

A Model-based Framework for System Configuration Management

Azadeh Jahanbanifar

A Thesis

in

The Department of

Computer Science and Software Engineering

Presented in Partial Fulfilment of the Requirements

for the Degree of Doctor of Philosophy at

Concordia University

Montreal, Quebec, Canada

April 2016

© Azadeh Jahanbanifar, 2016

**CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES**

This is to certify that the thesis prepared

By: Azadeh Jahanbanifar

Entitled: A Model-based Framework for System Configuration Management

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____	Chair
Dr. R. Dssouli	
_____	External Examiner
Dr. D. Petriu	
_____	External to Program
Dr. J. Bentahar	
_____	Examiner
Dr. J. Rilling	
_____	Examiner
Dr. D. Goswami	
_____	Thesis Co-Supervisor
Dr. F. Khendek	
_____	Thesis Co-Supervisor
Dr. M. Toeroe	

Approved by: _____
Dr. V. Haarslev, Graduate Program Director

March 31, 2016 _____
Dr. A. Asif, Dean
Faculty of Engineering and Computer Science

ABSTRACT

A Model-based Framework for System Configuration Management

Azadeh Jahanbanifar, Ph.D.

Concordia University, 2016

A system can be viewed from different perspectives, each focusing on a specific aspect such as availability, performance, security. Configurations reflect the manageable resources of the system, their attributes and organization which are necessary for the management of the system for each aspect. Thus, for management purposes a system is generally described through various partial configurations (also known as configuration fragments). To form a consistent system configuration, these independently developed configuration fragments need to be integrated together. The integration of configuration fragments is a challenging task. This is mainly due to overlapping entities (different logical representations of the same system resource) in the configuration fragments and/or complex relationships among the entities of the different configuration fragments. At runtime the system may be reconfigured to meet certain/new requirements or in response to performance degradations. These changes may lead to inconsistency as some changes may violate the constraints between entities. Maintaining the consistency and adjusting the system configuration at runtime is another challenging task. In our research, we propose to handle these two important issues in an integrated manner. We define a model-based framework for configuration management. We use the Unified Modeling Language (UML) and its profiling mechanism for representing the configuration fragments. Using model weaving and model transformation techniques, we propose a solution for the integration of configuration fragments tar-

getting specific system properties. To handle runtime changes, we propose a configuration validation and adjustment solution to check and preserve the consistency of the system configuration. We introduce a partial validation technique in which the runtime reconfigurations are checked against a reduced set of consistency rules instead of the complete set of rules and the reconfigurations are applied only if they are safe, i.e. they preserve the configuration consistency. For handling the changes that violate the consistency rules, we propose an adjustment technique to automatically resolve (if possible) the inconsistencies. This is achieved by propagating the changes in the configuration according to the system constraints following the possible impacts of the configuration entities on each other. Some heuristics are used to reduce the complementary changes and to limit the propagation. We evaluate the complexity of our adjustment technique and conduct experiments to evaluate its efficiency.

The Service Availability Forum middleware is used as an application domain in the examples throughout this thesis; however the proposed solutions are applicable in more general settings. We present proofs of concepts using different technologies. We use the Eclipse Modeling Framework (EMF) and Papyrus for implementing the UML profiles. The Atlas Model Weaver (AMW) and Atlas Transformation Language (ATL) are used to integrate the configuration fragments, and we also use the APIs of the Object Constraint Language (OCL) in the Eclipse environment and the Microsoft Z3 constraint solver to develop a prototype tool of our partial validation and adjustment agents.

Acknowledgments

I would like to express my sincere thanks and appreciation to those who patiently helped me accomplish this study: My supervisors Dr. Ferhat Khendek and Dr. Maria Toeroe for their tireless guidance, inspiration, encouragement and sharing their knowledge with me.

I would like to extend my thanks to the examining committee for their advice and support during the various stages of my PhD.

I express my deepest gratitude to my beloved family for their endless love and support.

I would like to thank my friends who always believed in me and supported me in these years.

This work has been partially supported by Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson and Concordia University. My gratitude is extended to Concordia University and Ericsson for their financial support and providing me the opportunity to learn and grow.

Table of Contents

List of Figures	ix
List of Tables	xi
List of Acronyms	xii
1 Introduction.....	1
1.1 Thesis Motivation	1
1.2 Contributions of this Thesis	3
1.3 Thesis Organization	7
2 Background.....	9
2.1 Model Driven Development.....	9
2.1.1 Domain Specific Modeling Languages.....	10
2.1.2 The Model Weaving Technique.....	11
2.2 The Service Availability Forum Middleware	14
2.2.1 The Availability Management Framework (AMF).....	14
2.2.2 Entity Types File (ETF).....	16
2.2.3 Platform Management (PLM).....	17
2.2.4 The Relation between the AMF and PLM Configurations	18
2.3 Open Virtualization Format (OVF).....	20
3 Related Work	23
3.1. Configuration Generation	23
3.2. Configuration Validation	25
3.3. Configuration Adjustment.....	28
3.4. Summary	30
4 Configuration Management Framework.....	32
4.1 Introduction.....	32
4.2 Extending OCL	34
4.3 System Configuration Design: Integration of Configuration Fragments	38
4.4 System Runtime: Consistency Preservation with Change Management	39
4.4.1 Runtime Configuration Validation.....	40
4.4.2 Configuration Adjustment.....	41

4.5	Summary	42
5	The Integration of Configuration Fragments	44
5.1	Introduction	44
5.2	The Challenges	46
5.2.1	Overlapping Entities	46
5.2.2	Integration Relations between Configuration Fragments	47
5.3	The Overall Approach	50
5.3.1	Extending the Generic Weaving Metamodel	52
5.3.2	Creating the Links in the Weaving Model	57
5.3.3	Generating the System Configuration from the Weaving Model	59
5.4	Constraint Generation from the Integration	64
5.4.1	Entity Derivation Tree	65
5.4.2	Translation of the ATL Operations to the OCL Expressions	69
5.4.3	Role Definition for the Constrained Entities	71
5.5	Implementation and Discussion	72
5.6	Summary	74
6	Partial Validation of Configurations at Runtime	76
6.1	Introduction	76
6.2	Partial Validation Technique	77
6.2.1	Filtering the Constraints	79
6.2.2	Categorizing the Constraints	81
6.2.3	Validation of the Constraints	83
6.3	A Semi-formal Proof for the Partial Validation Technique	83
6.3.1	Definitions	84
6.3.2	Modifying the Model	86
6.3.3	The Proof of Partial Validation	87
6.4	Prototype Implementation and Evaluation	91
6.4.1	Implementation Setup	91
6.4.2	Evaluation Scenarios	92
6.4.3	Results and Discussions	92
6.5	Summary	94

7	Runtime Adjustment of System Configuration.....	96
7.1	Introduction.....	96
7.2	Consistency Preservation through Adjustment.....	97
7.2.1	Adjustment Challenges.....	98
7.2.2	Preliminary Definitions.....	99
7.3	The Adjustment Process.....	105
7.3.1	Grouping the Overlapping Scopes.....	106
7.3.2	Depth-first Incremental Change Propagation for Solving Groups with a Single Scope.....	109
7.3.3	Path Bonding for Solving Groups with Multiple Scopes.....	114
7.3.4	The Overall Adjustment.....	117
7.3.5	Complexity Analysis for the Overall Adjustment Process.....	118
7.4	Prototype Implementation and Experimental Evaluation.....	121
7.4.1	Evaluation Scenarios.....	122
7.4.2	Solving a Group with a Single Scope.....	123
7.4.3	Solving Multiple Groups.....	125
7.4.4	Detection of the Not-adjustable Changes for Multiple Groups.....	127
7.5	Summary.....	128
8	Conclusion and Future Work.....	131
8.1	Conclusion.....	131
8.2	Future Research.....	133
8.2.1	The Integration of Configuration Fragments.....	133
8.2.2	Partial Validation of the Configuration.....	134
8.2.3	Runtime Adjustment of the System Configuration.....	135
	References.....	136
	Appendix A: An Excerpt of the Higher Order Transformation (HOT) for the Generation of System Configuration.....	142

List of Figures

Figure 1.1. Various configuration fragments of a system with different logical representations of the same physical entity	2
Figure 2.1. The weaving model	12
Figure 2.2. The core weaving metamodel.....	13
Figure 2.3. A simplified AMF configuration and a portion of the AMF configuration metamodel	15
Figure 2.4. A simplified PLM configuration and a portion of the PLM configuration metamodel	17
Figure 2.5. The relation between the AMF and PLM configurations	19
Figure 2.6. The structure of the Petstore OVF package	21
Figure 4.1. Overview of configuration management framework.....	33
Figure 4.2. Partial model of the Virtual Systems, their collection and placement policy in an OVF package	35
Figure 4.3. OCL constraint enriched by the leadershipInfo.....	36
Figure 4.4. Representation of entity roles in constraints.....	37
Figure 4.5. Overall view of the configuration validation and adjustment process.....	40
Figure 5.1. Different representations of a Virtual Machine in different configuration fragments.....	46
Figure 5.2. Host failure problem because of the relation between the AMF and PLM configuration fragments	48
Figure 5.3. Latency problem because of the relation between the AMF and PLM configuration fragments	49
Figure 5.4. The system configuration generation through model weaving.....	51
Figure 5.5. The generic weaving metamodel extended with new LinkTypes and LinkEnds	53
Figure 5.6. A partial PLM configuration and corresponding hardware dependency table	60
Figure 5.7. Generation of OCL constraints from the ATL transformation.....	65
Figure 5.8. An example of derivation tree for SystemEEVM.....	67
Figure 5.9. The derivation trees for the AmfNode, AmfNG, and AmfSU.....	68

