example 2.3 (on page 27)

(a) A config. of **MSF** (b) A config. of **AF**

Figure 5.6: A few more configurations of the CoT in FIMs

illustrate a scenario where this pattern may be useful.

Generating configuration of FIM pattern MSF Now, let us consider the case where two SPs are federated to a single IDP. Initially, the same sequence of productions can be followed that is used above for generating configuration of pattern **BF** but with the difference that before applying the **nosp** production the non-terminal p_s in the production **psps** is further refined using the production **sps**. This introduces an additional SP where either sp_1 or sp_2 in the production **sps** is on turn refined by applying production **sp**, while the other non-terminal is cancelled using productions **nosp**. Notice that the production **sps** can be applied recursively to generate a complex configuration of SPs. Consequently, a configuration of pattern **MSF** shown by Figure 5.6(a) is generated and its associated term representation is

$$\mathbf{fed}(\mathbf{pips}(\mathbf{noip}), \mathbf{psps}(\mathbf{sps}(\mathbf{sp}, \mathbf{nosp})))$$

Example 2.4 (on page 28) illustrates a scenario where this pattern may be useful.

Generating configuration of FIM pattern AF Figure 5.6(b) shows a configuration of pattern **AF** where two IDPs and two SPs are federated to each other. In order to generate such a configuration, one may follow the same sequence of productions used

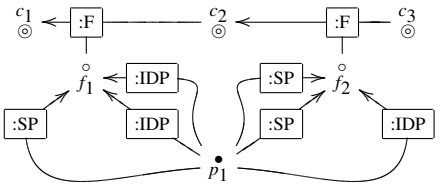


Figure 5.7: A chain of CoTs configuration in FIMs

to generate configurations of FIM patterns **MIF** and **MSF**. Term

$$\mathbf{fed}(\mathbf{pips}(\mathbf{ips}(\mathbf{ip},\mathbf{noip})),\mathbf{psps}(\mathbf{sps}(\mathbf{sp},\mathbf{nosp})))$$

represents this configuration while Example 2.5 (on page 28) describes such a scenario where this pattern may be useful.

Generating configuration of FIM pattern Chain of CoTs Figure 5.7 shows a configuration of a Chain of CoTs where two federations and their providers are attached to each other. In this chain; a CoT is created according to pattern **MIF** where multiple IDPs are federated to a single SP, and the other CoT is created according to pattern **MSF** where a single IDP is federated to multiple SPs.

Initially, the production **chain** is applied that generates two CoTs c_1 and c_2. Since CoT c_1 represents pattern **MIF**, one may follow the same sequence of productions used above for this pattern. Similarly, CoT c_2 can be refined according to pattern **MSF**. Term

$$\mathbf{chain}(\mathbf{fed}(\mathbf{pips}(\mathbf{ips}(\mathbf{ip},\mathbf{noip})),\mathbf{psps}(\mathbf{nosp})),\mathbf{fed}(\mathbf{pips}(\mathbf{noip}),\mathbf{psps}(\mathbf{sps}(\mathbf{sp},\mathbf{nosp}))))$$

describes how such a configuration is built and Example 2.6 (on page 29) describes the scenario where the pattern Chain of CoTs may be useful.

5.3 Modelling Reconfigurations of FIMs

In this section, we show how reconfigurations can be described for FIMs.

Architectural styles may offer a suitable modelling mechanism to guide the changes at the architectural level; in fact, patterns for FIMs can be given in terms of ADR architectural style as illustrated in Section 5.2.

At run time, systems may need to be reconfigured; for instance, adding one or more components. Noticeably, such changes may need to be reflected at the architectural level, namely they may induce *architectural reconfiguration*. FIMs are no exception. The architectures of FIM patterns may evolve during the development life-cycle where IDPs and SPs can be added to the federations. For instance, the configuration of pattern **BF** consists of an airline (i.e., the IDP) and a hotel (i.e., the SP) can be reconfigured (cf. Example 2.3 on page 27) by introducing a train service (i.e., a new IDP). Such a change (adding an additional IDP) in the architecture reshapes the systems from pattern **BF** to pattern **MIF** so as to allow users to book a room after booking a flight *or* a train.

This change of pattern can be defined at *basic* level (namely, one IDP, or one SP, or one instance of both is added) as well as at *abstract* level (namely, arbitrary collections of IDPs or SPs are added at once).

ADR offers a graphical support and a formal mechanism to deal with *style-preserving* architectural reconfigurations, namely architectural reconfigurations that do not modify the style. We remark that it is crucial for FIMs as style preserving reconfigurations correspond to modifying configurations by changing their pattern while preserving a valid (legal) architecture. ADR can also express reconfigurations that violate styles. For instance, it is easy to define reconfiguration rules that cancel components so as to obtain configurations without e.g., IDPs that are not considered valid FIMs. It is also worth remarking that the condition to preserve style is very simple; it is just necessary to ensure that LHS and RHS of the reconfiguration rule have the same type.

As we have seen in Section 5.2, the design rules can be given an algebraic formulation where a term in ADR describes a particular style-proof.

In order to illustrate how ADR reconfiguration rules can describe variations of FIMs we consider the following rules.

$$addIDP : \mathbf{noip} \longrightarrow \mathbf{ips}(\mathbf{ip}, \mathbf{noip}) \tag{5.2}$$

$$addIDPs(X) : \mathbf{noip} \longrightarrow \mathbf{ips}(\mathbf{X}, \mathbf{noip}) \tag{5.3}$$

Intuitively, such rules allow us to add components; more precisely, they respectively introduce a new IDP and a set of many IDPs. Rule (5.2) is defined at basic level (terms without variables) to add a single IDP and rule (5.3) is defined at abstract level (terms with variables) to add a collection of IDPs to the configurations of FIM patterns. Notice that in both rules the *LHS*s and the *RHS*s terms have the same type; this is central to preserve the style. In other words, ADR guarantees, by construction that when all reconfiguration rules preserve the types, then any derivation will not change the architectural style. Similarly, the rules for service providers can be defined

$$addSP : \mathbf{nosp} \longrightarrow \mathbf{sps}(\mathbf{sp}, \mathbf{nosp}) \tag{5.4}$$

$$addSPs(Y) : \mathbf{nosp} \longrightarrow \mathbf{sps}(\mathbf{Y}, \mathbf{nosp}) \tag{5.5}$$

that add one or more service provider components respectively to the configurations of FIM patterns.

Table 5.1: Effects of basic reconfiguration rules on FIM patterns

Reconfig. rules	**BF** to **MIF**	**BF** to **MSF**	**MIF** to **AF**	**MSF** to **AF**	**BF** to **AF**
Rule (5.2)	✓			✓	✓
Rule (5.4)		✓	✓		✓

Table 5.1 shows the effects of basic reconfiguration rules that add a single IDP and a

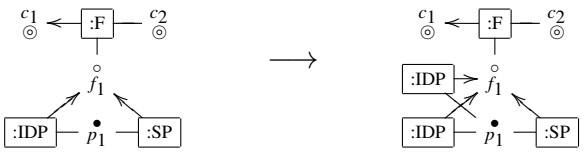


Figure 5.8: Rule to add an identity provider (from left to right)

single SP into FIM patterns **BF**, **MIF**, **MSF**, and **AF**. Figure 5.8 (for simplicity, names of the edges are omitted) illustrates the reconfiguration of pattern **BF** into pattern **MIF** architecture (cf. Example 2.3 on page 27) by applying the rule (5.2). The LHS graph shows configuration of pattern **BF** consisting of an IDP (i.e., an airline) and an SP (i.e., a hotel). This architecture, is reconfigured by introducing an additional IDP (i.e., a train service) that yields a new configuration (RHS graph) that confirms to pattern **MIF** with multiple IDPs federation. To illustrate such a change in the configuration obtained by applying the rule (5.2), transition

$$\mathbf{fed}(\mathbf{pips}(\mathbf{noip}), \mathbf{psps}(\mathbf{nosp})) \longrightarrow \mathbf{fed}(\mathbf{pips}(\mathbf{ips}(\mathbf{ip}, \mathbf{noip})), \mathbf{psps}(\mathbf{nosp}))$$

describes the reconfiguration where the LHS term defines configuration of pattern **BF** while the RHS term defines the new configuration that represents pattern **MIF**. In this reconfiguration, subterm **noip** of type IPs on the LHS replaced with a new term $\mathbf{ips}(\mathbf{ip}, \mathbf{noip})$ of same type on the RHS. Such a transition preserve FIM style. Similarly, the effects of applying the rule (5.4) that adds an *SP* reconfigures the architecture of pattern **BF** to pattern **MSF** (cf. Example 2.4 on page 28). Moreover, rule (5.2) and rule (5.4) can be applied together to reconfigure the architecture of pattern **BF** to pattern **AF**. Furthermore, these rules can be applied separately while moving from patterns **MIF** and **MSF** to pattern **AF**, for instance; rule (5.4) can be used to move from pattern **MIF** to pattern **AF** and rule (5.2) to move from pattern **MSF** to pattern **AF**.

Recall that the non-terminal hyperedge of type CoT can be refined using either the

production **fed** that generates a federation of providers or the production **chain** that generates a chain of CoTs (cf. Figure 5.2 on page 79). In this way, we define the reconfiguration rule

$$addCoT(c_3) : \mathbf{chain}(\mathbf{c_1}, \mathbf{c_2}) \longrightarrow \mathbf{chain}(\mathbf{chain}(\mathbf{c_1}, \mathbf{c_2}), \mathbf{c_3}) \tag{5.6}$$

that adds a CoT component in a chain of CoTs where the variable c_3 can either be instantiated as a federation using the production **fed** or a chain of CoTs using the production **chain**.

5.4 Evaluating the FIMs model

In this model, a few ADR productions are defined to generate valid configurations of FIMs. Recall that the fundamental property of FIMs specifies that at least one IDP and one SP should participate to form a legal CoT. While generating providers in the production **fed** in Figure 5.2(b) (page 79), this property is implicitly preserved via the productions (cf. Figure 5.3 on page 80 and Figure 5.4 on page 81) which can be applied immediately after the production **fed**. The other designated productions allow one to generate the desired configurations of providers.

Moreover, the productions in the model enable designers to systematically generate valid configurations, ranged from the simplest FIM scenarios to the large complex ones, in a precisely controlled way. For instance, Example 2.7 describes a large complex FIMs where 697 IDPs are federated to 12 SPs. To generate such a configuration, initially the production **fed** in Figure 5.2(b) (page 79) will be applied. To further refine abstract components in this production, the productions **pips** in Figure 5.3 (page 80) and **psps** in Figure 5.4 (page 81) will be used. As a result, a FIM configuration will be generated where a single IDP is federated to a single SP. To generate the rest of IDPs

and SPs, the abstract components of the productions **pips** and **psps** will be replaced with the desired configurations using the corresponding productions.

Observe that the the productions for IDPs in Figure 5.3 (page 80) are very similar to the productions for SPs in Figure 5.4 (page 81). Therefore, it is desirable to have a facility in ADR that allows one to define a kind of generic (or meta-)productions so that they can operate on various types of components. For instance, a compile time facility in programming languages such as "Generics" in Java allows a type or method to operate on various object types. In this way, a generic (or meta-)production that generates a provider may instantiate either an IDP or an SP. Similarly, a single production can be developed for the empty designs in this model. As a result, the number of productions can be reduced significantly that enables one to maintain the designs productions in an efficient way.

To support changes in the FIM style, the architectural elements (e.g., CoT) in the existing type graph in Figure 5.1 (page 78) can potentially be used to define new productions. For instance, to define a production that allows CoTs to be connected in star topology. For generating such a structure, one may attach the first two tentacles of hyperedge(s) of type CoT to a common node of type Channing (node $\odot$) in the proposed new set of productions. In this way, one may conveniently define new productions to support more complex topologies (i.e., tree) for the CoT. In this connection, an additional production that generate a federation of providers (i.e., similar to the production **fed** in Figure 5.2(b) on page 79) need to be defined in order to support the new topology of the enclosing CoT. Interestingly, such a change in the topologies of the CoT does not affect the use of existing productions that generate actual providers.

In this model, the required reconfiguration rules have been defined that add components into the FIM configurations. However, the model lacks in reconfiguration rules for removing the FIM components. But one may take advantage of the nature of design productions that generate a collection of IDPs and SPs. Indeed, these productions are

used to formulate the reconfiguration rules in Section 5.3 (page 88) to add one or more IDPs and SPs. Interestingly, these rules can potentially be applied in reverse order (i.e., RHS to LHS) to remove one or more IDPs and SPs.

Chapter 6

Comparing the Modelling Approaches

This chapter compares the approaches for modelling architectural aspects of the FIMs. Specifically, we consider UML, some graph-based approaches, and two ADLs (Acme and C2SADEL).

We first fix the criteria for the comparison and then we discuss the support provided by each of the considered approaches for the criteria. Finally, a comparison between ADR and the other approaches is given and the available tool support for each approach is discussed.

6.1 Criteria for the Comparison

The criteria for our comparison are classified in *general criteria* and *pattern specific criteria*.

6.1.1 General criteria

The general criteria of interest consider the linguistic aspects related to

- core architectural concepts,

- architectural styles,
- style checking,
- reconfiguration, and
- refinement.

For each criterion, we give a brief description and the main motivations that make it relevant for the FIMs.

Core architectural concepts We consider core architectural concepts that include components, connectors, and configurations (cf. Section 2.4 for details).

Architectural styles Architectural styles are one of the pillars of modern software architecture. Architectural styles offer a few benefits [65] including:

- styles enable the reuse of architectural designs;
- styles abstract away platform specific details;
- styles constrain the design space to permit specialised analysis.

More precisely, a style defines a family of related systems (e.g., the FIM patterns **BF**, **MIF**, **MSF**, and **AF**) by providing a common *design vocabulary* together with suitable *constraints*. Notice that we consider the modelling approaches that support this notion of architectural style for the comparison.

The vocabulary of a style defines a set of architectural element types (cf. Section 2.4 for details). Constraints specify allowable configurations of elements from the vocabulary. The vocabulary together with the constraints describe an architectural style.

Style checking The notion of style checking refers to the fact that a configuration can only be valid if it does not violate its style. Style checking can be considered as one of the important properties for architectural analysis [65]. A configuration can only conform to a style if it is a member of the family of configurations determined by that style. For example, the proposed architectural style of FIMs given in Section 5.1 formalises a family of systems by considering each of the FIM patterns as its instance. In this case, there should be a mechanism to check whether a given configuration matches with one of the legal configurations (i.e., the FIM patterns) defined by its style. Therefore, to model FIMs, it is desirable that the modelling approach provides a suitable mechanism to support style checking.

Reconfigurations Since SAs may change during the development life-cycle to satisfy new requirements, it is desirable to have a suitable mechanism in the modelling approaches to deal with the architectural changes. Typically, approaches to model dynamic SAs allow these changes to be formally defined by different kinds (i.e., *basic rules* and *complex rules*) of reconfiguration operations. A basic reconfiguration rule may add one or more concrete components to the configuration. On the other hand, a complex reconfiguration rule adds a collection of components to the configuration in an abstract way. Such a collection of components can suitably be represented by an abstract component which will be replaced by a complex configuration.

Moreover, reconfigurations can be made in SAs either before execution or at run-time. Hence, we distinguished between two types of reconfigurations where one requires changes to be introduced by designers and the other requires changes to be triggered at run-time. We remark that both types of these reconfigurations are indeed possible in FIMs. However, modelling reconfigurations that deal with the changes at run-time heavily depends on implementation details (e.g., realising a dynamic security and trust management systems for managing the CoT) and we abstract away these

details.

Moreover, reconfiguration can be *style-preserving* where any application of the reconfiguration does not violate the style. For instance, one or more providers may join a federation during the development life-cycle. Example 2.3 (page 27) describes a reconfiguration scenario that requires an additional IDP to be added to a FIM configuration where an IDP is connected to an SP.

Also, reconfiguration operations may change the configuration by introducing new types of architectural elements or replacing existing ones. It is out of scope of this dissertation to deal with such reconfigurations. We deal with describing style-preserving reconfigurations for the FIMs. Therefore, we focus on those modelling approaches that support style-preserving reconfigurations.