Figure 6.1. Model changes and affected constraints in the profile	78
Figure 6.2. Change profile and a simple change model.....	80
Figure 6.3. Filtering and categorizing the constraints based on the change model.....	82
Figure 7.1. Representation of OVF model with constraints and the role of entities in constraints....	100
Figure 7.2. An example of the propagation scope for an infringing entity	103
Figure 7.3. A PathCollection with multiple paths for an infringing entity	105
Figure 7.4. Multiple constraint violations with overlapping propagation scopes	106
Figure 7.5. The formation of a group and its Intersect from overlapping scopes	107
Figure 7.6. Depth-first incremental change propagation	111
Figure 7.7. Selecting the paths of the Group for the bonding	114
Figure 7.8. Comparison of the number of complementary changes using the overall adjustment resolution versus the group-based and total change resolutions	123
Figure 7.9. Comparison of the execution time using the overall adjustment resolution versus the group-based and total change resolutions	124
Figure 7.10. Comparison of the number of complementary changes using the overall adjustment resolution versus, the group-based and the total change resolutions	125
Figure 7.11. Comparison of the execution times using the overall adjustment resolution versus, the group-based and the total change resolutions	126
Figure 7.12. Comparison of the execution times of the overall adjustment resolution versus the total change resolution	128

List of Tables

Table 5.1. The mapping of ATL operations to OCL expressions	70
Table 5.2. The OCL expressions resulting from the derivation trees of Fig. 5.9	70
Table 6.1. Partial validation performance results.....	92

List of Acronyms

AMF	Availability Management Framework
AMW	Atlas Model Weaver
ATL	Atlas Transformation Language
COTS	Commercial Off-The-Shelf
CSP	Constraint Satisfaction Problem
DMTF	Distributed Management Task Force
DSML	Domain Specific Modeling Language
EE	Execution Environment
EMF	Eclipse Modeling Framework
ETF	Entity Types File
HA	High Availability
HE	Hardware Element
HOT	Higher Order Transformation
MDD	Model Driven Development
NG	Node Group
OCL	Object Constraint Language
OS	Operating System
OVF	Open Virtualization Format
PLM	Platform Management
SA Forum	Service Availability Forum
SG	Service Group
SI	Service Instance

SPL	Software Product Line
SU	Service Unit
UML	Unified Modeling Language
VM	Virtual Machine
VMG	Virtual Machine Group
VMM	Virtual Machine Monitor
WMM	Weaving MetaModel
WModel	Weaving Model

Chapter 1

Introduction

1.1 Thesis Motivation

The utilization of reusable commercial off-the-shelf (COTS) components promises a reduction in time and cost of software development as well as higher quality, more reliable and maintainable software. A system (e.g. new composite applications or a system of systems such as in the cloud architectures) is built by putting together such independently developed COTS components. Each of these components/sub-systems may have its own perspective of the system described as a configuration. This configuration specifies the organization and the characteristics of the resources the component/sub-system is aware of and potentially manages. A system can also be viewed from different perspectives or aspects (such as performance, security, availability) and thus have multiple configurations. Thus, a composite system is described through various independently developed configuration fragments. Fig. 1.1 shows an example of a system having multiple configurations, each reflecting a logical view of some physical resources of the system. One of the main challenges of such systems is the integration of these configuration fragments in a consistent manner that reflects the relations and constraints between the entities of the different

fragments and ensures that the resulting system meets the required properties such as availability, performance and security. The complexity of the integration task stems from the overlapping entities of the different configuration fragments (i.e. different logical representations of the same physical entity) and from the complex relationships among the entities of the different configuration fragments. Manual and ad-hoc integration of the fragments is difficult and error prone. The integration process needs to be repeated when the fragments change or a new fragment is added. Moreover when multiple systems with different configurations are needed to be built from the same set of components for multiple customers, the integration process is repeated for each system. Thus, the first challenge we tackle in this thesis is to define an automated approach for the integration of configuration fragments while taking into account special aspects or properties of the system.

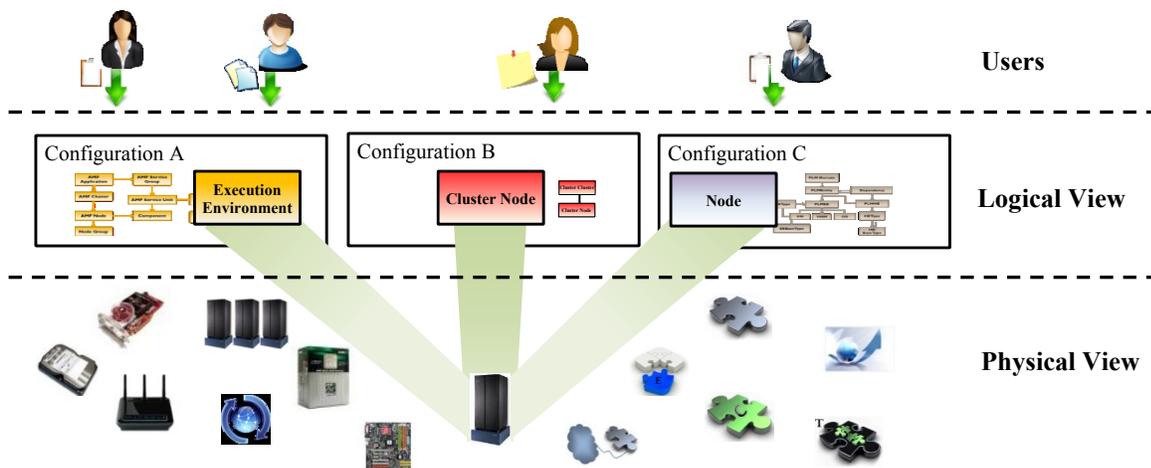


Figure 1.1. Various configuration fragments of a system with different logical representations of the same physical entity

At runtime a system actor (i.e. the administrator or a management application) may need to modify the configuration to control the system resources. These reconfigurations are needed in order to meet certain/new requirements, respond to performance degradations, for elasticity or upgrade

purposes. Changes made to a configuration entity may have an impact on other entities of the configuration because of the relations and dependencies between the entities. The changes should be conducted in a safe way not to endanger the consistency of the system configuration and therefore jeopardize the system operations. Thus, the proposed changes should be checked and the modified configuration needs to be validated to guarantee its consistency and therefore to protect the system from malfunctioning and from service outage. Following Moazzami-Goudarzi [1], the consistency of a configuration is defined as the correctness of the data which requires the satisfaction of the structural integrity requirements and the application/domain constraints. The system configuration, especially for large systems, or composite systems, may consist of thousands of entities each with several attributes and complex relations between the entities. In such systems, the management and control of the reconfiguration side-effects with an ad hoc or manual approach is a difficult and error-prone task as the actor must know and take care of all the relations and constraints. This problem is even worse for real-time and highly available systems as the validation and reconfiguration time should be minimal. Moreover, they should not be shut down or restarted for the reconfiguration. Therefore, an automated and efficient approach is required to manage the reconfiguration and protect the system consistency from invalid modifications.

1.2 Contributions of this Thesis

To address the aforementioned issues we define a model-based configuration management framework. The Model Driven Development (MDD) paradigm shifts the focus of software development from programming to modeling, thus models are the primary artifacts in the development process. This allows the developers to separate the application logic from the platform

technology and create/manipulate platform-independent models [2]. This paradigm is appropriate for our purpose as it allows the concepts and methods of a domain or application to be defined at a higher level of abstraction. It enables automation, which consequently increases the efficiency, portability, and reusability while reducing the time and cost. We propose a model-based framework that facilitates the integration, validation and adjustment of system configurations. In this framework Domain Specific Modeling Languages (DSML) capture the concepts, their relations and consistency rules (constraints) of configuration fragments. Unified Modeling Language (UML) and its profiling mechanism [3, 4] is our choice for defining the DSMLs. The constraints coming from the application domain restrict the configuration entities and their relations by governing both their structure and behavior. The constraints are expressed using the Object Constraint Language (OCL) [5].

Our configuration management framework consists of two parts: The integration of configuration fragments at design time and the consistency preservation of configurations after modifications at runtime. We use model weaving [6] and model transformation techniques to integrate the fragments into a system configuration in a consistent manner. We also consider certain requirements or aspects of the domain (i.e. availability in our work) during the integration. Model weaving allows us to define different types of links (link types) to capture the relation between the configuration profiles and use the links to hook together configuration fragments. This allows for the development of more abstract mappings and increases the reusability of link types since they can be used in future mappings when other configuration fragments need to be added. It also increases the portability as the developer can select the desirable interpretation and implementation

for the mapping because the declarative definition of the link types can be translated according to the features of the system.

To preserve the consistency of the configuration at runtime we propose configuration validation and adjustment techniques. Runtime validation is a prerequisite for systems with dynamic reconfiguration capabilities as it detects the potential inconsistencies that can be caused by the reconfigurations (changes to a single system entity or a bundle of changes to a number of entities). Runtime reconfigurations often target only parts of the configuration and an exhaustive validation can be time and resource consuming. We therefore define a model-based approach for partial validation of the configurations at runtime to reduce the validation time and overhead. A configuration model is validated against the configuration profile, including the OCL constraints. In our proposed partial validation only the constraints that are affected and need to be checked are selected as the other ones remain valid. We have extended OCL by defining roles for the entities participating in a constraint. This extension allows us to categorize the constraints of the configuration profile. Moreover, the output of the partial validation will serve as input for an adjustment technique which whenever needed and when possible performs corrective actions that mitigate potential inconsistencies.

The potential inconsistencies detected by the runtime validation technique can be due to the incompleteness of the set of changes as performed by the administrator who is not aware of all the relations between all involved entities/attributes. In order to resolve such inconsistencies we devise a technique for complementing an incomplete set of changes and therefore adjust automatically the configuration at runtime. The adjustment consists of modifications of other entities/attributes that re-establish the configuration consistency. We achieve this by propagating the

changes in the configuration according to the system constraints following the possible impacts of the configuration entities on each other. We aim at minimizing the complementary modifications to control the side-effects of the change. The problem is formulated as a Constraint Satisfaction Problem (CSP) [67, 68] which we solve using a constraint solver.