Refinement Architectures of large systems are often described by a hierarchy of related architectures. Each architectural design in the next lower level of such a hierarchy can be considered as a refinement of the previous level. In an architectural description, an abstract component can suitably represent a system (or part of the system) at the desired level of abstraction. For instance, an abstract CoT may represent a complex configuration of a FIM system or possibly a chain of CoTs. Each CoT in a given configuration may correspond to a different FIM pattern.

According to Garlan [55], despite the fact that no single definition of refinement exists, but its rules must be explicit about what kind of properties must be preserved in the refined design. One of the properties we are interested in is the preservation of interface (potentially realised by ports) of an abstract component when replacing it with the next level detailed architectural design.

For instance, consider the chain of CoTs scenario descried in Example 2.6 (page 29) where two federations are connected. While replacing each abstract CoT in the chain with an actual FIM configuration (e.g., a FIM pattern), one should preserve the inter-

face that allow providers in one CoT to interact with each other and with providers of the other CoTs in the chain.

The stepwise refinement of an abstract (or higher level) architecture into a relatively correct lower level architectures requires predefined *refinement patterns* (or refinement rules). Such a refinement mechanism provides routine solution to a standard architectural design problem. Once a refinement pattern is proven correct, its instances can be used to develop specific architectures [77].

The set of refinement rules can be defined in a style where each individual rule can be proved as a valid refinement relation between the design of an abstract component and the designs having an assembly of (abstract and) concrete components. This technique is known as *style-based refinement* which is naturally more powerful than the techniques that rely on the rules that compare instances of the style [55]. The modelling approaches should provide a suitable mechanism that permits the creation of the refinement rules.

6.1.2 Pattern specific criteria

We describe the criteria related to creating and identifying FIM patterns in terms of instances of architectural styles. The basic criteria we consider are related to pattern generation and pattern identification.

Generating patterns An architectural style allows the precise description of a family of related systems so as to specify systems that abide by a common style. Therefore, a facility is desirable in the modelling approach that specify how the components can suitably be composed in a given configuration. For instance, consider a configuration of FIM pattern **AF** where multiple IDPs can be federated to multiple SPs. (Example 2.5 on page 28 describes such a scenario.) Observe that the number of IDPs and SPs in this pattern may be known at the time when the actual configuration is being created.

In this case, styles offer certain benefits where the refinement rules can effectively be used to generate such a configuration.

Also, it is desirable to have a suitable mechanism to record the history about how the refinement rules were applied in order to generate a given configuration. Consequently, such information may serve the purpose of guiding the designers about how to generate either the same configuration again or the extended ones.

Identifying patterns Since valid configurations of systems may represent specific architectural patterns (e.g., the FIM patterns), applying changes to these systems may result into either reconfiguring one pattern into the other or preserving the same pattern.

Recall that each FIM pattern configuration is considered as an instance of the FIM style. In FIMs, adding one SP into FIM pattern **BF** (cf. Example 2.4 on page 28) will reconfigure it into FIM pattern **MSF**. On the other hand, adding one or more SPs into FIM pattern **MSF** configuration does not change the pattern. Similarly, adding one or more IDPs into the FIM configurations may or may not change the patterns. Since each of the FIM patterns is exposed to different security threats, reconfiguring one pattern to the other has a direct effect on the security requirements. Therefore, use of the pattern identification techniques have become more important while applying changes to these systems.

6.2 UML as an ADL

Recently, UML has been promoted as an ADL with the introduction of structured classifier (i.e., class, component) and port concepts. In describing the support provided by UML for the criteria given in Section 6.1, we mainly follow the approaches to UML-as-ADL in the literature (notably [71, 63, 33]). Also, Table 6.1 and Table 6.2 summarise the support provided by UML and the profile in [33], respectively.

Table 6.1: The UML's support for the criteria

Criteria			The UML profile
General	Core	Components	UML component extends UML's class concept in the metamodel. Both classes and UML components suitably describe SA components.
		Connectors	UML connector does not represent SA connector semantics (e.g., connector types). Alternatively, UML associations, association classes,classes or components may be used where each one has its own advantages and disadvantages to model SA connectors.
		Configurations	Object diagrams model flat view of the system while structure diagrams suitably describe SA configuration in a hierarchical way.
	Styles	Vocabulary	Class diagrams suitably describe vocabulary while structure diagrams (type level) may loosely define vocabulary of style that models various context specific views.
		Constraints	Multiplicity constraints may partially represent style constraints and OCL constraints are often needed to restrict the valid models (e.g., global OCL constraints to model complex association).
	Style checking		Requires the use of validation tools and techniques. Validation of models whose style uses OCL constraints is still problematic and OCL lacks support in the widely available UML tools.
	Reconfigurations		UML does not specify any notation to describe reconfigurations. Consequently, designers directly introduce changes in the configurations.
	Refinement		UML has limited support (e.g., inheritance) for architectural refinement and there is no concept of UML abstract component.
Pattern	**Pattern generation**		Developers use rules of thumb to create configurations.
	Pattern identification		Object diagrams that represent complex systems are difficult to analyse while structure diagrams that represent such systems can effectively be analysed.

We do not consider the approaches (such as [81] and [64]) that change the UML metamodel to directly support the required modelling concepts by introducing new kind of diagrams. As observed in [71], those approaches use the notations which will not conform to the UML specification and could become incompatible with UML compliant tools.

6.2.1 Support for general criteria

Core architectural concepts A UML component directly extends the class notation in the UML metamodel. This extension enables UML components to be associated

Table 6.2: The UML profile's support for the criteria

Criteria			The UML profile
General	Core	Components	Uses UML components to represent SA components.
		Connectors	Uses UML connectors to represent interactions between the components.
		Configurations	Uses structure diagrams to represent configurations.
	Styles	Vocabulary	Class diagram models abstract components and structure diagrams represent their internal structure.
		Constraints	Multiplicity and global constraints OCL to model complex association.
	Style checking		Requires the use of validation tools and techniques in UML/ADR.
	Reconfigurations		Changes one pattern configuration (or a specific context) to the other and provides unsatisfactory support to tackle reconfigurations in a general way.
	Refinement		Components tagged as «`refineable`» that suitably model abstract components.
Pattern	**Pattern generation**		It has the same limitations as UML but one may effectively use productions that create simple configurations (e.g., FIM pattern **BF**).
	Pattern identification		It has the same limitations as UML.

with additional information (e.g., deployment descriptors, property files, etc..). However, such an additional information may not necessarily be required by SA components which often can suitably be represented by UML components or classes.

On the other hand, UML provides unsatisfactory support for connectors. A UML connector models the simplest form of interconnections (e.g., a service call) and it is a simple link between architectural elements (e.g., the connector between an `IDP` and an `SP` associates `Provider_Access` ports in Figure 4.2 on page 69). Also, UML's connectors do not yield any semantical information (e.g., connector types) and one may choose from several mechanisms (see Table 6.1) to describe SA connectors in UML. Each mechanism has its own advantages and disadvantages [63]. For instance, associations are suitable when the interaction mechanisms (e.g., procedure or service calls) are well understood and similar to the interactions of UML connectors. On the other hand, classes (or components) allow one to describe connector semantics, specific component ports, and connector roles. However, it is very difficult to single

out components and connectors in a configuration when the same notation is also used to model components. As observed in [63], if SA components are modelled by classes then UML components can suitably represent connectors.

In UML, configurations can either be represented by object diagrams or by structure diagrams (at instance level). An object diagram (cf. Figure 3.2 on page 61) models a flat view of a running system by showing links between the objects. Conversely, a UML structure diagram (cf. Figure 4.2 on page 69) suitably represent an architectural configuration where connectors model interconnections by associating ports of the components. Moreover, this diagram models a structured view of the system by imposing a hierarchy over a complex system.

Architectural styles A class diagram suitably represents the vocabulary of a style. Constraints over complex associations between components may be expressed using multiplicity and OCL constraints (often not easy to express).

Example 6.1 *It is difficult to clearly specify **GC1** (page 59) for the model in Figure 3.1(b) (page 58). **GC1** specifies a mutually exclusive condition on two different sets of associations, one for chain of* `CoTs` *and the other for* `Federation` *of providers.*

In order to constrain Figure 3.1(b) with **GC1**, all associations relating the associations on chain of `CoTs` (resp. `Federation` of providers) must be linked together. Moreover, the links associated with each set need to be related in order to specify the mutually exclusive condition. Notice that this may render the resulting diagram complex. Moreover, UML classes and associations are not enough to define architectural composition. In fact global (OCL) constraints are often required.

Example 6.2 *The* `CoT` *in Figure 3.1(b) describes complex associations among various providers (i.e.,* `IDPs` *and* `SPs`*) and their associated components (i.e.,* `DTMSs`*). In this case, global constraints (e.g., **GC2** on page 60) are required even for simple conditions.*

On the other hand, UML structure diagrams may reduce the use of the constraints that are defined over the associations among the composed component and its associated components. For instance, **GC2** is immaterial while using those diagrams. However, these diagrams may still require to constrain complex associations between the involved (contained) components (e.g., **GC3** on page 60).

We remark that UML does not support the notion of abstract components. In [33], a UML profile has been introduced to support style-based (and reconfigurable) design of service-oriented systems. This profile exploits UML generalisation concept to associate an abstract component to the productions. In this way, the profile precisely describes the way abstract (or refineable) components can be refined using the corresponding productions. Such a refineable component can be replaced by one of the productions whose configuration is described in the corresponding structure diagram.

Style checking The use of certain modelling tools (e.g., [68, 47, 58]) and techniques (e.g., [41]) is required to validate a configuration. For example, validating an object diagram as an instance model requires checking whether the object diagram fulfils the constraints defined in its class diagram. However, the problem of well-formed instantiation of UML models that use OCL constraints is still not satisfactorily addressed [40]. We argue that OCL is not well supported in the UML practice. Since the profile in [33] is developed to capture variety of stylistic issues which are addressed by an ADL, style-checking support in UML needs to be studied and investigated in terms of that profile.

Reconfigurations In UML, there is no mechanism to describe SA reconfigurations. Consequently, designers have to introduce the changes directly in the configurations as requirements emerge. Such an approach may create consistency problems in UML models.

In Section 3.4 (page 62) we considered reconfigurations of FIM patterns defined according to the recent approach proposed in [33]. Such approach provides unsatisfactory support to tackle reconfigurations that allow components to be independently added or removed. In [33], the use of structure diagrams to define reconfiguration rules allows a reconfiguration rule to change a specific FIM pattern into another. However, the support for style-preserving reconfigurations is not ideal (cf. Section 3.4). Another disadvantage is that the profile in [33] makes difficult to figure out which part of a configuration changes in a reconfiguration.

Refinement UML promotes object-orientation as a general modelling paradigm. As noticed in [83], this is not sufficient to describe architectural refinement. Instead, the UML profile in [33] proves to be suitable for architectural refinement. Such an approach exploits the use of UML class diagrams to describe an abstract component and to associate it with a set of refinement rules. In this way, a structure diagram is used to represent the internal structure of an abstract component amenable to be refined with productions. Such profiles provide limited support to create general refinement rules which need to be tailored to specific configurations.

6.2.2 Support for pattern specific criteria

Generating patterns Typically UML designers use rules of thumb to generate configurations (e.g., in object diagrams or the structure diagrams) according to given specifications. This requires a careful understanding of the relationships between the architectural elements and the constraints described by the style. For configuration of FIM patterns, besides topological information, designers have to consider the complexity of the pattern (i.e., number and kinds of components). For instance, generating configuration for **BF** is easier than for **AF** because the former pattern has a fixed number of components.

As argued, object diagrams can suitably describe simple scenarios (e.g., pattern **BF**) but, for complex ones (e.g., chain of CoTs), structure diagrams are more appropriate. The profile in [33] uses the structure diagrams at the "type level" to describe architectural configurations; such profile yields design productions.

Identifying patterns To identify a pattern, one has to analyse the UML diagram representing its configuration. This requires to consider instances and their associations. For instance, FIM patterns differ on the number of their `IDP`s and `SP`s and/or how they are attached to a common `Federation`. Hence, to identify such patterns one may need to enumerate the instances of type `IDP` and `SP` in a given configuration. Typically, object diagrams (cf. Figure 3.2 on page 61) are difficult to be analysed[1] when they describe complex configurations. On the other hand, the structure diagrams (cf. Figure 4.2 on page 69) may effectively be used to identify such configurations (e.g., belonging to FIM patterns chain of CoTs).

6.3 ADR as an ad-hoc ADL

In this section, we briefly describe the support provided by ADR [38] for the criteria given in Section 6.1 and Table 6.3 summarises this support. The assessment is mainly based on our experience of using ADR to model architectural aspects of the FIMs described in Chapter 5.

6.3.1 Support for general criteria

Core architectural concepts In ADR, hyperedges model components and nodes model interconnections between the components. A tentacle leaving a hyperedge and

[1] Automated analysis is possible (e.g., by parsing XMI representation in UML tools) but it requires customised solutions that could be impractical.

Table 6.3: ADR's support for the criteria

Criteria			ADR
General	Styles Core concepts	Components	Hyperedges model SA components where non-terminal hyperedges represent abstract (or refineable) components and terminal hyperedges represent basic (or non-refineable) components.
		Connectors	Nodes of the hypergraph model interactions between the components. Tentacles leaving hyperedges and joining a common node model roles of the components in a given interaction.
		Configurations	Typed hypergraphs describes architectural configurations.
	Styles	Vocabulary	A type hypergraph suitably describe vocabulary of a style.
		Constraints	A set of productions that formally define legal connections between the components in a configuration.
	Style checking		ADR uses a general algebraic approach where a term provides an admissible justification for the well-typedness of SAs.
	Reconfigurations		ADR uses rewrite rules to define style-preserving SA reconfigurations.
	Refinement		Abstract components are refined in a step-wise hierarchical way using the corresponding productions.
Pattern specific	**Pattern generation**		The productions can be applied in a specific order to generate configurations of the patterns. Also, terms that describe valid configurations (e.g., the FIM patterns) can be reused to precisely generate the same configuration again or the updated one by applying the reconfiguration operations.
	Pattern identification		A configuration has an associated term like representation which may effectively be parsed to identify a pattern by enumerating instances of particular types.

joining a node describes the role of the component in the interaction. More precisely, tentacles represent the assignment of the roles of the components to their respective ports. Tentacles of two or more hyperedges joining a common node represents an interconnection between the components. Figure 6.5 (page 128) shows a few FIM configurations where components of types IDP and SP are associated through particular type of port (node •). A terminal hyperedge (single line box) represents a basic (or non-refinable) component and a non-terminal hyperedge (double line box) represents an abstract (or refinable) component.

The concept of typed design featured by ADR describes components and their interconnections in a configuration. Such a design is defined by a hierarchical (non-terminal) hyperedge whose internal structure ranges from an empty graph to an arbitrary complex graph [36]. Figure 6.5(a) (page 128) shows a configuration where a single IDP and a single SP are connected through port p_1 (node •) which are then connected to a federation through port f_1 (node ○). This configuration describes the FIM

pattern **BF** and whose interface is defined by a non-terminal hyperedge of type CoT.

Architectural styles In approaches based on graph grammars to describe SAs, a type graph suitably represents architectural elements and it has one edge and one node for each different type of component and port respectively. In ADR, the vocabulary of a style is given by the type graph while productions define the legal connections between the components. Figure 5.1 (page 78) shows the type graph that represents architectural elements to model the FIMs. In this type graph, non-terminal hyperedges (e.g., CoT) represent the types of abstract components and terminal hyperedges (i.e., F, IDP, and SP) represent the types of the (basic) components. Also, the type graph has one node for each type of port.

In ADR, the notion of design productions specify how components can legally be connected in a configuration where non admissible configurations can be ruled out by construction. ADR productions are very much like the designs that describe configurations but their underlying graph can have non-terminal hyperedges. These hyperedges will be refined using the corresponding productions to generate their configurations. Figure 5.2 (page 79) shows the productions that can be used to generate configurations of an abstract CoT in the FIMs. We remark that the constraint **GC1** (page 55) can be matched with these productions. The non-terminal hyperedges in these productions can further be refined using the corresponding productions given in Figure 5.3 (page 80) and Figure 5.4 (page 81). Constraints **GC3** and **GC6** can be matched with these productions as they attach all providers (i.e., IDPs and SPs) in the configurations of a CoT to a common federation of type F through federation access port (node $\circ$) and to each other through provider access port (node $\bullet$).