The main contributions of this thesis are summarized as follows:

- To integrate the configuration fragments we propose a model-based approach using model weaving while enriching the weaving technique with more semantics. This semantics allows for considering the special properties of the system (such as availability, security) in the weaving. We capture the relation between the entities of the configuration profiles as link types. Modeling the links in a higher level of abstraction has several advantages such as reusability of the links, easier extensibility (for adding other configuration models) and automation of the integration process.
- To maintain the consistency of the configuration at runtime we propose a model-based configuration validation to verify the requested modifications before applying them to the configuration. The requested modifications are checked against the system constraints. To reduce the validation time and overhead, instead of checking again all the constraints our partial validation technique determines the set of constraints that need to be checked again for validity. The detected inconsistencies, i.e. the output of the partial validation serve as input for the adjustment process.
- We propose an adjustment technique to maintain the system consistent by resolving the potential inconsistencies that may occur because of the reconfigurations. The purpose is to automatically neutralize the potential inconsistencies, which are detected by the partial validation process. The adjustment is done by manipulating relevant parts of the configuration in order to keep the system constraints satisfied. We evaluate the complexity of the technique and also conduct some experiments whose results indicate the efficiency of the technique.

To illustrate our work we use the Service Availability Forum (SA Forum) [7] middleware as an application domain throughout this thesis. However, our work is applicable in more general settings where partial models need to be integrated and consistency of the integrated models has to be maintained at runtime.

As a proof of concept we implemented our configuration integration approach for weaving two configuration profiles of the SA Forum middleware using the Atlas Model Weaver (AMW) [8] and the Atlas Transformation Language (ATL) [9]. A prototype of the partial validation technique has been implemented in the Eclipse Modeling Framework (EMF) [10] using OCL APIs. We also implemented a prototype of our adjustment mechanism with the help of Microsoft Z3 constraint solver [11] and we evaluate the performance of the approach with empirical experiments.

The contributions reported in this thesis have been published in papers [18, 22, 27, 28, 71, 72] and also filed patents [73, 74, 75, 76, 77].

1.3 Thesis Organization

The rest of this thesis is organized as follows: In Chapter 2, we introduce the background knowledge including the SA Forum which is used as an application domain throughout this thesis. We also introduce briefly the model-driven paradigm and model weaving. Chapter 3 reviews the related work in configuration integration, validation and adjustment. In Chapter 4 we introduce our model-based framework for configuration management. Chapter 5 describes our model-based solution for the integration of configuration fragments. In Chapter 6 we explain our partial validation method for the configuration at runtime. Chapter 7 presents our solution to automate

the adjustment of the configuration at runtime. In Chapter 8 we conclude the thesis by reviewing its main contributions and potential future work.

Chapter 2

Background

In this chapter, a brief overview of the model driven paradigm and the SA Forum middleware is provided. In the first part of this chapter we introduce the main concepts of model driven development such as Domain Specific Modeling languages (DSML), UML profiles and model weaving technique. The SA Forum middleware [7] is used as an application domain for the illustration of our proposed techniques. Therefore, we introduce the SA Forum middleware in the second part of this chapter. More specifically we focus on Availability Management Framework (AMF) [11], and Platform Management (PLM) [12] service of the middleware, their configurations and the relations between them. We also introduce Entity Types File (ETF) [21] and Open Virtualization Format (OVF) specification [19] and use them to describe our work.

2.1 Model Driven Development

The model driven is a promising engineering paradigm for software development and management which emphasizes abstracting the concepts of the domain by creating and analyzing models. The models are the primary artifacts which replace the codes in the software development.

Models provide an abstract representation of the real world concepts and hide the unnecessary details. This higher level of abstraction improves productivity of the software development processes as it allows the designers to focus on relevant aspects of the system and ignore the extraneous details. Moreover, platform independent models capture the information about the system and its behavior rather than the specific implementation and platform details which increases the portability and interoperability and makes it easy to migrate to other technologies.

Various modeling languages can be used. They can be generic such as Unified Modeling Language (UML) [3] or they can be custom-made to capture specific concepts and properties of an application domain i.e. Domain Specific Modeling Languages (DSMLs) [29].

Various operations are defined to produce other artifacts (such as source codes, configurations, and inputs for analysis tools) from models. Model-to-model and model-to-code transformations are the most common operations used to manipulate the models. The transformations are mapping functions from the source model(s) to the target model(s)/codes which facilitate the automation of the development process.

2.1.1 Domain Specific Modeling Languages

To describe models a modeling language is required. General-purpose modeling languages such as UML can be used as metamodels to define models. A metamodel defines the entities, their structure and semantics that can be specified in the models (i.e. instances of the metamodel). A model which is built according to the syntax and semantics of a metamodel is said to *conform* to the metamodel. A DSML is a specialized language for defining the models of a specific domain. DSMLs allow developers to express their application models with specific concepts of the appli-

cation domain. They increase the expressiveness and usability of the models and make the communication between users easier.

To benefit the UML as a standardized and popular modeling language and also take advantages of DSMLs, UML introduces extension mechanisms via the definition of profiles [3, 4]. The UML profiling mechanisms allows us to constrain and customize the UML for creating DSMLs for specific domains and platforms. A UML profile consists of stereotypes, tagged values (i.e. the attributes of the stereotypes), and constraints to restrict and customize the UML. OCL [5] is a formal language that can be used to specify application specific constraints in UML profiles and models. Although the constraints can express different semantics in the models, they are side-effect free, i.e. their evaluation cannot alter the state of the executing system [5]. In our work we use UML profiles to formally define the concepts, relations and constraints of the configuration domains.

2.1.2 The Model Weaving Technique

Model weaving is an MDD technique which aims at solving the problem of mapping between heterogeneous data. Model weaving is a model management operation that establishes fine-grained correspondences between model/metamodel entities [13,14]. The correspondences (mapping) between the entities are captured in a *Weaving Model*. The weaving models can be translated to desirable model transformation formats to become executable or used as an input for other model management operations [13].

In a weaving model, the relations between the elements are defined with so-called “*links*”. The weaving model should conform to a weaving metamodel (WMM) which describes the types of

the mappings (link types) that are valid in the weaving model. Fig. 2.1 shows how the weaving model relates two other models while each model conforms to its metamodel. A weaving model is useful in many application scenarios such as model merging, transformation generation, linking entities across models and traceability.

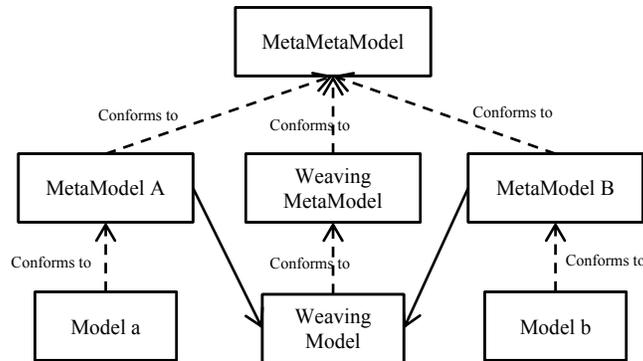


Figure 2.1. The weaving model

The core weaving metamodel has been developed by the AtlanMod group [15] at INRIA [16] and supports the creation of different kinds of links between the model/metamodel elements. In the core weaving metamodel which is shown in Fig. 2.2, *WElement* is the base element from which all other elements inherit. It has a name and a description. *WModel* represents the root element for all the model elements. *WLink* denotes the link type and has a reference i.e. *end* to associate it with a set of endpoints, *WLinkEnd*, for the link type. Each *WLinkEnd* references one *WElementRef*. The attribute *ref* contains the identifier of the linked element. The *WElementRef* is not associated directly to the *WLink* because it is possible to reference the same model element by different link endpoints, for example one model element may participate in more than one mapping expression. *WModelRef* is similar to *WElementRef*, but it contains references to the

models [8, 14]. The core weaving metamodel is extensible and the WLink and WLinkEnd can be specialized to represent special relations and concepts in different application domains.

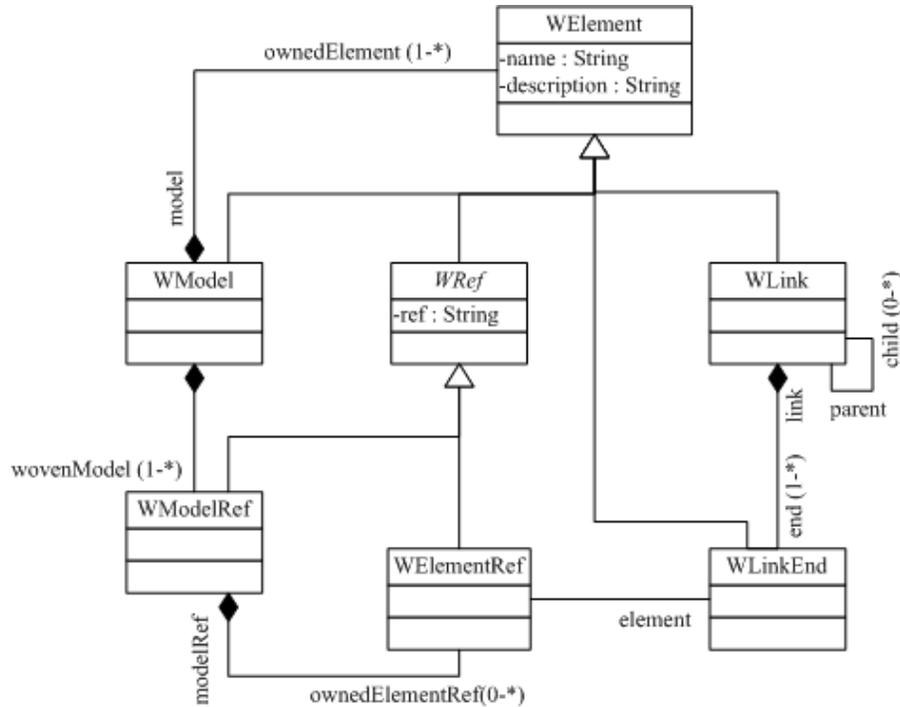


Figure 2.2. The core weaving metamodel

The model weaving offers several advantages: Because the information and correspondences between the models can be described by the weaving models, there is no need to capture all aspects of a system in a large metamodel. Several metamodels can be defined for a system, each focusing on a specific aspect or sub-domain of the system. While it is easier to define and maintain the individually defined metamodels (and their respective models), they are interconnected through the weaving links. Moreover defining abstract mappings as special link types in the weaving metamodel increases the reusability since they can be used in future mappings when other models need to be added with the same relations. It also allows for the selection of the de-

sirable interpretation and implementation for the mapping. This means that the declarative definition of the link types can be translated and implemented according to the requirements of each system.