Moreover, the design productions in ADR can be considered as hyperedge replacement rules that are fundamental to the graph grammar based approaches to architectural style. In ADR, the set of design productions together with the type graph represent an

architectural style.

Style checking The architectural configurations are represented in ADR through typed graph having types associated in the corresponding type graph. The relationship between the architectural elements given in a type graph and the actual elements used in the typed graph is represented by graph morphism that maps each instance to its type.

In ADR, style checking is done by using a general algebraic approach. ADR uses a style-based refinement approach where a non-terminal is refined by applying the corresponding productions. Such a refinement process is given the algebraic formulation used in Chapter 5. More precisely, a configuration in ADR has an associated term representation that describes how such a configuration is built and provides a witness of its construction. To check whether an architecture conforms to a style is reduced in ADR to writing a term that encodes the structure of the architecture. Such a term provides an admissible justification for the well-typedness of the actual architectures.

Reconfigurations ADR is a graph-based formal approach where rewrite rules define SA reconfigurations. The design rules (or productions) in ADR can be given an algebraic formulation where a term describes a particular style-proof. In ADR, style-preserving reconfigurations are operated at the level of style-proofs by exploiting term rewriting over style-proof terms.

Reconfiguration operations in ADR take the flavour of graph transformation rules. A graph transformation rule is defined as a rewrite rule $L \longrightarrow R$ where L and R are terms of the same type. For example, consider the FIM style given in Chapter 5, the (basic) rule

$$addIDP : noip \longrightarrow ips(ip, noip) \tag{6.1}$$

adds an IDP component to a CoT in the FIMs. The term on the LHS of (6.1) defines an

empty design of type IPs that will be replaced by a single IDP described by the term on the RHS of type IPS. Such a condition enforces style preservation by allowing both the LHS and the RHS of the reconfiguration rule with the same type of the interface graph [38].

In ADR, basic rule (without variables) can add a single component while a complex rule (with variables) allows a collection of components to be added at once. For example, the complex rule

$$addIDP(X) : noip \longrightarrow ips(\mathbf{X}, noip) \tag{6.2}$$

defines a reconfiguration operation which may add a collection of IDP components. In this case, variable **X** of type IPs may be replaced with a subterm that describes a complex configuration generated using the corresponding productions (cf. Figure 5.3 on page 80). Consequently, ADR allows reconfigurations that can be applied in any larger context by instantiating the parameters according to the required types.

Refinement The process of refining an abstract component is achieved by applying the corresponding productions in a step-wise hierarchical way. Such a refinement process allows designers to consider productions that captures the relevant details in terms of the components required by the configuration. Furthermore, ADR supports style-based refinement that precisely models the relationship between an abstract component and a set of design productions. For example, productions **ips**, **ip**, and **noip** (cf. Figure 5.3 on page 80) generate configurations of an abstract component of type IPs. These productions can be used freely to refine IPs (such as in the production **pips** given in Figure 5.3 on page 80) that may represent a collection of IDP components.

Also, a production in ADR describes the way components are composed in an internal structure of an abstract component. For instance, production **chain** in Figure 5.2(a)

(page 79) takes two designs of type CoT and returns a CoT. The composition pattern of the two CoTs connected in a chain can be seen in the underlying design of the production. Such a composition mechanism describes the complex relationships between the components. Consequently, ADR does not require explicit knowledge about the constraints where non admissible configurations (e.g., connecting clients) can be ruled out by construction.

6.3.2 Support for pattern specific criteria

Generating patterns Since an architectural style describes a family of systems, each instance of the style is to be considered as a valid configuration of a system. In an architectural description, an abstract component (e.g., a CoT in the FIMs) may represent such a system at the desired level of abstraction. The refinement mechanism of ADR can be used to generate configuration of an abstract component. ADR precisely defines the relationship between an abstract component and the corresponding design productions. When a set of design productions are defined within a style to refine an abstract component, ADR allows designers to freely choose the productions in order to generate the required configuration (e.g., a FIM pattern). For instance, non-terminals in productions **pips** (Figure 5.3 on page 80) and **psps** (Figure 5.4 on page 81) should be cancelled by applying the corresponding productions so as to generate configuration of FIM pattern **BF**. In this way, one may need to apply the productions in a specific order so as to generate configurations of the patterns.

Furthermore, one may exploit the expressive power of simple algebraic approach of ADR where a term suitably describes the composition of the components in a given architecture. In ADR, a term formalises a configuration that is described in the graph corresponding to it. Also, a term in ADR represents the way in which the configuration is built. Therefore, terms in ADR can effectively be used to guide the construction of

the same configuration again or the extended ones by applying the reconfiguration rules defined by the style. For example, consider the FIMs style given in Chapter 5, term

$$\mathbf{fed}(\mathbf{pips}(\mathbf{noip}), \mathbf{psps}(\mathbf{sps}(\mathbf{sp}, \mathbf{nosp})))$$

describes a valid architectural configuration of FIM pattern **MSF**. In this way, terms in ADR describe instances of styles and they can be used to guide the well-formed instantiation of the same configurations again.

Identifying patterns In ADR, (typed) designs describe configurations of the systems (e.g., the FIM patterns). A configuration in ADR is generated by applying the corresponding productions and it has an associated term like representation. One may exploit such a term associated with the configuration to identify the pattern. In order to do so, one has to parse the corresponding term representation of the configuration and consider the occurrences of the productions that have terminal hyperedges of particular types in their underlying designs.

Configuration of the FIM patterns are differentiated by the number and the kinds of providers (i.e., IDP and SP) participate in the CoT. The productions given in Figure 5.2(b) (page 79), Figure 5.3 (page 80), and Figure 5.4(page 81) generate configurations of a CoT, IDPs, and SPs respectively. Whenever productions **ips** and **ip** are applied each of them generates a single IDP. Similarly, productions **sps** and **sp** generate an SP. In order to identify a FIM pattern, one has to take into account the occurrences of these productions in the terms to enumerate components of type IDP and SP respectively. For example, consider the FIMs style, when term

$$\mathbf{fed}(\mathbf{pips}(\mathbf{noip}), \mathbf{psps}(\mathbf{nosp}))$$

is parsed following such an approach, one may recognise its associated configuration

as the FIM pattern **BF** where a single IDP is federated to a single SP.

6.4 Other Possible Approaches

We considered a few alternative graph-based architectural modelling approaches and ADLs that may support the criteria given in Section 6.1. More precisely, we considered the graph-based approaches in [73, 74, 61, 27] and two ADLs (*C2SADEL* [69] and *Acme* [56]). Notice that none of these approaches have explicitly been applied to FIMs. But those approaches are selected for the comparison due to the fact that they provide the support for most of the criteria given in Section 6.1 and particularly they describe

- core architectural concepts and
- architectural styles

to promote style-based architectural development.

6.4.1 Support for general criteria

Core architectural concepts Graph-based approaches give visual representation of the core concepts typically equipped with formal descriptions. Le Métayer [73, 74] represents nodes as components and edges as connectors. In this approach, a graph is formally defined as a multiset that describes a configuration.

Hirsch et al. [61] describe configurations by graphs. Edges of the graph model components and connectors. Nodes model communication ports and roles.

Baresi et al. [27] follow a general approach to describe components and connectors as nodes of the graph while edges describe the possible relationships between the architectural elements. They use UML object diagrams to describe configurations.

ADLs typically uses textual notations together with the means (e.g., a configuration language) to describe SAs. ADLs fully support description of the core architectural

concepts by explicitly modelling connectors (cf. [72]). Such a support for the core architectural concepts is also provided by *C2SADEL* and *Acme*.

Architectural styles Le Métayer [73, 74] uses context-free graph grammars to describe architectural styles and defines the constraints that specify the actual connections between the entities. To describe the constraints **GC3** and **GC6** (page 55) in Le Métayer's approach, one may consider the unary relations which characterise the roles of the FIM components (i.e., CoT, Federation, IDP, and SP) where the binary relations of such roles describe the directed links between these components. Further, a set of terms on the RHS of the productions can precisely produce such links with respect to the conditions described in the FIM constraints. Similarly, Hirsch et al. [61] describe architectural styles by hyperedge context-free grammars. In this approach, the productions of the grammars are grouped in three sets including *static productions*, *dynamic productions*, and *communication pattern productions*. Static productions are used to construct an initial configuration while the rest of the productions define reconfigurations. While describing such productions to model the FIM systems, one has to consider the constraints **GC3** and **GC6** over the relationships between the hyperedges that represent the FIM components. Since the hyperedges in [61] model basic components, one may consider the FIM components of type Federation, IDP, and SP. More precisely, a static production can be given that describes an initial FIM configuration where one or more IDPs and SPs are attached to each other and a common Federation in order to match with the constraints **GC3** and **GC6**, respectively. Furthermore, the productions which describe reconfigurations should also respect these conditions while introducing one or more IDPs and/or SPs in the FIM configurations.

Baresi et al. [27] formally define architectural style as graph transformation system having type graphs, constraints, and transformation rules. They represent type graphs as UML class diagrams where classes represent nodes and UML associations model

edges of the graphs. Notice that in the graph-based approaches that use hypergraphs to describe architectural styles, hyperedges suitably model components and nodes model the interconnections between the components. In addition to multiplicity constraints, they may need OCL constraints over associations to restrict valid configurations as in UML. Therefore, it would be necessary to give the OCL representation of the constraints **GC3** and **GC6** while modelling the FIM system using the approach of Baresi et al. [27]. Furthermore, they use an extended notion of architectural style to support structural constraints together with platform-specific (e.g., SOA) communication and reconfiguration mechanisms.

C2SADEL [69] allows one to define only a particular kind of style (i.e., *C2* style [88]), where connectors are explicitly modelled. In *C2* style, components and connectors are defined as types and they have *top* and *bottom* defined. The top of a component can be attached to the bottom of a single connector. Similarly, the bottom of a component is attached to the top of a single connector. Furthermore, a connector can have multiple components attached to its top and bottom. Furthermore, in C2 components are only aware of the components which are ”above” within the hierarchy. The components are completely unaware of the components that reside ”beneath” them within the hierarchy. In *C2SADEL*, any configuration created following such a style is restricted by these simple conditions. However, C2SADEL requires the use of analysis tools in order to further define the design constraints over the configurations. Therefore, one may need to transform architectural representation from C2SADEL into an other ADL (e.g., UML) that has such kind of tool support. To this purpose, Abi-Antoun and Medvidovic in [25] have defined a set of rules which can be used to transform C2SADEL models into UML models. Furthermore, they also provide a tool support for automatically generating UML models (i.e., class diagrams and object diagrams) corresponding to the C2SADEL ones. In this case, the constraints (i.e., **GC0**-**GC5**) defined over the UML models in Section 3.2 (page 57) are needed to further restrict such UML models.

In *Acme* [56] one can describe styles in a general way, in fact architectural style is defined as a set of architectural element types and a set of constraints. These constrains are defined using *first order logic predicates* with some additional information specifying architectural-relevant predicates. Acme has its first order predicates language extension called *Armani* (cf. [85] for further details) which can be used to represent structural constraints. To model the FIM systems in Acme, the constraints **GC3** and **GC6** whose Armani representation (i.e., similar to OCL) will be needed. Moreover, both *Acme* and *C2SADEL* support component sub-typing where a sub-type satisfies all structural properties of its super-type together with their constraints.

Style checking Le Métayer [73, 74] uses static type checking (without tool) to ensure that rewriting rules are consistent with the style. Similarly, Hirsch et al. [61] exploit graph rewriting, which defines the construction of SA by applying the productions, to represent valid SA configurations.

Baresi et al. [27] addressed style checking in a way that ensures that instance graph (configuration) is consistent with the type graph by using graph transformation tools (i.e., AGG, PROGRES, Fujaba, GTXL, etc.) or model checking (i.e., CheckVML, GROOVE, etc..).

C2SADEL [69] treats components as types and it performs type checking at runtime. The unwanted connections (e.g., interface mismatch) between the components can be detected by the type checking. However, one may require the use of analysis tools in order to ensure well-formedness of newly created architecture in *C2SADEL*.

The style editor of *Acme* allows rules to be defined for the correct composition of the architecture. These rules can be checked by its development tools (i.e., AcmeStudio [84]). In order to validate configurations, *Acme* requires the use of different analysis tools (e.g., Armani constraint analysis tool in [85]) that can be integrated with its development tool. Recently Kim and Garlan in [65] use *Alloy* to perform style checking

of a configuration defined in *Acme*.

Reconfigurations Le Métayer [73, 74] uses a *coordinator* that is responsible to control the changes in the architectures. The coordinator is expressed in terms of conditional graph rewriting and it reads public variables of components to control the changes. These changes can be done by creating and removing entities and links. This approach uses static type checking without tool support to prove that the rewriting rules are consistent with the style. Also, the Le Métayer's approach provides an algorithm to verify that the changes done by the rewriting rules preserve the style [31].

Hirsch et al. [61] describe dynamic productions (rules) for creating and removing architectural elements to define dynamic evolution of SAs. They use graph rewriting over productions combined with constraint solving to specify how components will evolve. As observed in [39], Hirsch et al. in [61] do not provide a specific verification mechanism to ensure that the changes are consistent with the given style.

Baresi et al. [27] use graph transformation rules to define reconfiguration mechanisms which allows architectural changes at run-time. The application of transformation rule to an instance graph requires rewriting a part of that graph while preserving the style.

C2SADEL [69] supports basic kind of reconfiguration operations. It can add or remove a single component at a time. In *C2SADEL*, components (and connectors) can be added and removed by using *Weld* and *Unweld* operations respectively.

Batista et al. [28] proposed a meta-framework called *Plastik* that extends *Acme* to support reconfigurations. They provide three extensions to *Acme* by introducing new constructs for describing reconfigurations. The first extension is a conditional construct describing run-time conditions for programmed reconfigurations. The second extension provides a pair of constructs for removing an existing architectural elements from the configuration. The third extension deals with describing run-time dependen-

cies between the architectural elements so that architectural mismatches can be avoided when new elements are added. Those ADLs require analysis tools to check that reconfigurations are style preserving.

Refinement Le Métayer [73, 74] defines a set of rules for the refinement of an abstract component in a given style. Similarly, Hirsch et al. [61] define all productions as rewrite rules. However, they use an implicit notion of abstract components where the relationships between abstract components and the corresponding rewriting rules are not explicitly defined.

Baresi et al. [27] address structure preserving and behaviour preserving refinement of an abstract *platform-independent* (PI) style into a *platform-specific* (PS) style. They define one-to-one structural mapping between the elements of PI style and PS style. However, their focus is more on behavioural refinement where the elements of PS style are further refined using UML's generalisation concept.

C2SADEL provides certain features to support refinement of components across the levels of abstraction. It allows sub-typing of components and places additional constraints on refinement maps to prove certain properties of the architecture [70]. Architectural level descriptions can be mapped to their implementations in *C2SADEL*.

In *Acme*, architectures can be described in a hierarchical way where a component and a connector can be represented by one or more lower-level descriptions called *Acme representations*. An *Acme* component can have multiple alternative implementations and it also supports multiple refinement levels. The concept of *rep-map* in *Acme* defines the actual mapping between an architectural element and its associated multiple alternative implementations.

6.4.2 Support for pattern specific criteria

Generating patterns Le Métayer [73, 74] defines a style as a graph grammar whose rules formally represent configurations. In order to generate configurations of patterns, the rules defined by a given style should be applied in a particular order (cf. [74, page 3]). The actual configuration of a pattern can formally be represented in the corresponding multisets of the underlying graph.

Hirsch et al. [61] describe static productions, which can be used to obtain an initial graph that represents a configuration. These productions can effectively be used to generate the simple patterns (e.g., FIM pattern **BF**). But the patterns (e.g., FIM pattern **AF**) having arbitrary length of components cannot suitably be generated using such a mechanism.

Baresi et al. [27] do not provide any detail about how to construct an initial (instance) graph. However, they assume that the designers may need to use heuristics or the existing techniques to derive correct platform-specific configurations from platform-independent ones.