We use model weaving as the key operation for the configuration integration in our configuration management framework where a weaving model captures the special relations between the configuration fragments.

2.2 The Service Availability Forum Middleware

The SA Forum [7], a consortium of telecommunication and computing companies has defined standard specifications to support the development of highly available systems. The SA Forum defines a platform independent middleware for managing the high availability for the applications under its control. The middleware consists of several services and frameworks, which represent and control specific aspects of the system and collaborate with each other. In this section we briefly introduce the AMF [11] and PLM [12] services of the middleware, their configurations and the relation between the configurations.

2.2.1 The Availability Management Framework (AMF)

AMF is responsible for maintaining the availability of application services by managing and coordinating the redundant software entities that compose the application under its control [11]. This management is based on the AMF configuration of the application. A simplified example of an AMF configuration of an application is shown in the left side of Fig. 2.3. This configuration includes several logical entities i.e. an abstract description of an application components and services. In an AMF configuration a component is the smallest service provider entity. A combina-

tion of collaborating components forms a Service Unit (SU) and the workload provisioned by an SU is represented as a Service Instance (SI). A group of redundant SUs capable of providing the same SIs forms a Service Group (SG). An application may consist of a number of SGs.

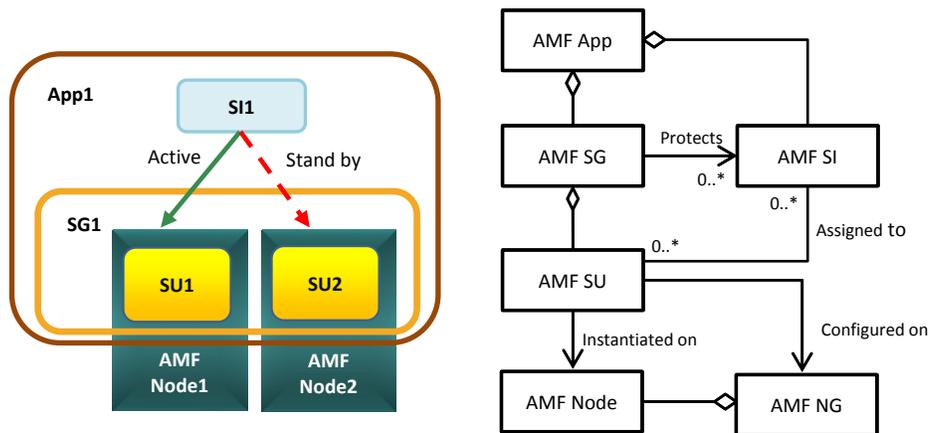


Figure 2.3. A simplified AMF configuration and a portion of the AMF configuration metamodel

At runtime to protect each SI, AMF assigns it in the active and standby roles to the SUs of the SG. In case of the failure of the SU with the active assignment AMF dynamically moves the active assignment from the faulty SU to the standby. Each SU is instantiated on an AMF Node, which is a logical container of the AMF components and SUs [11]. AMF Nodes can also be grouped into subsets of the cluster called Node Groups (NGs), which can be configured to host SUs and SGs. The NG configured for an SG determines the set of nodes its SUs can be instantiated on. Similarly, the NG configured for an SU refers to the set of nodes from which AMF selects one at runtime, which will host the SU. In the example AMF configuration on the left hand side of Fig. 2.3 App1 consists of one SG (SG1). This SG has two SUs (SU1, SU2) and protects the service represented by SI1. Each of the SU1 and SU2 is hosted on a separate node: Node1 and Node2. An AMF configuration consists of all these entities, their types and their attributes. Please note that in

this section we only introduced the entities required for the rest of this dissertation. More information on the AMF configuration can be found in [11].

The concepts in an AMF configuration, their relationships, and the related constraints have been captured in an AMF configuration metamodel. A portion of this metamodel is shown on the right hand side of Fig. 2.3. Subsequently, an AMF UML profile has been defined by mapping the AMF configuration metamodel to the UML metamodel. The complete definitions of the AMF configuration metamodel and the respective AMF UML profile are discussed in [65]. We use this AMF UML profile as an example for explaining our configuration integration approach.

2.2.2 Entity Types File (ETF)

The Entity Types File (ETF) [21] is a standardized software description file in the SA Forum context. More specifically ETF is a component catalog in the availability domain which describes the software components, their capabilities, dependencies, limitations and also the software deployment options. The ETF is provided by the software vendor and describes the characteristics and constraints of the component types included in the software. Hence ETF can be seen as a configuration prototype for the AMF.

ETF is used as one of the inputs for design and generation of the AMF configuration for a given software. It provides information about the capabilities of the components, their dependencies and how they can be combined together.

2.2.3 Platform Management (PLM)

The PLM service is responsible for providing a logical view of the platform entities of the system and managing them. The platform entities are the Hardware Elements (HEs) and the low level software entities also known as the Execution Environments (EEs) [12]. A simple example of a PLM configuration is shown on the left hand side of Fig. 2.4.

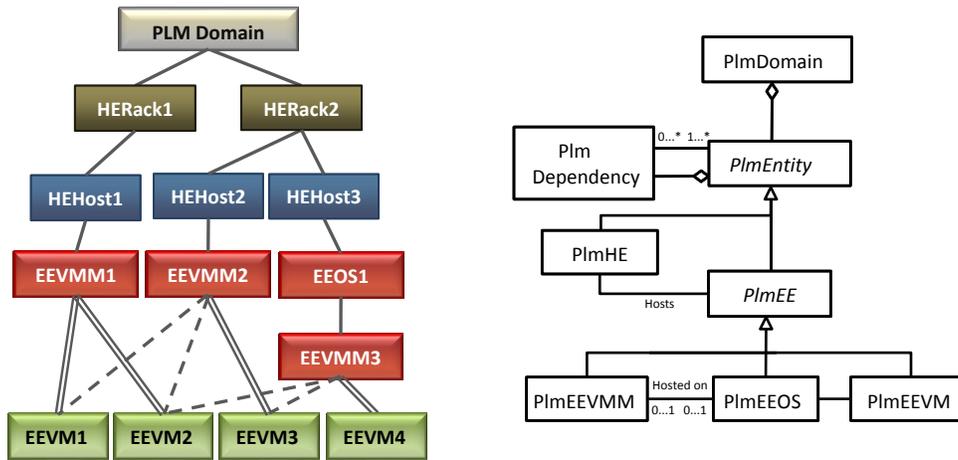


Figure 2.4. A simplified PLM configuration and a portion of the PLM configuration metamodel

In a PLM configuration PlmEEs represent software environments that can execute other software. A PlmEE can be an Operating System (OS), a Virtual Machine Monitor (VMM) or a Virtual Machine (VM) [12]. A PLM HE with computational capabilities can host a VMM or an OS. An OS can be the parent of other PLM EEs, i.e. VMMs and VMs can be hosted on VMMs.

As for the AMF a PLM metamodel is defined to capture the PLM configuration concepts, their relationships and their constraints. The PLM metamodel is based on the PLM specification in [12], but further refines the standard PLM concepts and their relationships. For instance, the PlmEE is specialized into PlmEEVM, PlmEEVMM, and PlmEEOS. The relationship among these concepts has also been redefined: The relationship between the PlmEEVM and the

PlmEEVMM is now defined through the PlmDependency. The PlmEEVMM has an association with its PlmEEOS. The PlmEEOS may have an association with a PlmEEVMM, i.e. host it. These refinements are required to handle appropriately the virtualized environments and cloud architectures. A portion of the PLM configuration metamodel is shown on the right hand side of Fig. 2.4. Multiple layers of PlmHEs may exist in a PLM configuration, e.g. in the configuration shown on the left hand side of Fig. 2.4 there are HEHosts which are hosting the VMMs and the host OS while these Hosts themselves reside on HERacks. We also use dashed lines between each VM and the VMMs that can host the VM. These VMMs are listed in the PlmDependency object for each VM. For the purpose of presentation, the PlmDependency objects and the containment relations between the VMs and the PLM Domain are not shown in the PLM configuration of Fig. 2.4. Each VM is connected to its current hosting VMM, (i.e. one of the VMMs in the PlmDependency object of the VM) with a double line.

Following the same approach as for the AMF UML profile, the PLM UML profile is defined by mapping the PLM configuration metamodel to the UML metaclasses, with the closest semantics.

2.2.4 The Relation between the AMF and PLM Configurations

According to the SA Forum specifications [11,12], each AMF Node is eventually hosted on (mapped to) a PlmEE so that the software entities of the AMF Node can be executed and provide services. This is basically the connection point between the two configuration fragments. In our work we assume this PlmEE is an OS instance installed on a VM instance. Therefore, an AMF Node is mapped to a PlmEEVMM, and this is how the two configurations are put into relation (shown in Fig. 2.5).

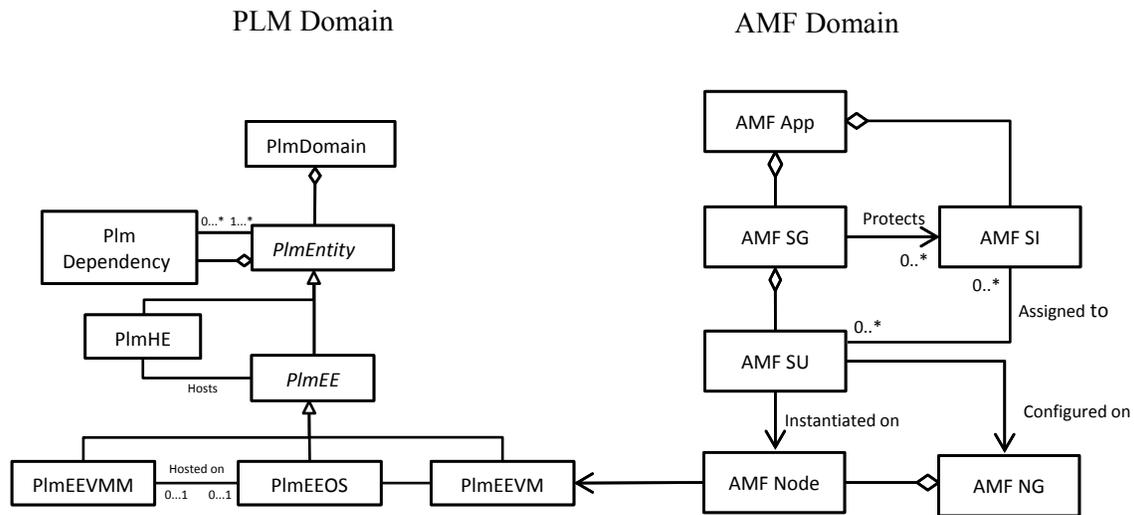


Figure 2.5. The relation between the AMF and PLM configurations

This mapping should be defined in such a way to avoid having a single point of failure due to the failure of the hosting hardware and to ensure that the hardware redundancy is provided for the redundant software entities. This basically means that we should make sure that the SUs of an SG which protect the same service instances, are not hosted on the same HE, so in case that an HE crashes, the other redundant SUs on other HEs can still provide service.

Open Virtualization Format (OVF) standard [19] allows expressing restrictions for VM placements based on HW availability/affinity requirements for the VMs, however the standard does not provide any solution for the correct integration of the configurations.

The feature of ensuring hardware redundancy is an example of a property (availability) that should be taken into consideration for the integration of the configuration fragments. Other system properties such as the affinities between SUs of different SGs can be also considered to target the optimization of the communications between resources.

We use the configuration fragments of the SA Forum middleware as an application domain to illustrate our model-based approach for the configuration integration while considering other aspects or requirements of the system (such as availability, performance, etc.).

2.3 Open Virtualization Format (OVF)

OVF [19] defined by the Distributed Management Task Force (DMTF) [20]. OVF is a packaging standard, which describes an extensible format for the packaging and distribution of software products for virtual systems (i.e. VMs in OVF context). It enables the cross-platform portability by allowing software vendors to create pre-packaged appliances for which the customers can have different choices of virtualization platforms [19].

The upper part in Fig. 2.6 shows the structure of a simple two-tier Petstore appliance on OVF package. It consists of a Web Tier and a Database Tier. The Database Tier itself consists of two Virtual Systems for fault tolerance. So, three Virtual Systems (Web Server, DB1, and DB2) and three Virtual System Collections (Petstore, Web Tier, and DB Tier) are included in the Petstore OVF package. The OVF package definition allows for specifying the deployment of Virtual Systems with specific proximity needs through the definition of placement group policies for the Virtual Systems and Virtual System Collections. The policies are [19]:

Affinity Policy: It is used to specify that two or more Virtual Systems should be deployed closely together, for example, because they need fast communication.

Availability Policy: It is used to specify that two or more Virtual Systems should be deployed separately because of HA or disaster recovery considerations.

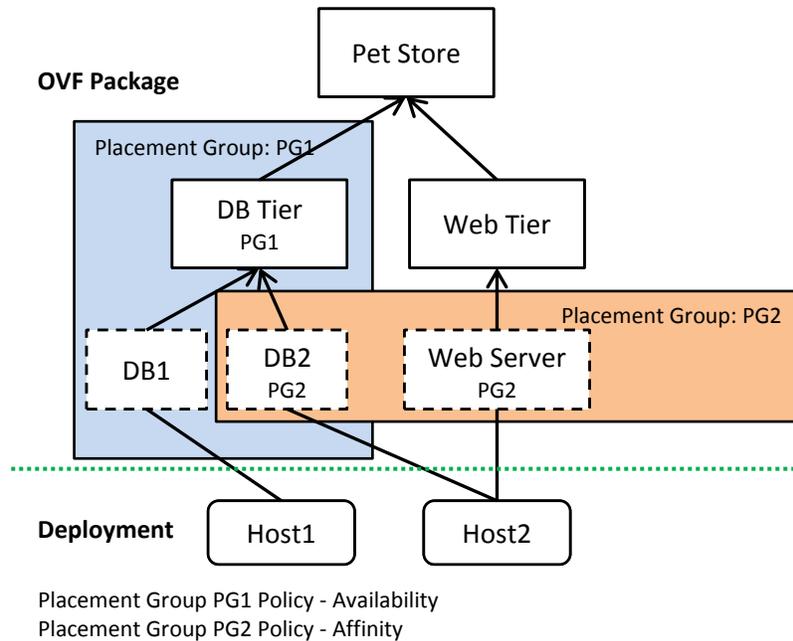


Figure 2.6. The structure of the Petstore OVF package

In the illustrated Petstore appliance of Fig. 2.6 the DB Virtual Systems (DB1 and DB2) should be deployed on different hosts for fault tolerance. Thus the PG1 placement group with the availability policy is specified for the Virtual System Collection of the DB Tier. PG1 is a property of the DB Tier. On the other hand the DB2 and Web Server Virtual Systems should be deployed on the same host for fast communication, so a placement group, i.e. PG2, with the affinity policy is specified for these two Virtual Systems. PG2 is defined as a property for each the DB2 and the Web Server Virtual Systems.

At deployment time the Virtual Systems with their placement groups dictate how they should be deployed on the Hosts that are shown at the bottom of Fig. 2.6. Note that the placement group

may be defined for the Virtual System (e.g. in the Web Server Virtual System), implied by the parent Virtual System collection (e.g. DB1 Virtual System), or a combination of these two cases (e.g. DB2 Virtual System).

The DB Tier has a placement group PG1, which in turn has the “Availability” policy, thus, all the Virtual Systems of the DB Tier (DB1, DB2) should be hosted on different Hosts as shown on Host1 and Host2. On the other hand the placement group PG2 defined for DB2 and for the Web Server has the “Affinity” policy, thus, they should be placed on the same Host i.e. Host2

Although OVF describes the deployment requirements of the applications in a virtualized environment but it is up to the platform managers to maintain the VM placement at runtime as specified by the OVF package.

Chapter 3

Related Work

In this section we review the work related to our configuration management framework in three aspects: configuration generation (integration), validation and adjustment. As most of the related work focuses only on one aspect of our framework (the integration, validation or adjustment), we organize this chapter into three sections, one for each aspect.

3.1. Configuration Generation

The challenge of generating a consistent configuration for large scale systems has been reviewed in the literature from different aspects and for various systems covering proprietary to more general and standardized solutions. Some research trends use the constraint satisfaction techniques and policies for this purpose [39, 40, 41]. For instance Hinrichs, et al. describe their approach in [39] by posing the problem as an object oriented constraint satisfaction problem (OOCSP) and translating it into first order logic. Their OOCSP solver can find the possible solutions (if any exists), otherwise it will run forever because the first order logic is fundamentally undecidable. In addition, although all the possible solutions are returned, some of them might not be desirable

or optimized for certain environments, so a configuration analyzer/optimizer is needed to synthesize and select the best solution.

The idea of data mapping and data integration has been widely investigated in the literature [31, 32, 33]. Defining the mapping between the models and model integration can be seen as the successor of the data mapping research. A number of approaches have defined model management operations (such as merging, subtracting, integration, etc.) focusing on the mapping definition between the models and proposing the operations for manipulating the model mappings and the models for different scenarios.

In Rondo [34] the model management operators, such as merge, match, extract, are defined for solving the mapping problem of metadata in XML schema format. However, the defined operators can only create mappings with respect to a fixed semantics and they are not flexible enough to represent domain specific mappings. A set of generic model management operations are introduced in [6] and the author explains how these operations can be applied to the models and their mappings for different application scenarios. The operations are defined in algebra, while the implementation and execution of the abstract operators are left to the users.

A more specific study on defining model management operations for integrating heterogeneous models is discussed in [35]. The authors introduce weaving and sewing processes and a set of operators for each process. Their input models are the aspect models and the relations between the models are the cross-cutting concepts of the models. Their weaving process is defined by specific operations (i.e. overrides, references, and prune) between entities and using constraints as pre/post conditions for selecting the entities. Their sewing process connects models using me-

diators (defined by synchronize or depend operators) without affecting the model entities. However, their weaving concept is different from what we use in our work and their weaving operators for connecting the model entities are restricted to specific operators such as override, prune, and rename while integrating configuration models may require broader range of connections and also special semantics to link the entities. The integration approach should be extensible to allow the definition of different types of connections with respect to the required properties of the system. We use and extend the weaving concept introduced in [14] which allows for the definition of the extensible corresponding entities that can be translated and executed with model transformations.

In [36, 37, 38] model weaving is used for integrating software architecture models. The mapping links between the entities of the models are created and then filtered based on some similarities, such as type or name, between the entities. Basically, the links are used to map similar entities. In our work however the links between the entities represent the semantics of the relations between the entities and they are more complex and carry target system properties (such as HW availability and/or affinity in the case of AMF and PLM configuration models). We also extend the normal model weaving by extracting the constraint model during the configuration generation (the model integration) process.

3.2. Configuration Validation

The validation of the configuration against runtime changes has been also the topic of many research investigations; it is not only critical for preventing invalid changes that risk the integrity and consistency of the configuration but also because it is a necessity for self-managing systems. Some works in this context focus on the structural checking of the functional configuration pa-

rameters [17, 42, 43] (e.g. type correctness, checking the validity of the values of the system entities' attributes with respect to the constraints of each entity and the relation between the entities. In the SmartFrog configuration management framework [42], the components consist of three parts: the configuration data, the life cycle manager and the functionality of the component itself. Constraints of each component are considered within its configuration data by attaching the conditions as predicates on a description. For combining the components the configuration data should be extended and the conditions are propagated and additional predicates may be added grouping the old and new predicates. The component developer is in charge of defining the conditions (restrictions) for the components and their combination in the configuration data templates. The authors indicate that the validation of the configuration data happens by checking these conditions; however they do not mention how the conditions are checked. In addition the constraints for combining the components can be expressed as simple conditions but it might not be possible to describe more sophisticated constraints (coming from special requirements of the domain) with the conditions in the configuration data templates.