In *C2SADEL*, the rules for composing architectural elements can effectively be used to generate specific architectural patterns. Instead, *Acme* lacks similar support for generating the specific patterns of configurations.

Identifying patterns Le Métayer [73, 74] formally represents configuration graphs as multisets. In order to identify a pattern, one may exploit such a representation by parsing the multiset and taking into account the occurrences of instances of a particular type.

Hirsch et al. [61] represent configurations as graphs that are generated by applying the corresponding productions. Baresi et al. [27] use UML object diagrams to describe configurations as instance graphs. The approaches in [27] and [61] require one to analyse the graphs that describe configurations in order to identify the patterns. Therefore,

identifying a pattern in those approaches has the same limitations as with UML object diagrams.

In *C2SADEL* and *Acme*, it may be difficult to identify a pattern in a configuration because one may require the use of ad-hoc techniques. For example, a facility may need to be incorporated in the executables so that the patterns they represent can be identified at run-time. To the best of our knowledge, neither *C2SADEL* nor *Acme* provides a mechanism that can be used to identify the patterns at run-time.

6.5 A Comparison

In this section, we give the comparison of ADR, UML, and the other approaches using the criteria described in Section 6.1. We refer to the previous sections of this chapter which respectively describe the support provided by these approaches for the criteria. Also, Table 6.4 and Table 6.5 outline this comparison. Notice that we call the approach of Baresi et al. [27] GT4SA (after graph transformation for SAs), Hirsch et al. [61] HR4SA (after hyperedge replacement for SAs), and Le Métayer [73, 74] GG4SA (after graph grammars for SAs).

6.5.1 Using general criteria

Core architectural concepts To model SA components, C2SADEL describes a particular type of components (i.e., C2 components) with fixed number and kinds of ports. All the other approaches suitably describe SA components in a general way. UML's connector concept does not represent SA connector semantics (e.g., connector types). Alternatively, one may choose from various other notations (i.e., classes, UML components, etc.) to describe SA connectors in UML. All of the considered approaches suitably describes configurations of the systems.

Architectural styles All of the approaches suitably describe style vocabulary. However, a few of them lack a mechanism that may suitably describe constraints within a style. For instance, UML, the profile in [33], and GT4SA may require OCL constraints to restrict the valid models. Such constraints are often not easy to express. Also, OCL lacks support in the UML tools which are widely available.

Style checking Style checking of configuration in UML (and the profile in [33]) and GT4SA require the use of the validation tools and techniques. As observed in [40], validation of the configuration whose style uses OCL constraints is still problematic. In the ADLs, their run-time systems support type-checking of the configurations. As a result, configurations in the ADLs may further require the use of certain analysis tools (e.g., Alloy) so as to validate such configurations against their styles. On the other hand, the graph grammar based approaches including ADR, HR4SA, and GG4SA use a formal mechanism that ensures construction of valid configurations.

Reconfigurations In UML, designers directly introduce the changes in configurations which may create consistency problems. Also, the UML profile in [33] provides limited support to describes the changes by reconfiguring one FIM pattern to the other. We remark that this profile provides unsatisfactory support to reconfigure a FIM pattern while preserving the pattern (e.g., by adding one or more IDPs in FIM pattern **MIF**). In the ADLs, C2SADEL allows a single component (or connector) can be added or removed at a time. On the other hand, the rest of the approaches suitably describe reconfigurations. Furthermore, these approaches (except HR4SA) also provide the means to check whether the changes preserve the style or not.

Refinement As noticed in [83], UML supports the refinement through its generalisation concept (i.e., inheritance) which is not sufficient to deal with the architectural refinement. We remark that UML does not support the notion of abstract architec-

tural components. GT4SA uses style-based refinement to define one-to-one structural mapping between the elements of two styles and it mainly focuses on behavioural refinement. As observed in [70], *C2SADEL* also provides limited support to deal with architectural refinement. On the other hand, the Profile in [33], ADR, GG4SA, HR4SA, and Acme suitably describe architectural refinement.

Approaches			Core concepts			Style definition		Style checking	Reconfig.	Refinement
			Component	Connector	Configuration	Vocabulary	Constraints			
UML		*"as-is"*	class/UML comp.	UML conn.	object/structure diag.	class diag.	cardinality and OCL	validation tools	No	Limited
		Profile	UML comp.	UML conn.	structure diag.	class diag.	cardinality and OCL	validations tools	Limited	Yes
ADR			hyperedge	node	typed hypergraph	type hypergraph	productions	style-proof terms	Yes	Yes
Others	**Graph**	*GT4SA*	node	node	instance graph	type graph	cardinality and OCL	graph trans. tools	Yes	Limited
		GG4SA	node	edges	graphs (multisets)	graph grammars	productions	static type checking	Yes	Yes
		HR4SA	edge	nodes	graphs	graph grammars	productions	rewrite rules	Limited	Yes
	ADLs	*C2SADEL*	C2 comp.	C2 conn.	C2 configuration	types in C2	C2 constraints	analysis tools	Limited	Limited
		Acme	Acme comp.	Acme conn.	Acme configuration	set of elements	constraints in FOL	analysis tools	Yes (external)	Yes

Table 6.4: Comparing against the general criteria

Approaches			Pattern specific criteria	
			Pattern generation	Pattern identification
UML		*"as-is"*	no specific guidelines	analysing object/structure diagram
		Profile	limited (i.e., support for simple scenarios such as FIM pattern **BF**)	analysing structure diagrams
ADR			explicit order of the productions or exploiting the terms	parsing the terms
Others	**Graph**	*GT4SA*	no specific guidelines	analysing object diagrams
		GG4SA	explicit order of the production.	parsing multisets
		HR4SA	limited (i.e. via static graphical productions)	analysing the graphs
	ADLs	*C2SADEL*	limited (e.g., C2 composition rules)	analysing C2 configurations
		Acme	no specific guidelines	analysing Acme configurations

Table 6.5: Comparing against the pattern specific criteria

6.5.2 Using pattern specific criteria

Generating patterns In UML, developers use rules of thumb to instantiate configurations of the patterns. Also, the profile in [33] and GT4SA have the same limitation as with UML to generate configurations of the patterns. On the other hand, ADR, HR4SA and GG4SA formally define the refinement rules to generate configurations. Such rules may guide designers in a precise way to generate the configurations of the FIM patterns by applying the rules in a specific order. The ADLs (Acme and C2SADEL) lack such a formal mechanism that precisely guide the users about how to create configurations of the patterns. However, the composition rules in *C2SADEL* may effectively be used to generate specific patterns as instances of C2 style.

Identifying patterns In UML, the formal description (i.e., XMI representation) of the diagrams can potentially be used to automate pattern identification. The disadvantage of this approach is that one has to rely on specific UML tools that provides such information. Alternatively, one has to read object diagrams or (instance level) structure diagrams to identify patterns. However, this approach is not feasible to analyse the UML diagrams that represent complex configurations. Also, the profile in [33] and GT4SA have these limitations.

In ADR, HR4SA and GG4SA, the formal description of the configurations can effectively be parsed to identify patterns. To identify patterns at run-time (e.g., after reconfigurations) in the ADLs (Acme and C2SADEL), they require specific techniques to be incorporated in the executables. To the best of our knowledge, these ADLs lack a mechanism (e.g., programming language features) that can be used to realise such techniques. For instance, an architectural programming language called *Java/A* integrates architectural representations into Java. As observed in [39], Java/A is in the inception phase and its compiler is yet to be completed.

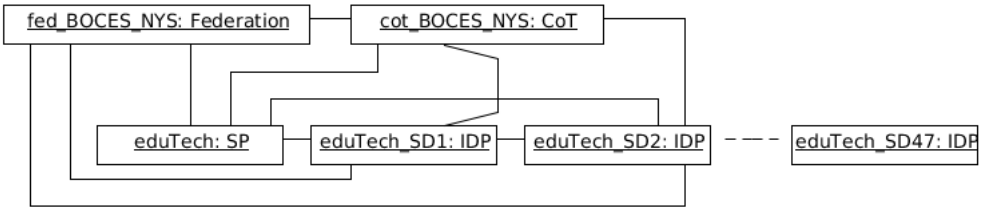


Figure 6.1: An object diagram for the educational FIM

6.5.3 Using a case study

In this section, we evaluate UML, the profile in [33], and ADR by applying these approaches to the scenario described in Example 2.7 (page 29). First, we apply UML, the profile in [33], and ADR and then we draw some conclusions.

Applying the Approaches to the Case Study

UML When using UML, we use object diagrams to represent FIM configurations. Such diagrams model flat view of the system and they can suitably represent a snapshot of a complex configuration of a large running system.

Example 6.3 *In the scenario of Example 2.7 (page 29) the regional information center (RIC) acts as an IDP and its is called EdutTech. EdutTech is responsible for providing services to the teachers (and administrators) in 47 school districts (which play the role of IDPs) associated with the* Boards of Cooperative Educational Services *(BOCES) in* New York State *(NYS).*

The scenario in Example 6.3 is rather complex. The underlying FIM pattern of such a configuration is **MIF** where a single SP is federated to multiple IDPs.

Figure 6.1 represents an object diagram which describes a configuration of the FIM scenario described by Example 6.3. In this diagram, the SP `eduTech` is attached to the IDPs `eduTech_SD1` and `eduTech_SD2` while the IDP `eduTech_SD47` followed by

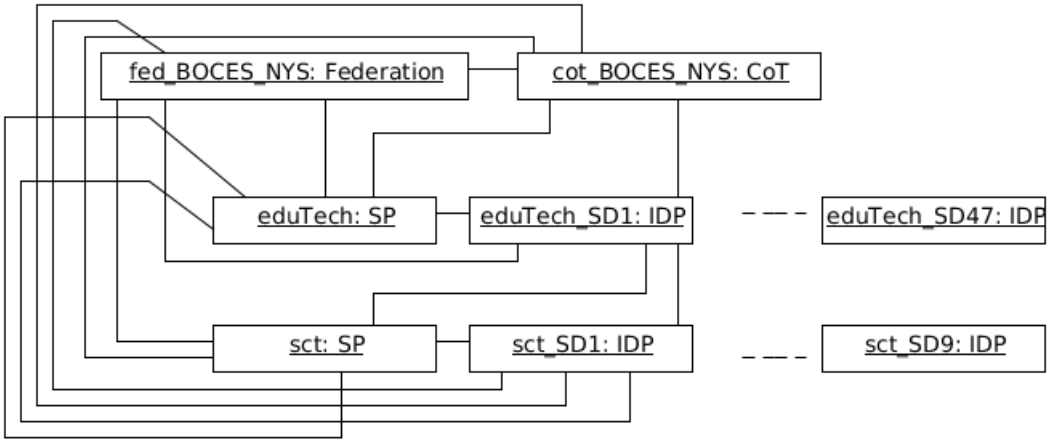


Figure 6.2: An object diagram for the reconfigured educational FIM

dashed line represents the last IDP within the list. In this way, we represent a few IDPs to demonstrate the configuration with respect to FIM pattern **MIF**. In the diagram, the SP and the IDPs are attached to each other and to their Federation `fed_BOCES_NYS` and CoT `cot_BOCES_NYS`. Also, the Federation is attached to the CoT. As a result, the configuration described via the object diagram in Figure 6.1 is created according to the class diagram given in Figure 3.1(b) (page 58).

Example 6.4 *The RIC known as* SCT *provides services to the teachers (and administrators) in 9 school districts. To deploy the extended FIM system, the scenario of Example 6.3 has been reconfigured to allow the users in the school districts federated to both RICs EduTech and SCT to access the services those RICs offer.*

Example 6.4 describes a reconfiguration scenario where the existing configuration of a FIM system has been reconfigured by introducing a RIC and several school districts. In this scenario, the RIC SCT is the SP and the school districts associated with the RIC can be considered as the IDPs. Notice that such a reconfiguration changes the underlying FIM pattern of the existing configuration. More precisely, the recon-

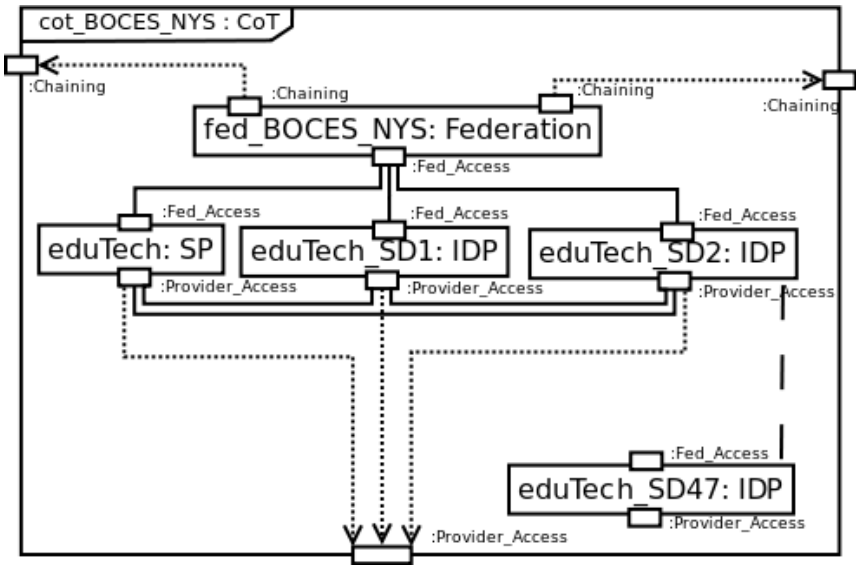


Figure 6.3: A composite structure diagram for the educational FIM

figuration changes the configuration of FIM pattern **MIF** described in Figure 6.1 to a configuration of FIM pattern **AF** where multiple SPs are federated to multiple IDPs. The object diagram in Figure 6.2 represents such an updated configuration where the SP `SCT` and the IDP `sct_SD1` are added to the configuration described by Figure 6.1. In Figure 6.2, for simplicity we do not represent all IDPs and also it does not have any impact on the underlying FIM pattern of the updated configuration. For instance, according to the reconfiguration scenario in Example 6.4 the updated configuration should conform to the FIM pattern **AF** where multiple SPs are federated to multiple IDPs. Figure 6.2 represents such a configuration.

The UML profile When using the profile in [33], we use instance level composite structure diagrams to represent FIM configurations. The instance level composite structure diagram in Figure 6.3 represents the configuration of the scenario described by Example 6.3. In this configuration, the component `fed_BOCES_NYS` of type `Federation` is attached to a provider `eduTech` of type `SP` and two providers `eduTech_SD1` and `eduTech_SD2` of type `IDP` via `Fed_Access` ports. (The component

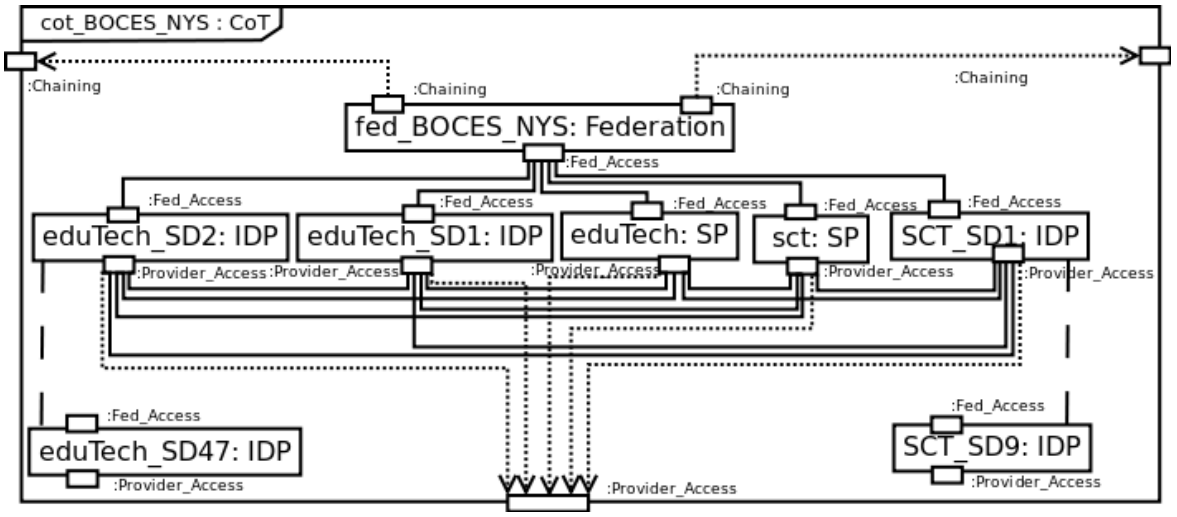
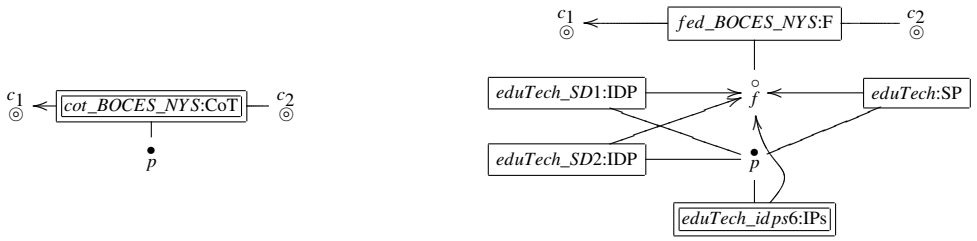


Figure 6.4: A composite structure for the reconfigured educational FIM

`eduTech_SD47` of type `IDP` represents the last IDP in the list) These providers are attached to each other via `Provider_Access` ports. The providers and the federation are attached to the enclosing component `cot_BOCES_NYS` of type `CoT` via `Provider_Access` and `Chaning` ports, respectively. The diagram in Figure 6.3 is created according to the class diagram of the production `Fed` given in Figure 4.1 (page 66).

Since the reconfiguration scenario described by Example 6.4 changes the underlying FIM pattern of the configuration from FIM pattern **MIF** to **AF**, we apply the reconfiguration rule `MIFtoAF` in Figure 4.5 (page 71) to such a scenario. Figure 6.4 shows the updated configuration where the SP `SCT` and the IDP `sct_SD1` (together with the rest of IDPs) are added to the configuration described by Figure 6.3. Recall that for simplicity we represent a few IDPs in the configurations described in UML.

ADR When using ADR, we use the productions given in Section 5.1.2 (page 79) to generate configuration of the scenario described by Example 6.3. To this purpose, the abstract CoT *cot_BOCES_NYS* in Figure 6.5(a) will be replaced with the configuration described in Figure 6.5(b). In Figure 6.5(b), the edge *eduTech* of Type SP and the edges *eduTech_SD*1 and *eduTech_SD*2 of type IDP are attached to each other via node *p*.



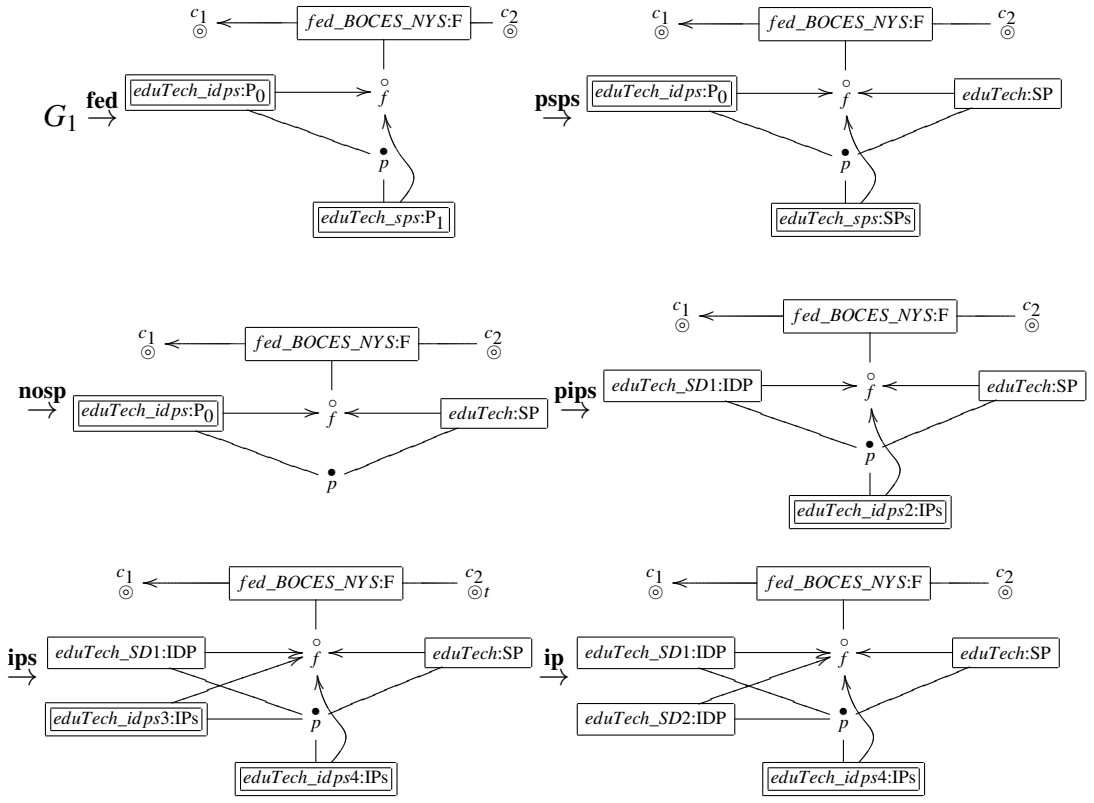
(a) Graph (G_1) describing the abstract educational FIM

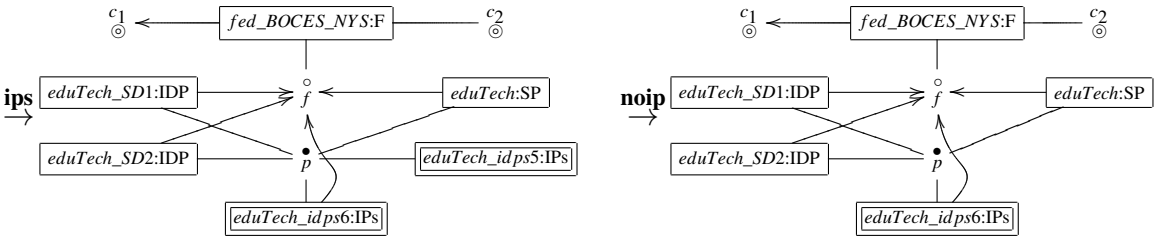
(b) Graph (G_2) describing the educational FIM

Figure 6.5: The graphs describing the educational FIM

Also, these edges are attached to the edge fed_BOCES_NYS of type F via node f.

To illustrate how the graph G_1 in Figure 6.5(a) is refined into the graph G_2 in Figure 6.5(b) consider the sequence of reductions





Namely, in the first step, **fed** (cf. Figure 5.2 on page 79) is applied to generate the federation fed_BOCES_NYS; then the edge *eduTech_sps* is refined by applying **psps** (cf. Figure 5.4 on page 81) yielding a provider *eduTech_sps* of type SPs and a provider *eduTech* of type SP. Since there is a single SP in the scenario of Example 6.3, the edge *eduTech_sps* is cancelled by applying **nosp**.

To generate configuration of the IDPs, the edge *eduTech_idps* of type P_0 is refined by applying **pips** (cf. Figure 5.3 on page 80) yielding a provider *eduTech_SD*1 of type IDP and a provider *eduTech_idps*2 of type IPs. The edge *eduTech_idps*2 is refined by applying **ips** yielding two provides *eduTech_idps*3 and *eduTech_idps*4 of type IPs. The edge *eduTech_idps*3 is refined by applying **ip** yielding a provider *eduTech_SD*2 of type IDP. The edge *eduTech_idps*4 is refined by applying **ips** yielding two providers *eduTech_idps*5 and *eduTech_idps*6 of type P_0. The edge *eduTech_idps*5 is cancelled by applying **noip**.

Observe that the existing configuration of the IDPs (i.e., edges *eduTech_SD*1 and *eduTech_SD*2 of type IDP) serves our purpose to demonstrate the underlying FIM pattern of the configuration described by the scenario in Example 6.3. However, the configuration of the rest of IDPs can be obtained by refining edge *eduTech_idps*6 by applying the corresponding productions in (cf. Figure 5.3 on page 80). In this way, any configuration x refining the edge *eduTech_idps*6 for the IDPs yields a term-like representation

$$\mathbf{fed}(\mathbf{psps}(\mathbf{nosp}),\mathbf{pips}(\mathbf{ips}(\mathbf{ips}(x,\mathbf{noip}),\mathbf{ip})))$$

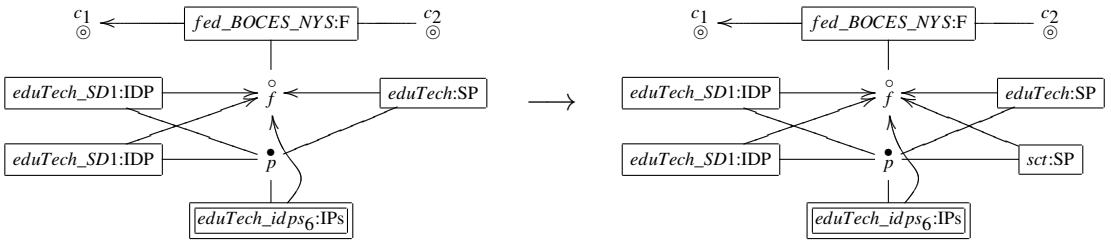


Figure 6.6: Rule to add the RIC SCT in the educational FIM (from left to right)

which precisely describes how configuration G_2 in Figure 6.5(b) is built.

To apply the reconfiguration described by Example 6.4 over G_2, we use two different reconfiguration rules of Section 5.3 (page 88) where one adds a single SP while the other adds a collection of IDPs at abstract level. In this connection, we apply the rule (5.4) (page 89) that adds a single SP. Figure 6.6 demonstrates such a reconfiguration where the SP *sct* of type SP is added to the configuration on the LHS and the transition

$$\mathbf{fed}(\mathbf{psps}(\mathbf{nosp}),\mathbf{pips}(\mathbf{ips}(\mathbf{ips}(x,\mathbf{noip}),\mathbf{ip}))) \longrightarrow$$
$$\mathbf{fed}(\mathbf{psps}(\mathbf{sps}(\mathbf{sp},\mathbf{nosp})),\mathbf{pips}(\mathbf{ips}(\mathbf{ips}(x,\mathbf{noip}),\mathbf{ip})))$$

describes the reconfiguration where subterm **nosp** of type SP on the LHS is replaced with a new term $\mathbf{sps}(\mathbf{sp},\mathbf{nosp})$ of same type on the RHS.

Since the reconfiguration scenario of Example 6.4 requires several IDPs to be added in the existing configuration, we use the rule (5.3) (page 89) to add the collection of IDPs at abstract level. Figure 6.7 demonstrates such a reconfiguration and the transition

$$\mathbf{fed}(\mathbf{psps}(\mathbf{sps}(\mathbf{sp},\mathbf{nosp})),\mathbf{pips}(\mathbf{ips}(\mathbf{ips}(x,\mathbf{noip}),\mathbf{ip}))) \longrightarrow$$
$$\mathbf{fed}(\mathbf{psps}(\mathbf{sps}(\mathbf{sp},\mathbf{nosp})),\mathbf{pips}(\mathbf{ips}(\mathbf{ips}(x,\mathbf{ips}(y,\mathbf{noip})),\mathbf{ip})))$$

describes the reconfiguration where subterm **noip** of type IPs on the LHS is replaced

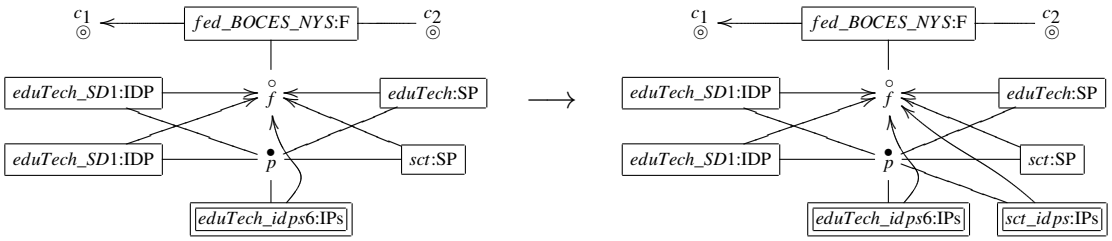


Figure 6.7: Rule to add all the school districts associated with the RIC SCT (from left to right)

with a new term $\mathbf{ips}(y, \mathbf{noip})$ of same type on the RHS. To generate the actual configuration of IDPs, the edge *sct_idps* of type IPs in Figure 6.7 will be refined by applying the corresponding productions (cf. Figure 5.3 on page 80).

Evaluation

In UML and the profile in [33], it was not feasible to represent the whole FIM configuration described by the scenario in Example 6.3. Therefore, we represent a few IDPs and the SP in the FIM configurations (cf. Figure 6.1 and Figure 6.4). We remark that such an approach not only allow us to demonstrate the simplified view of the FIM system but it also allow us to represent the underlying FIM pattern of the configuration described by the scenario. Since the FIM systems involve complex associations, we represent a few components in the FIM configuration represented by the object diagram (cf. Figure 6.1). However, it was difficult to create such a diagram to represent a FIM configuration. For instance, each provider in the FIM configuration has to be linked with the CoT, the federation, and the rest of providers[2]. In this case, introducing a few more components in the FIM configuration could hinder the design process. On the other hand, the composite structure diagram (cf. Figure 6.4) could effectively be manipulated while creating the FIM configuration. This is due to fact that such a diagram imposes a structure over the CoT in FIM and it also simplifies interconnections

[2]For simplicity, we do not represent the DTMS components associated with the providers.

between the FIM components via attaching their ports.

In ADR, productions allowed us to precisely generate the FIM configuration. To highlight the underlying FIM pattern of the configuration, we represent a few IDPs and the SP in ADR (cf. Figure 6.5(b)) in the same way as we described these components in UML and the profile. Furthermore, ADR allowed us to abstract away the irrelevant details in the FIM configuration. For instance, it was not necessary to represent all IDPs where a few IDPs were sufficient to highlight the underlying FIM pattern of the configuration. In this case, the notion of non-terminal edges (i.e., $eduTech\_idps6$ in Figure 6.5(b)) in ADR allow to represent the rest of IDPs at abstract level where such an edge could be replaced later with the actual configuration of IDPs by using the corresponding productions.

Since UML does not provide any mechanism to describe the reconfigurations, we directly introduce the changes in the configuration of Figure 6.1. Due to complex association between the FIM components, it is difficult to add such components in the configuration. For instance, consider the object diagram in Figure 6.2 where the SP `sct` and the IDP `sct_SD1` are added to the object diagram in Figure 6.1. In this case, we first introduce the SP and then we introduce the IDP in the updated configuration. Observe that, for our convenience (and readability), we do not represent the IDP `eduTech_SD2` in the updated configuration in order to represent the additional IDP `sct_SD1`.

While using the UML profile in [33], we found the application of the rule `MIFtoAF` described in Figure 4.5 (page 71) which transforms the underlying FIM pattern of the configuration. We remark that the profile has the similar challenges as with the object diagrams in UML while introducing components into the configurations described by instance level composite structure diagrams. For instance, to apply the reconfiguration scenario in Example 6.4 we updated the composite structure diagram in Figure 6.3 in the same way as the object diagram in UML. However, we were able to effectively manipulate such a diagram since it imposes a structure over the CoT `cot_BOCES_NYS`

where UML connectors as compared to links connecting the FIM components in object diagrams simplify the interconnections between such components.

While using ADR, we apply the two reconfigurations rules where one add a single SP and the other adds a collection of IDPs. The reconfiguration described by Example 6.4 changes one FIM pattern to the other by introducing an additional SP and several IDPs. Such a change is actually triggered by merely introducing the SP in the existing configuration. To this purpose, we initially apply the rule (5.4) (page 89) which precisely guided us to realise such a change in the configuration. Finally, we apply the complex rule (5.3) (page 5.3) in ADR that allowed us to add several IDPs in one go at an abstract level.