Akue et al. proposes a solution for dynamic reconfiguration by considering the validation of the structural integrity and runtime changes in [44, 45]. They use the predefined constraints for validating the requested modification on the structural and current operational conditions of the system. Their architecture has a model repository (storing the reference model and the constraints), and an online validator for performing the dynamic constraint evaluation. Their online validator receives the configuration modification requests and the current system states as input and validates the requests by checking the configuration instance against the reference model and the constraints. However their exhaustive validation (checking all the system constraints) for large

configuration models can degrade the validation performance especially when the changes are small in scope or the number of constraints increases. In our work we check the structural integrity and also validating the affected constraints by the changes.

Existing approaches for *re-validating* models after changes also try to reduce the number of constraints and/or the model entities that need to be checked. In [46, 47] a list of events that can violate the OCL constraints is defined and added to their configuration schema to be able to check the constraints only if changes are related to these constraints and only on entities that are relevant. Their approach cannot handle complex constraints especially the ones with recursive, loops or complex iterations. Bergmann et al. use a query language (IncQuery) on EMF models in [49] based on graph pattern formalism. The queries are permanently stored in memory and they update the values of the partial matches used in queries after each model change. Thus their approach has considerable memory consumption. In [48] an approach for incremental validation of the OCL constraints is explained. They store the validation log of checking each constraint over the model entities. A re-validation is triggered when the stored parts are changed. The authors extend their work in [50] to improve the performance of constraint re-validation by checking parts of the constraints that are affected by changes (to avoid checking all the constraints). They achieve better performance but by sacrificing more memory.

In our partial validation approach we also select the constraints that are affected by changes. In addition we categorize the selected constraints based on the role of the changed entities in those constraints. This allows us to make a distinction between the potential inconsistencies that have the chance to be resolved by the adjustment mechanism from those potential inconsistencies which cannot be resolved.

3.3. Configuration Adjustment

Model refactoring/refinement/evolution has been widely investigated [51, 52]. This branch of the modeling has a close relation with software evolution, it deals with changes in the models that may occur for many reasons, such as adding new functional requirements, improving some quality aspects, or adapting to a new technological or architectural environment [51]. Our work is different from the conventional model refactoring problems as we deal with the runtime (or instance) models. The goal is to find a way for adjusting the changed models with respect to the model constraints.

Constraint solving is used widely in the literature for configuration generation and its adaptation [54, 55, 56, 57]. In [57] the authors propose a range fix approach that is based on constraint solving. Instead of finding a specific value for a configuration entity/attribute they find ranges (options) that fix the violated constraints. Although the ranges give the user more options to choose from, but still the user needs to have knowledge about the configuration so he/she can select values from the ranges. In our approach we automate the configuration adjustment to decrease the user's involvement and risk of inconsistency. We also try to minimize the adjustments not to destabilize the system at runtime. Authors in [56] use constraint solving to automate the configuration generation in Software Product Line (SPL) [30]. They follow a multi-step approach using the feature model and a set of constraints for selecting the features (constraints such as cost or priority). At the end of each step a valid configuration with a subset of features is created and the desired target configuration is obtained in the last step. The configurations are created offline, and large modifications may be applied in each step. Sawyer et al. [55] combine goal modeling with constraint solving for creating configurations for SPL that meet QoS requirement. They also

indicate that the result is also useful for runtime adaptation. Neema et al. [54] also propose a constraint guided adaptation framework that formalizes the non-functional requirements of the system as constraints. A symbolic constraint satisfaction method based on Ordered Binary Decision Diagram is used to find a solution. Compared to our approach we handle consistency of the reconfigurations at runtime thus we are concerned about minimizing the modifications.

Considering the user preferences in self adaptive systems is discussed in some studies [26, 53, 58]. Poladian et al. [58] use a utility function to formulate the user preferences into an optimization problem for dynamic configuration of resource-aware services. In [26] the user preferences are considered to adapt the runtime models. Their objective is to solve the CSP by satisfying as many constraints as possible. After diagnosing the interrelated constraints, less important constraints (with lower weight) are ignored to satisfy the remaining constraints. Users can revise the model or modify the weight of constraints in order to express their preferences. In our work constraints cannot be ignored, however our approach directs the adaptation (propagation) to relax the problem for interrelated constraints. Our adjustment also aims to reduce the role of user in the process. Instead the role (impact) of the entities in relation to other entities is the key feature in our adjustment process.

Fixing the inconsistencies of refined models has also been the subject of many research studies. In [59] authors propose an approach to identify a set of valid choices (values) for each model entity/attribute through incremental consistency checking. They argue that the result is a set of choices which fix the initial inconsistency while it does not violate any other constraint. However it is not always possible to find the set of valid values or the set includes numerous members (e.g. for the attributes with integer, string datatypes). Moreover the authors did not mention how

to solve interrelated constraints. Nentwich et al. designed repair semantics which maps the constraints to repair actions [60]. Their approach cannot handle interrelated constraints and the repair action for one constraint can violate another constraint. Xiong et al. defined a language similar to OCL for the constraints and also to define the fixes in case of violation of each constraint [61]. However for defining the fixes, the developer needs to consider all the relations between the constraints and consider them in the fixes. In addition analyzing numerous invalid values of the model entities requires defining numerous fixes. These challenges make the development of the fixes very complicated, especially as the number of constraints increases.

3.4. Summary

Considering the configuration generation as a CSP and solving it with constraint solvers can provide correct configurations but they require further work to analyze and select among the solutions. Various model management operations defined to solve the mapping problem between the models; however they introduce fixed operations or static links and do not consider the special relations or semantics between the models. To ensure the consistency of the configuration at runtime against changes, a configuration validator is required. However the exhaustive validation techniques can be time and resource consuming especially when the changes are small and/or the configuration is large and has a large number of constraints. Some proposed approaches in the literature try to reduce the validation scope e.g. by keeping a list of events that can violate the constraints or keeping the queries permanently in memory. They achieve better validation time but by sacrificing more memory. Runtime adjustment of the configuration is used for consistency preservation against incomplete change requests. Some approaches provide certain valid ranges for the configuration entities and involve the users to select among these ranges. This requires

that users have knowledge about the relations and constraints of the system. Other approaches give weight to the system constraints and through their configuration adjustment they may ignore the low weight constraints. This is not always possible as all the constraints are important to be maintained valid especially in HA systems. Other proposed approaches define all the invalid cases that may happen by the change requests and for each case they also have a predefined fix. This is not applicable in large systems with large number of constraints as so many invalid cases may happen.

To address the mentioned issues, we proposed a configuration management which consists of the management operations required for handling the system configuration from its generation (integration) through runtime changes with validation and adjustment. Besides the solution proposed for each of these problems, the other advantage of our framework is that each phase can benefit from its predecessor, e.g. when we generate the configuration, the constraints can be extracted automatically and used in the validation phase and there is no need to create the integration constraints manually. Similarly, the adjustment phase uses the validation results. We intend to automate the configuration management process and therefore decrease the administrative efforts and also the risk of errors in the system due to potential inconsistencies at configuration design or reconfiguration times.

Chapter 4

Configuration Management Framework

In this chapter we introduce our model-based framework for configuration management which includes a module for configuration generation at system design time and a module for runtime change management which itself consists of configuration validation and adjustment parts. In this chapter, we briefly introduce our modeling framework and the different parts of our configuration management framework. We elaborate further on each of the contributions in the following chapters.

4.1 Introduction

A configuration is a logical representation of the system resources, their relations and dependencies. A system may have several configurations (e.g. platform, security, software management...) each of which focuses on a specific aspect of the system which might be developed separately. These configuration fragments need to be integrated at design time to form a consistent

system configuration to avoid system malfunctions. The consistency of the system configuration should also be preserved at runtime against the changes.

Fig. 4.1 illustrates the overall view of the model-based framework for configuration management to generate a system configuration at design time by integrating the configuration fragments and also to manage the runtime changes in order to preserve the configuration consistency.

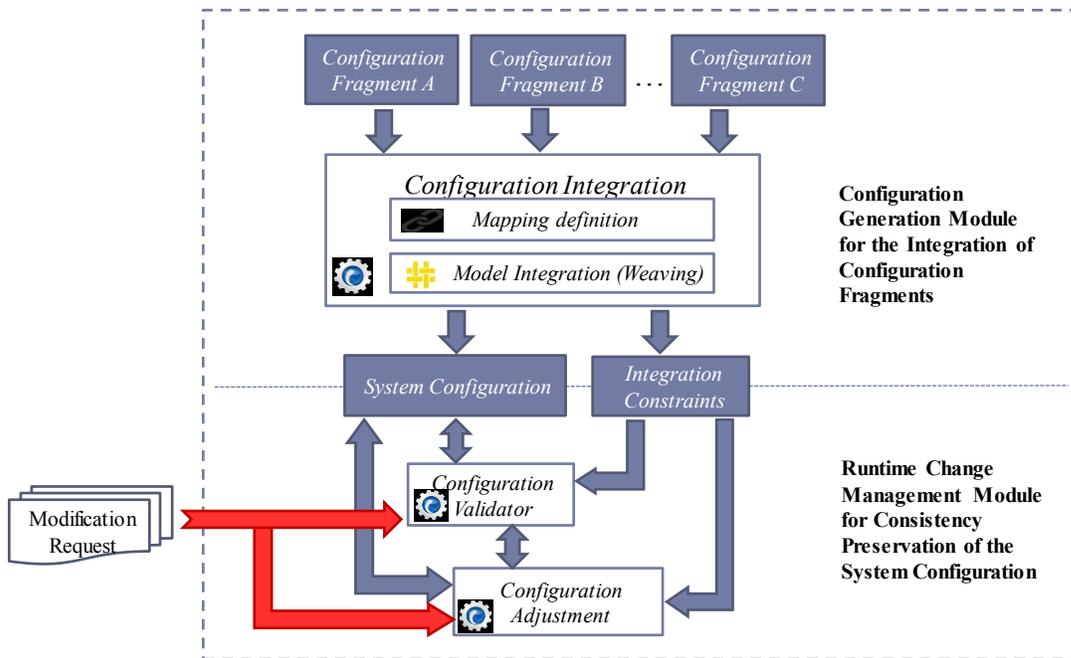


Figure 4.1. Overview of configuration management framework

For the representation and manipulation of each configuration fragment, a configuration schema is usually used to specify the correct structure of the configuration entities, their relations and the constraints. To be valid and consistent, a configuration must respect its schema, i.e. satisfy the structural and semantic constraints of the schema. The constraints are rules which force special

restrictions on the entities of the configuration and their relations. We use UML [3] and its profiling mechanism to capture the concepts and relations of a configuration in a configuration profile. The system constraints are added to the profile as OCL [5] constraints. We enrich the OCL constraints and use them in our configuration management framework to represent the system consistency rules.