6.6 Tool Support

In this section, we provide a description of the availability of tool support for UML, the profile in [33], and ADR.

UML To describe the FIM style in UML, we use the structural diagrams (i.e., class diagrams). In addition to multiplicity constraints, we also define a few constraints in OCL to restrict the valid models of FIMs. In this connection, we discus the UML tools that allow one to describe such a system. There are several free/open-source (or non-commercial) and commercial tools that have been developed to support various kinds of UML diagrams (cf. [13] and [14]). Also, there are at least 15 tools which support OCL where each implementing distinct features (cf. [46] for survey). However, a few UML tools provide the support for OCL.

Among the most widely used commercial UML tools, Together [23] provides satisfactory support for OCL while MagicDraw [12], Rose [18], Tau [19], and Poseidon [17] lack the similar support for OCL [46]. More precisely, Together provides

OCL support both at metamodel level (e.g., class diagrams describing the style) and model level (i.e., object diagrams describing the configurations).

In the non-commercial UML tools, ArgoUML [9], OCLE [15], and USE (UML Specification Environment) [8] provide OCL support [46]. In these tools, ArgoUML and OCLE are the visual modelling UML tools while USE supports textual description of the UML models. Furthermore, ArgoUML integrates the Dresden OCL Toolkit [10]. Due to continues improvements in the Dresden OCL Toolkit, the toolkit has widely been adopted by the modelling frameworks (e.g., Eclipse Modelling Framework) to provide OCL support in their modelling languages (e.g., EMF/ECORE). However, ArgoUML supports the UML notations defined in the standard UML 1.4 version and it also provides limited support for the profiles (i.e., it supports only the profile provided with tool). Similarly, OCLE is compliant with UML version 1.5. As a result, the UML models (i.e., composite structure diagrams) which use UML 2 features (e.g., ports) can not be described while using such tools. To the best of our knowledge, among the non-commercial tools OCLE is the only visual modelling UML tool which supports modelling of the software systems both at metamodel level and model level. According to Gogolla et al. [57], USE [8] is one of the first CASE tool that supports OCL. It allows validation of UML models with OCL constraints defined over them. Such models are textually described via the USE specifications. Interestingly, USE allows one to create object diagrams to represent snapshots showing system states. Typically, such snapshots are created via an explicit sequence of commands. To avoid this, in [57] a declarative language called *A Snapshot Sequence Language* (ASSL) is given to enable the construction of snapshots (i.e., FIM configurations to highlight the patterns) in an automated way.

The UML profile While using the profile in [33], composite structure diagrams (with ports attached)[3] are used to describe the FIM style. Also, OCL constraints have been defined over these diagrams within the style. Such a style requires the profile specific support to be provided in the UML tools. For instance, in Figure 4.1 (page 66) the «`refineable`» components need to be replaced with the configurations described via the «`production`» components. To the best of our knowledge, currently neither any commercial UML tool nor any free/open-source UML tool is available that supports such a refinement process. However, a few commercial UML 2 compliant tools (e.g., Together, Rose, Tau, etc.) can suitably be used to layout the composite structure diagrams with OCL constraints defined over them.

Since the formal semantics of the UML profile in [33] has been defined in terms of ADR, one may use the tool support introduced in [35] (described later) for ADR. In order to do this, one has to represent the same style in ADR as described using the profile. In this way, the prototypical tool support for ADR given in [35] can potentially be exploited for the UML profile in [33].

ADR In [35], a prototypical tool support for ADR is provided using Maude. Also, a methodology is given in [35] to implement ADR-based specifications in Maude. In [35], an implementation of graphs is given in Maude via a couple of functional modules in order to represent low level language of ADR. Furthermore, an ADR language has been defined in terms of hierarchical designs. The ADR language can be used at two different levels namely, symbolic level and interpreted level. At the symbolic level, the ADR language requires one to define the signature of the style (i.e., defining sorts and signature of various operations). At the interpreted level, the interpretation of the abstract view in hierarchical design algebra is needed. To this purpose, a module defines the type graph which represents the style vocabulary and a *constant* for each design

[3]These diagrams use the features provided by the latest version of UML i.e., UML 2

production. The constant consists of a design that corresponds to the design production. Also, an interpreted version is defined for each operation. Furthermore, the tool in [35] allows one to implement the module to support the refinement process in ADR by giving a rewrite theory to simulate such a process. To support the visual representation of the designs in ADR and debugging activities, the tool in [35] implements modules to export various graphical formats (e.g., dot [54] and GraphML [32]) which enables designers to use the graphical tools (e.g., Graphviz [54] and yEd [24]). For instance, the visual tool yEd is used in [35] to suitably layout the hierarchical designs.

6.7 Summary

In this section, we summarise the comparison and we also draw some conclusions over it. For the comparison, we considered the approaches including UML, the profile in [33], ADR, the graph-based approaches in [73, 74, 61, 27], and two ADLs (*C2SADEL* [69] and *Acme* [56]).

In order to compare the approaches, we fixed and described certain criteria pertaining to general and pattern specific architectural aspects of FIM systems. In architectural aspects, we consider core architectural concepts, styles, style checking, reconfiguration, and refinement. In pattern specific criteria, we consider pattern generation and pattern identification. Once the criteria for the comparison are described, we discuss the support provided by each approach for those criteria. Based on such an assessment, we compare the approaches against those criteria. We outline the comparison against the general criteria as follows:

- **Core concepts:** The approaches (except C2SADEL) suitably describe core concepts. C2SADEL supports a particular kind of components (i.e., in C2 style).

- **Style:** Vocabulary of the style is well supported by all approaches. Similarly, the

approaches other than UML, the profile in [33] and [27] support style constraints. UML, the profile in [33], and [27] requires constraints to be defined in OCL. Such constraints are often not easy to express.

- **Style checking:** the ADLs' run-time systems perform type checking. Consequently, one may further require the use of certain analysis tools to perform style checking in the ADLs. Similarly, UML and [27] require the use of certain validation tools and techniques. However, this could be problematic due to lack of the support available for OCL in the widely available UML tools. On the other hand, ADR and the approaches of [61] and [74] use a formal mechanism that guarantees construction of valid configurations.

- **Reconfigurations:** UML does not provide any mechanism to describe reconfigurations. The profile in [33] provides unsatisfactory support to describe reconfigurations in a general way. The ADLs allow a single component (or connector) to be added at a time. The rest of approaches suitably describe reconfigurations where they (except [61]) also provide the means to check whether the style is preserved or not.

- **Refinement:** UML does not support the notion of abstract architectural components which could be replaced with the detailed designs. The approach in [27] provides limited support for such a refinement process where it defines one-to-one structural mapping between the elements of two styles. C2SADEL also provides limited support such as component sub-typing to deal with architectural refinement. On the other hand, the profile in [33], ADR, Acme and the approaches in [74, 61] suitably describe architectural refinement.

We remark that the graph based approaches including ADR and the approach of [74] can be singled out for their support to pattern specific criteria where they

- provide a formal mechanism to generate the configurations of the FIM patterns in a precise controlled way
- and allow designers to effectively parse the formal description associated with the configurations to identify their underlying FIM patterns.

On the other hand, the rest of approaches clearly lack such mechanisms. For the comparison, we also apply UML, the profile [33], and ADR to a case study which describes a real world FIM scenario. We mainly show that these approaches can be used to create the initial configurations with respect to pattern then we apply the changes to these configurations according to the reconfiguration scenario.

Finally, we discuss the available tool support for UML, the profile in [33], and ADR. In particular, we discus a few free/open-source and commercial UML tools which provides the support for OCL. To the best of our knowledge, currently there is no free/open-source UML tool (except a few commercial UML tools) available that provides such a support for the UML model (i.e., composite structure diagram with ports) which uses features of the latest version of UML. Also, a UML tool needs to be developed (or extended) to provide the features (e.g., refinement) specific to the profile in [33]. However, one may potentially use the prototypical tools support of ADR. This is because of the formal semantics of the profile are defined in terms of ADR. We also discuss the prototypical tools support of ADR.

Chapter 7

Conclusions and Future Work

In this chapter, we draw some conclusions over the modelling approaches used to describe the models of FIMs proposed in this thesis. Finally, we describe possible future research directions.

7.1 Modelling FIMs

In this thesis, architectural and reconfigurations aspects of FIMs have been modelled in UML and ADR. More precisely, we introduced architectural styles for FIMs which characterise a few "patterns" which have informally been described in [66]. To this purpose, we developed FIM styles by considering:

- structural diagrams in standard UML where a class diagram with a few OCL constraints models the FIM style while object/structure diagrams represent FIM configurations,
- the UML profile in [33] to support architectural refinement and reconfigurations for FIMs where refinement of abstract FIM architecture are possible, and
- ADR to describe a formal model of FIMs where a type graph with a few produc-

tions suitably represent the FIM style while the reconfiguration rules precisely describe the architectural changes for FIMs in a general way.

Our research goal was to investigate

How can architectural description of FIM systems be exploited to model and analyse threats associated with structural configurations?

The direct answer to this research question (repeated here from page 8 for the convenience of readers) is that the use of architectural styles helps in formally representing patterns of FIM systems to study their configurations and some security threats associated with them.

Each of the FIM styles introduced in this dissertation (cf. Chapter 3, 4, and 5) characterises architectural configurations of the FIM patterns in a general way. In other words, the FIM patterns given in Section 2.2 are described under a single style. Since a FIM configuration may change (i.e., add/remove a FIM component) during the development life cycle, this may result into changing or preserving its underlying FIM pattern. Therefore, describing a generic style for FIM systems allowed us to define such reconfigurations. In addition to this, the support provided by ADLs for pattern specific criteria (detailed in Section 6.1.2), which includes *pattern generation* and *pattern identification*, is crucial for FIM systems. Any configuration created according to one of the FIM models proposed in this thesis will belong to a particular pattern. More precisely, we refer the process of instantiating configurations of FIM systems as pattern generation. In particular, such a process can effectively be used by designers when a specific FIM pattern is chosen (together with a particular mechanism to deal with its associated threats).

While using the proposed UML models of FIM systems, the support for pattern generation is not ideal since designers have to rely on rules of thumb to instantiate the configurations and UML tools to validate them. On the other hand, ADR design

rules defined in the formal model of FIM allow designers to precisely instantiate valid configurations of interest.

Since the proposed FIM styles model FIM patterns which have different security requirements and are exposed to different threats, applying changes to the configurations created with respect to those styles may impact on their security requirements and their associated threats too. More precisely, an application of the reconfiguration may change the underlying FIM pattern of the configuration. For instance, one may need to add an SP into an existing configuration of FIM pattern **MIF**. While allowing such a reconfiguration, this will result into an updated configuration which now conforms to FIM pattern **AF**. In this way, underlying FIM pattern of the configurations may change from one pattern to the other. Therefore, the support for pattern identification can be beneficial in order to identify the underlying FIM pattern of the updated configuration. This may enable designers to control the threats associated with the updated configuration (i.e., via realising a given security mechanism). For identifying the underlying FIM patterns, the configurations of FIM systems need to be analysed by enumerating instances of particular type (i.e., SP and IDP). In order to do this, UML instance level (object or structure) diagrams need to be analysed. On the other hand, ADR terms can effectively be parsed to identify the underlying patterns of the configurations.

UML has recently been promoted as an ADL [71]. For instance, in [33] it is shown how an extension of UML enables the modelling of architectural aspects of service-oriented applications. The use of UML as an ADL appears natural as some UML diagrams can express aspects of software architectures. For instance, class diagrams provide a reasonably suitable language to represent architectural elements and their interconnections.

Our main result is a critical analysis of the usage of UML as ADL conducted by modelling architectural aspects of FIMs. We argue that

- despite the fact that some UML diagrams (e.g., class/structure diagrams) can be used to model software architectures, overall UML carries a heavy overhead as most of the architectural information is scattered across many (loosely related) diagrams.

- The UML designer is typically forced to use complex features of UML to model architectural styles; for instance, even for straightforward conditions on the association relations of some class diagrams, the designer has to introduce OCL constraints.

- It is very hard to precisely characterise an architectural style; in fact, styles can intuitively be thought of as "types" that can be assigned to architectures; UML fails to precisely characterise such styles and often UML models over- or under-specify the classes of architectures of interest.

Our contention is that ad-hoc architectural design languages are more suitable than UML when it comes to modelling architectural aspects of systems. For instance, many ad-hoc ADLs have been proposed (see [72] for a survey). We consider ADR which features a mathematically rigorous style preserving modelling approach in terms of suitable graphs, whose edges model components and whose nodes model the ports through which components are connected. Architectural styles are seen in ADR as a hierarchy over the architecture based on an algebraic presentation of style-based design. In fact, ADR design rules (or productions) correspond to basic operations for the typed composition of architectures. Such operations can be applied to graphs (representing architectures) to transform them while preserving their types (i.e., their style). ADR uses graph transformations to formally model architectural styles (e.g., [86, 53]) and it features a precise notion of correctness.

7.2 Future Work

Currently, ADR provides a mechanism to describe design-time reconfigurations namely, changes in a software architecture introduced by designers. Since ADR is a graph-based approach to model style-based reconfigurable software systems, graphs in ADR can potentially represent behaviour of a system where graph rewrites can be used to model execution of the system [34]. In this regard, we intend to investigate the approach in [67] on how to describe the changes triggered by the system at run time, for instance, when providers leave/join a CoT (e.g., in a Cloud) based on its overall reputation, or, when the federation component governing the CoT forces a provider to leave the CoT if the provider violates certain rules of cooperation (e.g., a service level agreement in service-oriented computing). In this connection, we refer to these changes as *behavioural reconfigurations* in FIM systems. We believe that such kind of changes may potentially be realised via implementing a dynamic security and trust management component in a FIM system. Therefore, a major aspect to be tackled in future work is the extension of our ADR model to represent behavioural reconfigurations of FIMs. The most challenging facet of this research is to give a new model were behavioural reconfigurations preserve patterns or, when this is not possible, they happen in a ”controlled” way so that designers and architects are aware of the security threats a reconfiguration could introduce.

The formal model given in Chapter 5 has been designed by considering structural requirements of FIMs (and structural reconfigurations). That model has two main drawbacks which we plan to address in future research.

The first drawback we plan to address is the fact that the model does not consider *users* and *services*. Specifically, we intend to model users at a suitable level of abstraction. For instance, a user may encapsulate information related to one of several kinds of users (i.e., human-users, services, applications, etc.) that can invoke services

at the SPs within and across the CoTs in a FIM system. We believe that such an extension to the model is crucial to represent behavioural aspects of FIMs. To this purpose, we propose to investigate in the future how to formalise these aspects in ADR (e.g., via formalising one of the existing FIM specifications [3, 2]). In this way, a formally described executable FIM architecture could emerge. The obvious benefit of such an architecture is that it may provide the basis for realising a dynamic CoT in the FIMs. For instance, a dynamic CoT may enable providers to join up and leave the FIMs at run time. In order to do so, it is worth considering some evolutions to the FIM components in the future. For instance, a dynamic CoT may be developed by considering a detailed design of an underlying DTMS (after dynamic trust management system).

The second drawback of the model is due to a limitation of the ADR theory and it concerns a technical issue. In fact, we intend to work on a few enhancements to ADR in order to provide the support for describing (*i*) sub-types and (*ii*) mechanisms to specify generic (or meta) productions. Below, we comment on those lines of research.

(*i*) Sub-types. We believe that component sub-types will certainly be beneficial to achieve polymorphic behaviour in the FIM components while producing executable FIM architectures in ADR. For instance, two component (sub-)types IDP and SP can be related to a component of (super-)type provider. This may allow us to realise a possible common behaviour of the providers in FIMs. Similarly, a (super-)type of user in a FIM system can be specialised via more specific classes of users. Furthermore, component sub-typing in ADR may be beneficial to deal with reconfiguration in a general way. For instance, this may help in migrating one or more particular IDPs and SPs from one CoT to the other in a complex FIM system in a single step. Similar reconfiguration rules can be defined for the CoTs.