4.2 Extending OCL

OCL is a declarative language to describe the constraints for UML models and profiles. It specifies what conditions should be met by the entities involved in the OCL expressions rather than how they should be achieved. Although the standard OCL is suitable for many applications, it is not always sufficient. By extending OCL we can add more information to the constraints. An example of OCL extension is the addition of severity and descriptions to the constraints as explained in [62] to provide a more understandable and precise representation of the constraints.

The configuration constraints are restrictions on the attribute values and relations of the configuration entities. They are defined when the configuration schema is designed to reflect the requirements of the system/application domain. In the case of configurations, a constraint puts some restrictions on some entities but it does not characterize the role of these entities in the constraint: I.e. if some entities in the constraint influence the others. These dominant and dominated roles of entities cannot be expressed by standard OCL.

To illustrate the influence of the constrained entities on each other we present an example with respect to the OVF standard. Fig. 4.2 illustrates the relation between the entities of a simplified OVF domain model. The restrictions that the policies impose on the deployment of Virtual Sys-

tems are expressed with the OCL constraints included in the figure. Note that OVF allows for a combination of availability and affinity policies, but we have shown a simple example, which serves our purpose.

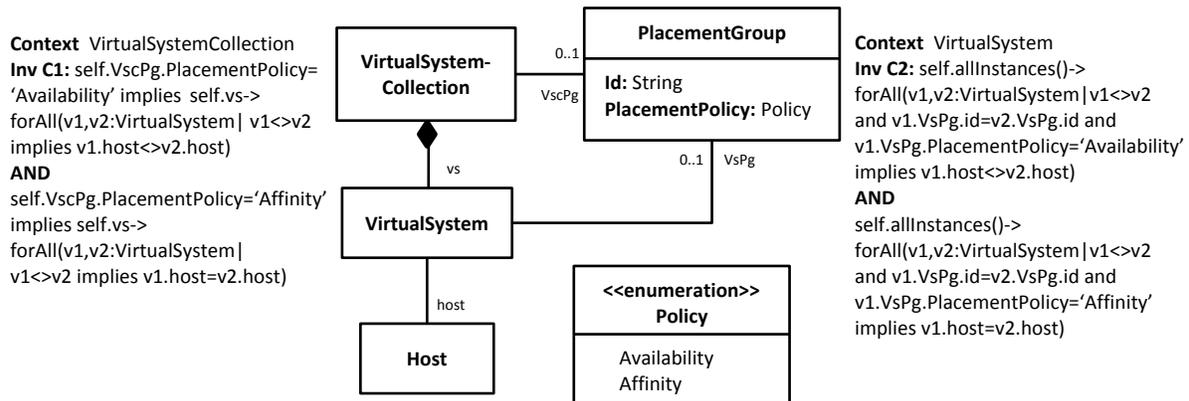


Figure 4.2. Partial model of the Virtual Systems, their collection and placement policy in an OVF package

An OCL constraint captures the restrictions on the relation between the Virtual Systems and their Host(s) imposed by the placement group policies. However an OCL constraint cannot capture the role of the Virtual Systems in the constraint as determining the Host entity selection. In the relation between the Virtual System and the Host entities, the Virtual System entity has a leader role and drives the Host selection, that is, the follower. This means that if the Virtual System entity (including its PlacementGroup) changes and the constraint becomes violated, the Host of the Virtual System should change too to follow the Virtual System change and satisfy the constraint. On the other hand if the Host of the Virtual System changes and this change violates the constraint, the Virtual System and its PlacementGroup cannot be changed as the role semantics does not allow the leader entity to adjust to the changes of the follower entities. For instance in the Petstore example in Fig. 2.6 of Chapter 2, the DB Tier (i.e. DB1, DB2) and the Web Server Tier are the leader entities while Host1 and Host2 are the followers. Now let us assume that Host1

fails. Since PG1 does not allow the collocation of DB1 and DB2, DB1 cannot be re-deployed. On the other hand if we want to change the placement group of the DB Tier from PG1 to PG2, this change of the leader results in changing the Host entity (follower) which means that now DB1 and DB2 should be placed on the same Host (Host2).

As the standard OCL cannot express these roles for entities, we extended the OCL by defining roles for the constrained entities to show the influence of some entities over others in the constraint. Considering the semantics of the relations between the entities we can identify a leadership flow between them. In other words in a constraint with multiple entities involved, changes in some entities (*Leader*) may impact the others (*Follower*). In other relations where the entities have equal influence over each other, we call them *Peer* entities.

Fig. 4.3 shows the extension of the constraints with the leadership information. We added this small extension to the OCL to enrich the constraints without changing the OCL grammar and metamodel so the parsers and validators designed for the standard OCL remain usable. We consider OCL together with our extension as our constraint profile.

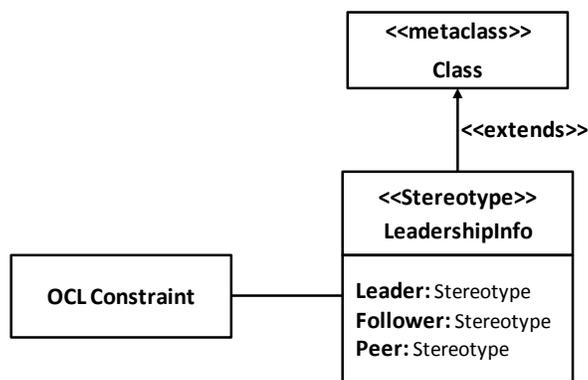


Figure 4.3. OCL constraint enriched by the leadershipInfo

The role of the entities in the constraints of OVF example is shown in of Fig. 4.4. In this figure constraints are shown as ovals. The participation of each entity in a constraint is represented by an edge between the constraint and the constrained entity. The role of the entity in the constraint is shown as a label on this edge (e.g. label “L” represents the Leader role). This representation focuses on the role of entities in the constraints and depicts how the constrained entities can affect each other.

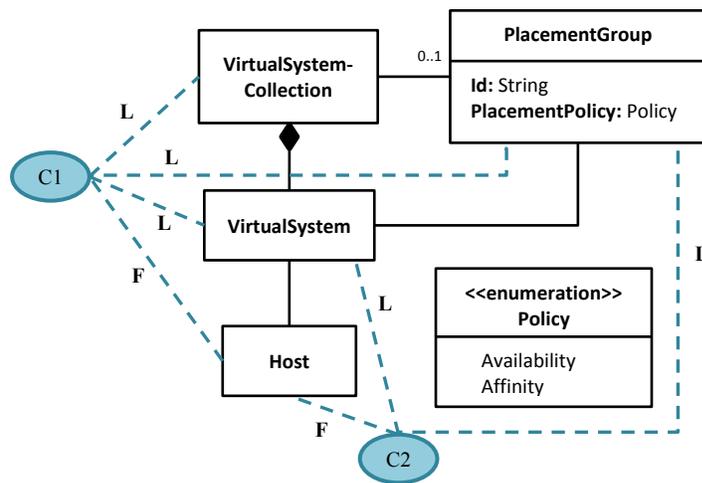


Figure 4.4. Representation of entity roles in constraints

It is worth mentioning that the roles of the constrained entities may change with the application scenario. More specifically we may define the leader/follower/peer roles for the entities differently for design time and for runtime. At design time we generate the configuration according to an optimal design method. Once the system is deployed we may be limited in the changes allowed. For example due to budget reasons we may not add new hosts to the system and as a result we want the Virtual Systems (including the software products) adopt and follow the Host restrictions in this respect. This means that now the Host entity becomes a leader and the Virtual System and Virtual System collection are followers. Note that the standard portion of the OCL

constraint between them remains unchanged. Defining the roles for the entities through the leadership information has this advantage that we can define and change the roles whenever it is needed without affecting the constraints themselves. In the next sections we introduce each part of our configuration management framework and the usage of the leadership information in that part.

4.3 System Configuration Design: Integration of Configuration Fragments

Large systems are usually built by integrating independently developed components. Each component may have its configuration which describes the resources managed/controlled by the component. Thus, the system is described through the various configuration fragments. These configuration fragments need to be integrated together to form a consistent system configuration to ensure smooth and correct operation of the system. The system configuration should ensure that the resulting system meets the required properties. The integration of configuration fragments is a challenging task i.e. due to overlapping entities (different logical representations of the same system resource) in the configurations and/or complex relationships among the entities of the different configuration fragments.

Extending and using the model weaving and model transformation techniques, we develop an approach for the integration of configuration fragments targeting specific system properties. We define the semantic of the relations between the entities of the configuration fragments as links at a higher level of abstraction which has several advantages:

- It increases the reusability of the defined links (relations) to map other entities between configurations.

- It allows adding/modifying the interpretation of the links and embedding them into the integration process without modifying the links.
- It is easily extendible as various configuration profiles can be added to the integration process using predefined or new links.
- The integration process is automated. The system configuration can be generated automatically with the same rules for different input configurations.

We also define the integration semantics as integration constraints and add them to the system configuration profile. The integration constraints (describing the semantics of the relation between the fragments) in addition to the union of the constraints of the fragments form the system configuration constraints which guard the consistency of system configuration models against unsafe runtime modifications.

Our proposed approach for the configuration integration and constraint generation is discussed in detail in Chapter 5.

4.4 System Runtime: Consistency Preservation with Change Management

A system reconfiguration may be performed for many reasons, such as in response to environment changes or users' requests for fine-tuning. These changes should not endanger the consistency of the configuration. To manage configuration changes we propose the architecture shown in Fig. 4.5 which includes a configuration validator to check the change requests and an adjustment agent that attempts to add complementary modifications to resolve the potential inconsistencies detected in the validation phase.