(*ii*) Generic (or meta-)productions. We believe that once support for sub-types in ADR is provided one may potentially exploit such a concept to describe generic productions. Observing the model given in Chapter 5, one can notice that productions for

IDP and SP are quite similar. Therefore, it would be rather helpful to specify generic productions which may be instantiated to generate both of these FIM components. This would enable us to substantially simplify our model and, as a result, be able to efficiently maintain the designs during the development life-cycle of FIMs.

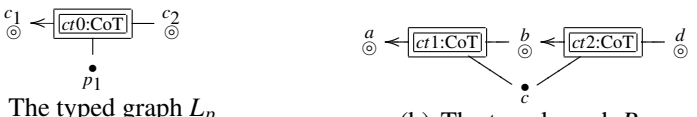
Appendix A

Formal Definitions of the Productions

In this section, we give the formal definitions for all of the design productions given in Section 5.1.2. The type graph (named as H) depicted in Figure 5.1 (page 78), which represents the architectural elements of FIMs, together with these productions describe architectural style for FIMs. The formal definitions of the productions are given as follows:

A.1 Productions for *CoT*

Figure 5.2 (page 79) shows productions **fed** and **chain** that can be used to generate a federation of providers and a chain of CoT respectively.



(a) The typed graph L_p (b) The typed graph R_p

Figure A.1: LHS and RHS graphs of production **chain** typed over the graph H

The production *chain* The morphism between the LHS typed graph L_p (Figure A.1(a)) of the production **chain** (Figure 5.2 on page 79) and the type graph H is given below;

where p is **chain**

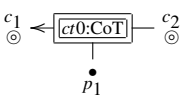
$$f_{V_{L_p}} : \begin{cases} c_1 \longmapsto \circledcirc \\ c_2 \longmapsto \circledcirc \\ p_1 \longmapsto \bullet \end{cases} \qquad f_{E_{L_p}} : ct0 \longmapsto CoT$$

Similarly, the morphism between the RHS typed graph R_p (Figure A.1(b)) of the production **chain** and the type graph H is given below; where p is **chain**

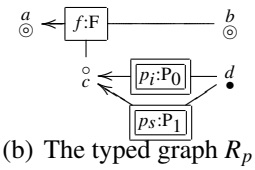
$$f_{V_{R_p}} : \begin{cases} a \longmapsto \circledcirc \\ b \longmapsto \circledcirc \\ d \longmapsto \circledcirc \\ c \longmapsto \bullet \end{cases} \qquad f_{E_{R_p}} : \begin{cases} ct1 \longmapsto CoT \\ ct2 \longmapsto CoT \end{cases}$$

In production **chain**, function i_p maps the interface nodes of L_p (Figure A.1(a)) with the interface nodes of R_p (Figure A.1(b)) and function l_p is the bijective mapping of the non-terminals in the R_p on an initial segment $[1,2,...,n_p]$; where p is **chain**

$$i_p : \begin{cases} c_1 \longmapsto a \\ c_2 \longmapsto d \\ p_1 \longmapsto c \end{cases} \qquad l_p : \begin{cases} ct1 \longmapsto 1 \\ ct2 \longmapsto 2 \end{cases}$$



(a) The typed graph L_p



(b) The typed graph R_p

Figure A.2: LHS and RHS graphs of production **fed** typed over the graph H

The production *fed* The morphism between the LHS typed graph L_p (Figure A.2(a)) of the production **fed** (Figure 5.2 on page 79) and the type graph H is given below;

where p is **fed**

$$f_{V_{L_p}} : \begin{cases} c_1 \longmapsto \circledcirc \\ c_2 \longmapsto \circledcirc \\ p_1 \longmapsto \bullet \end{cases} \qquad f_{E_{L_p}} : ct0 \longmapsto CoT$$

Similarly, the morphism between the RHS typed graph R_p (Figure A.2(b)) of the production **fed** and the type graph H is given below; where p is **fed**

$$f_{V_{R_p}} : \begin{cases} a \longmapsto \circledcirc \\ b \longmapsto \circledcirc \\ c \longmapsto \circ \\ d \longmapsto \bullet \end{cases} \qquad f_{E_{R_p}} : \begin{cases} f \longmapsto \mathrm{F} \\ p_i \longmapsto \mathrm{P}_0 \\ p_s \longmapsto \mathrm{P}_1 \end{cases}$$

In production **fed**, function i_p maps the interface nodes of L_p (Figure A.2(a)) with the interface nodes of R_p (Figure A.2(b)) and function l_p is the bijective mapping of the non-terminals in the R_p on an initial segment $[1,2,...,n_p]$; where p is **fed**

$$i_p : \begin{cases} c_1 \longmapsto a \\ c_2 \longmapsto b \\ p_1 \longmapsto d \end{cases} \qquad l_p : \begin{cases} p_i \longmapsto 1 \\ p_s \longmapsto 2 \end{cases}$$

A.2 Productions for identity providers

Now, we define the productions **pips**, **ips**, **ip**, and **noip** given in Figure 5.3 (page 80) that can be used to generate identity providers in the CoT.

(a) The typed graph L_p (b) The typed graph R_p

Figure A.3: LHS and RHS graphs of production **pips** typed over the graph H

The production *pips* The morphism between the LHS typed graph L_p (Figure A.3(a)) of the production **pips** (Figure 5.3 on page 80) and the type graph H is given below; where p is **pips**

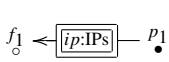
$$f_{V_{L_p}} : \begin{cases} f_1 \longmapsto \circ \\ p_1 \longmapsto \bullet \end{cases} \qquad f_{E_{L_p}} : p0 \longmapsto P_0$$

Similarly, the morphism between the RHS typed graph R_p (Figure A.3(b)) of the production **pips** and the type graph H is given below; where p is **pips**

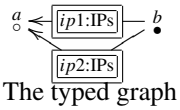
$$f_{V_{R_p}} : \begin{cases} a \longmapsto \circ \\ b \longmapsto \bullet \end{cases} \qquad f_{E_{R_p}} : \begin{cases} pi \longmapsto IPs \\ i \longmapsto IDP \end{cases}$$

In production **pips**, function i_p maps the interface nodes of L_p with the interface nodes of R_p and function l_p is the bijective mapping of the non-terminals in the R_p on an initial segment $[1,2,...,n_p]$; where p is **pips**

$$i_p : \begin{cases} f_1 \longmapsto a \\ p_1 \longmapsto b \end{cases} \qquad l_p : pi \longmapsto 1$$



(a) The typed graph L_p



(b) The typed graph R_p

Figure A.4: LHS and RHS graphs of production **ips** typed over the graph H

The production *ips* The morphism between the LHS typed graph L_p (Figure A.4(a)) of the production **ips** (Figure 5.3 on page 80) and the type graph H is given below;

where p is **ips**

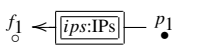
$$f_{V_{L_p}} : \begin{cases} f_1 \longmapsto \circ \\ p_1 \longmapsto \bullet \end{cases} \qquad f_{E_{L_p}} : ip \longmapsto IPs$$

Similarly, the morphism between the RHS typed graph R_p (Figure A.4(b)) of the production **ips** and the type graph H is given below; where p is **ips**

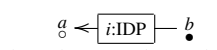
$$f_{V_{R_p}} : \begin{cases} a \longmapsto \circ \\ b \longmapsto \bullet \end{cases} \qquad f_{E_{R_p}} : \begin{cases} ip1 \longmapsto IPs \\ ip2 \longmapsto IPs \end{cases}$$

In production **ips**, function i_p maps the interface nodes of L_p with the interface nodes of R_p and function l_p is the bijective mapping of the non-terminals in the R_p on an initial segment $[1,2,...,n_p]$; where p is **ips**

$$i_p : \begin{cases} f_1 \longmapsto a \\ p_1 \longmapsto b \end{cases} \qquad l_p : \begin{cases} ip1 \longmapsto 1 \\ ip2 \longmapsto 2 \end{cases}$$



(a) The typed graph L_p



(b) The typed graph R_p

Figure A.5: LHS and RHS graphs of production **ip** typed over the graph H

The production *ip* The morphism between the LHS typed graph L_p (Figure A.5(a)) of the production **ip** (Figure 5.3 on page 80) and the type graph H is given below; where p is **ip**

$$f_{V_{L_p}} : \begin{cases} f_1 \longmapsto \circ \\ p_1 \longmapsto \bullet \end{cases} \qquad f_{E_{L_p}} : ips \longmapsto IPs$$

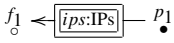
Similarly, the morphism between the RHS typed graph R_p (Figure A.5(b)) of the production **ip** and the type graph H is given below; where p is **ip**

$$f_{V_{R_p}} : \begin{cases} a \longmapsto \circ \\ b \longmapsto \bullet \end{cases} \qquad f_{E_{R_p}} : \{\ i \longmapsto IDP$$

The function i_p in the production **ip** maps the interface nodes of L_p with the interface nodes of R_p; where p is **ip**

$$i_p : \begin{cases} f_1 \longmapsto a \\ p_1 \longmapsto b \end{cases}$$

Since there is no non-terminal in the production **ip**, the function l_p in the production does not represent the mapping.



(a) The typed graph L_p

(b) The typed graph R_p

Figure A.6: LHS and RHS graphs of production **noip** typed over the graph H

The production *noip* The morphism between the LHS typed graph L_p (Figure A.6(a)) of the production **noip** (Figure 5.3 on page 80) and the type graph H is given below; where p is **noip**

$$f_{V_{L_p}} : \begin{cases} f_1 \longmapsto \circ \\ p_1 \longmapsto \bullet \end{cases} \qquad f_{E_{L_p}} : ips \longmapsto IPs$$

Similarly, the morphism between the RHS typed graph R_p (Figure A.6(b)) of the production **noip** and the type graph H is given below; where p is **noip**

$$f_{V_{R_p}} : \begin{cases} a \longmapsto \circ \\ b \longmapsto \bullet \end{cases}$$

The function i_p in production **noip** maps the interface nodes of L_p with the interface nodes of R_p; where p is **noip**

$$i_p : \begin{cases} f_1 \longmapsto a \\ p_1 \longmapsto b \end{cases}$$

Since there is no non-terminal in the production **noip**, the function l_p in the production does not represent the mapping.

A.3 Productions for service providers

Figure 5.4 (page 81) shows the productions **psps**, **sps**, **sp**, and **nosp** that can be used to generate service providers in the CoT.

(a) The typed graph L_p (b) The typed graph R_p

Figure A.7: LHS and RHS graphs of production **psps** typed over the graph H

The production *psps* The morphism between the LHS typed graph L_p (Figure A.7(a)) of the production **psps** (Figure 5.4 on page 81) and the type graph H is given below;

where p is **psps**

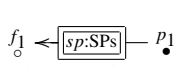
$$f_{V_{L_p}} : \begin{cases} f_1 \longmapsto \circ \\ p_1 \longmapsto \bullet \end{cases} \qquad f_{E_{L_p}} : p1 \longmapsto P_1$$

Similarly, the morphism between the RHS typed graph R_p (Figure A.7(b)) of the production **psps** and the type graph H is given below; where p is **psps**

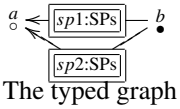
$$f_{V_{R_p}} : \begin{cases} a \longmapsto \circ \\ b \longmapsto \bullet \end{cases} \qquad f_{E_{R_p}} : \begin{cases} ps \longmapsto SPs \\ s \longmapsto SP \end{cases}$$

In production **psps**, function i_p maps the interface nodes of L_p with the interface nodes of R_p and function l_p is the bijective mapping of the non-terminals in the R_p on an initial segment $[1,2,...,n_p]$; where p is **psps**

$$i_p : \begin{cases} f_1 \longmapsto a \\ p_1 \longmapsto b \end{cases} \qquad l_p : ps \longmapsto 1$$



(a) The typed graph L_p



(b) The typed graph R_p

Figure A.8: LHS and RHS graphs of production **sps** typed over the graph H

The production *sps* The morphism between the LHS typed graph L_p (Figure A.8(a)) of the production **sps** (Figure 5.4 on page 81) and the type graph H is given below; where p is **sps**

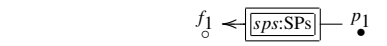
$$f_{V_{L_p}} : \begin{cases} f_1 \longmapsto \circ \\ p_1 \longmapsto \bullet \end{cases} \qquad f_{E_{L_p}} : sp \longmapsto SPs$$

Similarly, the morphism between the RHS typed graph R_p (Figure A.8(b)) of the production **sps** and the type graph H is given below; where p is **sps**

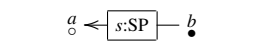
$$f_{V_{R_p}} : \begin{cases} a \longmapsto \circ \\ b \longmapsto \bullet \end{cases} \qquad f_{E_{R_p}} : \begin{cases} sp1 \longmapsto SPs \\ sp1 \longmapsto SPs \end{cases}$$

In production **sps**, function i_p maps the interface nodes of L_p with the interface nodes of R_p and function l_p is the bijective mapping of the non-terminals in the R_p on an initial segment $[1,2,...,n_p]$; where p is **sps**

$$i_p : \begin{cases} f_1 \longmapsto a \\ p_1 \longmapsto b \end{cases} \qquad l_p : \begin{cases} sp1 \longmapsto 1 \\ sp2 \longmapsto 2 \end{cases}$$



(a) The typed graph L_p



(b) The typed graph R_p

Figure A.9: LHS and RHS graphs of production **sp** typed over the graph H

The production *sp* The morphism between the LHS typed graph L_p (Figure A.9(a)) of the production **sp** (Figure 5.4 on page 81) and the type graph H is given below; where p is **sp**

$$f_{V_{L_p}} : \begin{cases} f_1 \longmapsto \circ \\ p_1 \longmapsto \bullet \end{cases} \qquad f_{E_{L_p}} : sps \longmapsto SPs$$

Similarly, the morphism between the RHS typed graph R_p (Figure A.9(b)) of the production **sp** and the type graph H is given below; where p is **sp**

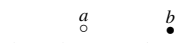
$$f_{V_{R_p}} : \begin{cases} a \longmapsto \circ \\ b \longmapsto \bullet \end{cases} \qquad f_{E_{R_p}} : \{ s \longmapsto SP$$

The function i_p of the production **sp** maps the interface nodes of L_p with the interface nodes of R_p; where p is **sp**

$$i_p : \begin{cases} f_1 \longmapsto a \\ p_1 \longmapsto b \end{cases}$$

Since there is no non-terminal in the production **sp**, the function l_p does not represent the mapping.

(a) The typed graph L_p



(b) The typed graph R_p

Figure A.10: LHS and RHS graphs of production **nosp** typed over the graph H

The production *nosp* The morphism between the LHS typed graph L_p (Figure A.10(a)) of the production **nosp** (Figure 5.4 on page 81) and the type graph H is given below; where p is **nosp**

$$f_{V_{L_p}} : \begin{cases} f_1 \longmapsto \circ \\ p_1 \longmapsto \bullet \end{cases} \qquad f_{E_{L_p}} : sps \longmapsto SPs$$

Similarly, the morphism between the RHS typed graph R_p (Figure A.10(b)) of the production **nosp** and the type graph H is given below; where p is **nosp**

$$f_{V_{R_p}} : \begin{cases} a \longmapsto \circ \\ b \longmapsto \bullet \end{cases}$$

The function i_p maps the interface nodes of L_p with the interface nodes of R_p; where p is **nosp**

$$i_p : \begin{cases} f_1 \longmapsto a \\ p_1 \longmapsto b \end{cases}$$

Since there is no non-terminal in the production **nosp**, the function l_p in the production does not represent the mapping.

Bibliography

[1] Kerberos: The Network Authentication Protocol. http://web.mit.edu/Kerberos/.

[2] Case study: EduTech Deploys Federated Identity for Maximum Impact. http://www.projectliberty.org/liberty/content/download/2458/15859/file/EduTech-CaseStudy.pdf, 2006.

[3] Case Study: Federation of the Danish Public Sector. http://projectliberty.org/liberty/content/download/4301/28788/file/denmark_libertycasestudy6.08.pdf, 2007.

[4] Case Study: GM Drives Federated Identity. http://projectliberty.org/liberty/content/download/418/2817/file/gm_MARCH07.pdf, 2007.

[5] Security assertion markup language (saml) specifications. http://saml.xml.org/saml-specifications, 2009.

[6] Ws-federation specification. http://www.ibm.com/developerworks/library/specification/ws-fed/, 2009.

[7] Object Constraint Language (version 2.2). http://www.omg.org/spec/OCL/2.2/PDF, 2010.

[8] A UML-based Specification Environment (USE). http://www.db.informatik.uni-bremen.de/projects/USE/, 2011.

[9] ArgoUML. http://argouml.tigris.org/, 2011.

[10] Dresden OCL Toolkit. http://www.dresden-ocl.org/index.php/DresdenOCL, 2011.

[11] Liberty Alliance Project: Case Studies. http://www.projectliberty.org/liberty/resource_center/case_studies/, 2011.

[12] MagicDraw. https://www.magicdraw.com/, 2011.

[13] Object by Design. http://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools, 2011.

[14] Object by Design. http://www.objectsbydesign.com/tools/umltools_byCompany.html, 2011.

[15] OCLE. http://lci.cs.ubbcluj.ro/ocle/index.htm, 2011.

[16] OpenID. http://openid.net/, 2011.

[17] Poseidon for UML. http://www.gentleware.com/products.html, 2011.

[18] Rational Rose. http://www-01.ibm.com/software/awdtools/developer/rose/java/index.html, 2011.

[19] Rational Tau. http://www-01.ibm.com/software/awdtools/tau/, 2011.

[20] Saml v2.0 executive overview. http://www.oasis-open.org/committees/download.php/13525/sstc-saml-exec-overview-2.0-cd-01-2col.pdf, 2011.

[21] Shibboleth Information Sheet Overview. http://www.internet2.edu/pubs/shibboleth-infosheet.pdf, 2011.

[22] The Liberty Alliance. http://www.projectliberty.org/, 2011.

[23] Together. http://www.borland.com/us/products/together/index.aspx, 2011.

[24] yED. http://www.yworks.com/en/products_yed_about.html, 2011.

[25] Marwan Abi-Antoun and Nenad Medvidovic. Enabling the Refinement of a Software Architecture into a Design. In *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard*, UML'99, pages 17–31. Springer-Verlag, Berlin, Heidelberg, 1999. ISBN 3-540-66712-1.

[26] Hyder Ali Nizamani and Emilio Tuosto. Patterns of Federated Identity Management Systems as Architectural Reconfigurations. *ECEASST*, 31, 2010.

[27] Luciano Baresi, Reiko Heckel, Sebastian Thöne, and Dániel Varró. Style-based modeling and refinement of service-oriented architectures. *Software and System Modeling*, 5(2):187–207, 2006.

[28] Thais Batista, Ackbar Joolia, and Geoff Coulson. Managing Dynamic Reconfiguration in Component-Based Systems. In Ron Morrison and Flavio Oquendo, editors, *Software Architecture*, volume 3527 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin / Heidelberg, 2005.

[29] L. Boursas. Virtualization of the Circle of Trust amongst Identity Federations. In *1st International DMTF Academic Alliance Workshop on Systems and Virtualization Management: Standards and New Technologies*, 2007.

[30] Latifa Boursas and Vitalian A. Danciu. Dynamic Inter-organizational Cooperation Setup in Circle-of-Trust Environments. In *Proceedings of IEEE/IFIP Network Operations and Management Symposium: Pervasive Management for Ubioquitous Networks and Services,(NOMS)*, pages 113–120, 2008.

[31] Jeremy S. Bradbury. Organizing Definitions and Formalisms of Dynamic Software Architectures. Technical Report 2004-477, School of Computing, Queen‘s University, Kingston, Ontario, Canada, 2004.

[32] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. Marshall. GraphML Progress Report: Structural Layer Proposal. *Proceedings of the 9th International Symposium on Graph Drawing (GD '01)*, pages 501–512, 2002.

[33] Roberto Bruni, Matthias M. Hölzl, Nora Koch, Alberto Lluch-Lafuente, Philip Mayer, Ugo Montanari, Andreas Schroeder, and Martin Wirsing. A Service-Oriented UML Profile with Formal Support. In *Proceedings of 7th International Conference on Service Oriented Computing, (ICSOC)*, pages 455–469, 2009.

[34] Roberto Bruni and Alberto Lluch-Lafuente. Ten Virtues of Structured Graphs. *ECEASST*, 18, 2009.

[35] Roberto Bruni, Alberto Lluch-Lafuente, and Ugo Montanari. Hierarchical Design Rewriting with Maude. *Electronic Notes in Theoretical Computer Science*, 238 (3):45–62, 2009.

[36] Roberto Bruni, Alberto Lluch-Lafuente, Ugo Montanari, and Emilio Tuosto. Architectural Design Rewriting as an Architecture Description Language. In *R2D2 Microsoft Research Meeting*. 2008.

[37] Roberto Bruni, Alberto Lluch-Lafuente, Ugo Montanari, and Emilio Tuosto. Service Oriented Architectural Design. In Gilles Barthe and Cédric Fournet, editors, *Trustworthy Global Computing*, volume 4912 of *LNCS*, pages 186–203. Springer-Verlag, 2008. ISBN: 978-3-540-78662-7.

[38] Roberto Bruni, Alberto Lluch-Lafuente, Ugo Montanari, and Emilio Tuosto. Style-Based Architectural Reconfigurations. In Vladimiro Sassone, editor, *Euro-*

pean Association for Theoretical Computer Science, number 94, pages 161–180. 2008. ISSN:0252-9742.

[39] Antonio Bucchiarone. *Dynamic Software Architectures for Global Computing Systems*. PhD thesis, IMT Institute for Advanced Studies, Lucca, Italy, 2008.

[40] Jordi Cabot and Robert Clarisó. UML/OCL Verification In Practice. In *Proceedings of the 1st International Workshop on Challenges in Model-Driven Software Engineering (ChaMDE'08)*, 2008.

[41] Jordi Cabot, Robert Clarisó, and Daniel Riera. Verification of UML/OCL Class Diagrams using Constraint Programming. In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 73–80. IEEE Computer Society, Washington, DC, USA, 2008. ISBN 978-0-7695-3388-9.

[42] Jan Camenisch, Thomas Gross, and Dieter Sommer. Enhancing Privacy of Federated Identity Management Protocols:Anonymous Credentials in WS-Security. In *Proceedings of the 5th ACM workshop on Privacy in electronic society, (WPES '06)*. ACM, New York, NY, USA, 2006.

[43] Scott Cantor, Jeff Hodges, John Kemp, and Peter Thompson. Liberty ID-FF Architecture Overview (Version: 1.2-errata-v1.0). `http://www.projectliberty.org/liberty/content/download/318/2366/file/draft-liberty-idff-arch-overview-1.2-errata-v1.0.pdf`, 2011.

[44] David Chadwick. Federated Identity Management. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design V*, volume 5705 of *Lecture Notes in Computer Science*, pages 96–120. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-03828-0.

[45] David Chappell. Introducing Windows CardSpace. http://msdn.microsoft.com/en-us/library/aa480189.aspx, 2011.

[46] Dan Ioan Chiorean, Vladiela Petrascu, and Dragos Petrascu. How My Favourite Tool Supporting OCL Must Look Like. *ECEASST*, 15, 2008.

[47] Manuel Clavel and Marina Egea. ITP/OCL: A Rewriting-based Validation Tool for UML+OCL Static Class Diagrams. In *Proceedings of 11th International Conference on Algebraic Methodology and Software Technology (AMAST'06)*, pages 368–373. Springer-Verlag, 2006. ISBN 3-540-35633-9.

[48] Nelly Delessy, Eduardo B. Fernandez, and Maria M. Larrondo-Petrie. A Pattern Language for Identity Management. In *Proceedings of the 2nd International Multi-Conference on Computing in the Global Information Technology (ICCGI)*, page 31. IEEE Computer Society, USA, 2007. ISBN 0-7695-2798-1.

[49] Roland Erber, Christian Schlager, and Gunther Pernul. Patterns for Authentication and Authorisation Infrastructures. In *Proceedings of 18th International Workshop on Database and Expert Systems Applications (DEXA 2007)*, pages 755–759. IEEE Computer Society, USA, 2007. ISSN 1529-4188.

[50] Avetisyan et al. Open Cirrus: A Global Cloud Computing Testbed. *Computer*, 43:35–43, 2010. ISSN 0018-9162.

[51] Howard Foster, László Gönczy, Nora Koch, Philip Mayer, Carlo Montangero, and Dániel Varró. Uml extensions for service-oriented systems. In *Results of the SENSORIA Project*, pages 35–60. 2011.

[52] Martin Fowler and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 3rd edition, 2003. ISBN 0-321-19368-7.

[53] Pascal Fradet and Daniel Le Métayer. Shape Types. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 27–39. ACM, New York, USA, 1997. ISBN 0-89791-853-3.

[54] Emden R. Gansner and Stephen C. North. An Open Graph Visualization System and its Applications to Software Engineering. *Software: Practice and Experience*, 30:1203–1233, September 2000. ISSN 0038-0644.

[55] David Garlan. Style-Based Refinement for Software Architecture. In *Proceedings of the 2nd International Software Architecture Workshop (ISAW-2)*, pages 72–75. ACM, New York, USA, 1996.

[56] David Garlan, Robert Monroe, and David Wile. Acme: An Architecture Description Interchange Language. In *CASCON First Decade High Impact Papers*, CASCON '10, pages 159–173. ACM, New York, USA, 2010.

[57] Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and System Modeling*, 4(4):386–398, 2005.

[58] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69(1-3):27–34, 2007.

[59] Marc Goodner, Maryann Hondo, Anthony Nadalin, Michael McIntosh, and Don Schmidt. Understanding WS-Federation. Technical report, 2007.

[60] Manish Gupta and Raj Sharman. Dimensions of Identity Federation: A Case Study in Financial Services. *Journal of Information Assurance and Security*, 3: 244–256, 2008.

[61] Dan Hirsch, Paolo Inverardi, and Ugo Montanari. Graph Grammars and Constraint Solving for Software Architecture Styles. In *Proceedings of the third international workshop on Software architecture (ISAW '98)*, pages 69–72. ACM, New York, USA, 1998. ISBN 1-58113-081-3.

[62] R. Housley, W. Polk, W. Ford, and D. Solo. *Internet X.509 Public Key Infrastructure - Certificate and Certificate Revocation List (CRL) Profile. RFC 3280.* 2002.

[63] James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl, Jaime Rodrigo Oviedo Silva. Documenting Component and Connector Views with UML 2.0. Technical Report CMU/SEI-2004-TR-008, ESC-TR-2004-008, Software Engineering Institute, CMU, 2004.

[64] Mohamed Hadj Kacem, Ahmed Hadj Kacem, Mohamed Jmaiel, and Khalil Drira. Describing Dynamic Software Architectures Using an Extended UML Model. In *Proceedings of the 21st Annual ACM Symposium on Applied Computing*, pages 1245–1249. ACM, 2006.

[65] Jung Soo Kim and David Garlan. Analyzing Architectural Styles. *Journal of Systems and Software*, 83(7):1216 – 1235, 2010. ISSN 0164-1212. SPLC 2008.

[66] Uwe Kylau, Ivonne Thomas, Michael Menzel, and Christoph Meinel. Trust Requirements in Identity Federation Topologies. *AINA*, pages 137–145, 2009.

[67] Ivan Lanese and Emilio Tuosto. Synchronized Hyperedge Replacement for Heterogeneous Systems. In *Proceedings of 7th International Conference on Coordination Models and Languages (COORDINATION)*, pages 220–235. Springer, 2005. ISBN 3-540-25630-X.

[68] Manuel Clavel, Marina Egea and Viviane Torres Da Silva. Mova: A Tool for Modeling, Measuring and Validating UML Class Diagrams. *Academic Posters and Demonstrations Session of MODELS 2007*, 2007.

[69] N. Medvidovic, D.S. Rosenblum, and R.N. Taylor. A Language and Environment for Architecture-based Software Development and Evolution. In *Proceedings of the 21st International Conference on Software Engineering*, pages 44–53. ACM, New York,USA, 1999. ISSN 0270-5257.

[70] Nenad Medvidovic. *Architecture-based Specification-time Software Evolution*. PhD thesis, 1999. AAI9912025.

[71] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology*, 11(1), 2002.

[72] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *ACM Transactions on Software Engineering and Methodology*, 26(1):70–93, 2000. ISSN 0098-5589.

[73] Daniel Le Métayer. Software Architecture Styles as Graph Grammars. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 15–23. ACM press, 1996.

[74] Daniel Le Métayer. Describing Software Architecture Styles Using Graph Grammars. *IEEE Transactions on Software Engineering*, 24(7):521–533, 1998.

[75] Russ Miles and Kim Hamilton. *Learning UML 2.0*. O'Reilly Media, Inc., 2006. ISBN 0596009828.

[76] R. L. Morgan, Scott Cantor, Steven Carmody, Walter Hoehn, and Ken Klingenstein. Federated Security: The Shibboleth Approach. *EDUCAUSE Quarterly*, 27 (4):12–17, 2004.

[77] Mark Moriconi, Xiaolei Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21:356–372, 1995.

[78] Patrick Morrison and Eduardo B. Fernandez. The Credentials Pattern. In *Proceedings of the 2006 conference on Pattern languages of programs (PLoP '06)*, pages 1–4. ACM, New York, NY, USA, 2006. ISBN 978-1-60558-372-3.

[79] Jon Oltsik. Services-Oriented Architecture (SOA) and Federated Identity Management (FIM). White paper, ESG, 2006.

[80] OMG. Unified Modeling Language (Version 2.2). `http://www.omg.org/spec/UML/2.2/Superstructure/PDF/`, 2009.

[81] Jorge Enrique Pérez-Martínez. Heavyweight Extensions to the UML Metamodel to Describe the C3 Architectural Style. *SIGSOFT Software Engineering Notes*, 28(3):5, 2003. ISSN 0163-5948.

[82] Birgit Pfitzmann and Michael Waidner. Federated Identity-Management Protocols. In Bruce Christianson, Bruno Crispo, James Malcolm, and Michael Roe, editors, *Security Protocols*, volume 3364 of *Lecture Notes in Computer Science*, pages 153–174. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-28389-8.

[83] Sunghwan Roh, Kyungrae Kim, and Taewoong Jeon. Architecture Modeling Language Based on UML 2.0. In *Proceedings of the 11th Asia-Pacific Conference on Software Engineering*, pages 663–669, 2004. ISSN 1530-1362.

[84] B. Schmerl and D. Garlan. AcmeStudio: Supporting Style-centered Architecture Development. In *Proceedings of 26th International Conference on Software En-*

gineering, (ICSE 2004), pages 704 – 705. IEEE Computer Society, may 2004. ISSN 0270-5257.

[85] Bradley Schmerl and David Garlan. Exploiting Architectural Design Knowledge to Support Self-Repairing systems. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, SEKE '02, pages 241–248. ACM, New York, NY, USA, 2002. ISBN 1-58113-556-4.

[86] Mary Shaw and David Garlan. *Software Architectures: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[87] S.S.Y. Shim, Geetanjali Bhalla, and Vishnu Pendyala. Federated Identity Management. *Computer*, 38(12):120 – 122, 2005. ISSN 0018-9162.

[88] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., and Jason E. Robbins. A Component-and Message-based Architectural Style for GUI Software. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 295–304. ACM, New York, USA, 1995. ISBN 0-89791-708-1.

[89] Qurat ul Ain Nizamani and Hyder A. Nizamani. Analysis of a Federated Identity Management Protocol in SOC. In *Proceedings of the 3rd Young Researchers Workshop on Service Oriented Computing (YRSOC'08)*, 2008.

[90] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003. ISBN 0321179366.

[91] Hironori Washizaki, Eduardo B. Fernández, Katsuhisa Maruyama, Atsuto Kubo, and Nobukazu Yoshioka. Improving the Classification of Security Patterns. In *DEXA Workshops*, pages 165–170, 2009.

[92] Martin Wirsing and Matthias M. Hölzl, editors. *Rigorous Software Engineering for Service-Oriented Systems - Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing*, volume 6582 of *Lecture Notes in Computer Science*. Springer, 2011. ISBN 978-3-642-20400-5.