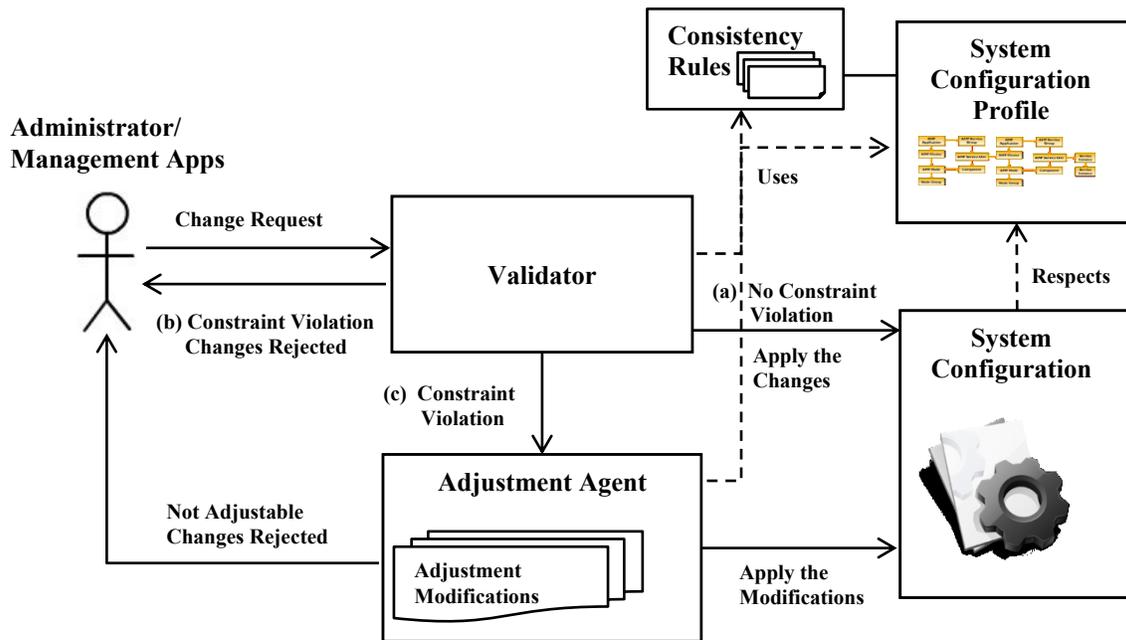


Figure 4.5. Overall view of the configuration validation and adjustment process

4.4.1 Runtime Configuration Validation

At runtime the administrator or the management applications may need to change the system configuration to control/manage the resources under their management. These changes must not endanger the consistency of the system configuration and jeopardize the system's operation. Thus the requested changes should be checked and the modified configuration needs to be validated to guarantee its consistency. A configuration validator is responsible for performing the validation with respect to the system configuration profile and its constraint. As shown in Fig. 4.5, the validation result would be one of the three following cases:

- (a) The requested changes do not violate the configuration constraints and respect the profile.

Therefore the changes are safe to be applied in the system configuration.

- (b) The requested changes violate one or more constraints of the profile and these violations cannot be resolved as there is no chance to propagate the changes to other entities of the violated constraints to resolve the violations. Thus the requested changes are rejected.
- (c) The requested changes violate one or more constraints of the profile; however the changed entities/attributes can impact other entities/attributes of the violated constraints. Therefore there may exist a chance to resolve the constraint violation by changing other constrained entities. Thus the result of the validation will be passed to an adjustment agent.

The decision of rejecting the requested changes (i.e. case b) or trying to adjust the configuration (i.e. case c) is made based on the ability of the changed entities to impact other entities of the violated constraints. The possibility of the impact is determined regarding the leadership information. Our proposed configuration validation method is discussed in detail in Chapter6.

4.4.2 Configuration Adjustment

Although a configuration validator can detect the constraint violations caused by unsafe/incomplete requested changes, such violations or conflicts might be resolvable by adding complementary modifications that complete the initial set of changes.

In order to resolve such inconsistencies the initial changes need to be propagated in the configuration and other entities that are related to the changed entities should be changed respectively to satisfy the violated constraints. This propagation is done with respect to the system constraints and by following the impacts of the configuration entities on each other (i.e. defined through the leadership info of the constraints).

In our proposed framework this task is done by the adjustment agent which is shown in Fig. 4.5. It takes the validation result from the validator and uses the system configuration profile and constraints. If a set of complementary changes can be found that along with the requested changes satisfies all the constraints, the adjustment is successful and the changes can be applied to the configuration. Otherwise the initially requested changes are rejected. In Chapter 7 our adjustment approach and its contribution is explained in more details.

4.5 Summary

In this chapter we described a model-based framework for configuration management to 1) integrate configuration fragments in a consistent manner at system design time and 2) maintain the consistency of the configuration at runtime using a configuration validation and an adjustment technique.

In this model-based framework for each configuration fragment, a configuration profile is used to capture the configuration entities and their relations (system structure). The constraints among the system entities are expressed as OCL constraints. These constraints have been extended so that they capture the roles of the configuration entities in the constraints. Namely, the leader/follower/peer roles define which entity can impact the other ones in the constraint. This leadership information can be derived from the configuration integration and be used at runtime for consistency preservation of the system configuration at runtime.

To integrate the configuration fragments into a consistent system configuration we use an extended model weaving technique and define the semantics of the relations between the entities of the configuration fragments at a higher level of abstraction which increases the reusability, ex-

tensibility of the approach. To manage the configuration changes at runtime we use configuration validation to verify the validity of the changes against the configuration constraint. If a violation is detected, we try to adjust the configuration by adding the complementary changes to neutralize the potential inconsistencies detected in the validation phase. In the next chapters we explain each contribution of the framework in details.

Chapter 5

The Integration of Configuration Fragments

In this chapter first we introduce the challenges of the integration of AMF and PLM configurations as motivation example and then, we describe the overall integration approach i.e. based on model weaving concept and describe our extension. We also describe our method to automate the generation of the integration constraints that reflect the relations between entities from the different fragments. Finally we discuss the implementation and the results reported in this chapter.

The contents of this chapter have been published in [27, 28, 71].

5.1 Introduction

As mentioned earlier a system may be described through various independently developed configuration fragments. One of the main challenges of such composite systems is the integration of these configuration fragments in a consistent manner that reflects the relations and constraints between the entities of the different fragments and ensures that the resulting system meets the

required properties such as availability, performance and security. The complexity of this integration task stems from the potential overlap between entities of the different configuration fragments (i.e. different logical representation of the same physical entity) and from the complex relationships among the entities of the different configuration fragments. The integrated system configuration should reflect properly the relations and constraints between the entities of the different fragments and ensure that the resulting system meets the required properties of the system, like availability, performance, security, etc.

We tackle the problem of integration of configuration fragments with a model-based approach based on the concept of model weaving [14]. Model weaving allows for relating different models – in our case representing configuration fragments by defining links between their entities. These links form a weaving model which conforms to a weaving metamodel. Model weaving has been widely used for model integration, model transformation, model merging, etc. [6, 35, 36, 37]. However the focus so far has been primarily on the static mapping of entities without considering the semantics of these relations. In our approach we take into account the semantics of these relations and target some desired properties of the resulting system configuration model. Our approach generates a consistent system configuration model which contains all the entities from the configuration fragment models, the constraints of each configuration fragment as well as the constraints reflecting the desired properties of the integration. The latter are generated automatically to capture the targeted properties entailed by the weaving links.

We illustrate our integration approach in the context of the SA Forum middleware. The middleware consists of several services and frameworks, which represent and control specific aspects of the system and collaborate with each other [7]. We focus on the configurations of two SA Forum

services: AMF [11], and PLM [12] which have been introduced in Chapter 2. The configurations for these services are described using UML profiles. We capture the structure and semantics of the relations between these profiles in a weaving model, which is later used to generate the system configuration. Defining the relations between the profiles at a higher level of abstraction through a weaving model has several advantages such as reusability of the link types, increasing the extensibility (by allowing more models to be added) and automating the integration process [14].

5.2 The Challenges

In this section we first discuss the main challenges of the integration before introducing our model-based integration approach.

5.2.1 Overlapping Entities

A configuration fragment is a logical representation of the resources and their organization for the management purposes. A resource may exist in multiple configuration fragments with different logical representations. An example of a resource with multiple representations is a Virtual Machine (VM). As shown in Fig. 5.1 a VM is represented in the AMF configuration as an AMF Node and the same VM in the PLM configuration is represented as an EEVM.

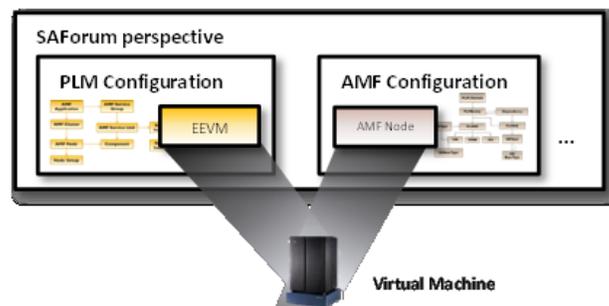


Figure 5.1. Different representations of a Virtual Machine in different configuration fragments

Managing or modifying the overlapping entities (e.g. the entities with multiple logical representations) independently in each configuration fragment will lead to inconsistency in the system as they all affect the same resource. Thus, these logical representations of the same entity need to be related.

5.2.2 Integration Relations between Configuration Fragments

The integration of configuration fragments usually targets certain properties for the system configuration. These properties depend and may involve more than one aspect of the system and thus require the capturing and the description of the required relations between these aspects. Let us consider the AMF & PLM configurations again and explain the required relations between them further.

According to the SA Forum specifications [11,12], each AMF Node is eventually hosted on (mapped to) a PlmEE so that the software entities of the AmfNode can be executed and provide services. This is basically the connection point between the two configuration fragments. In our work we assume this PlmEE is an OS instance installed on a VM instance. Therefore, an AMF Node is mapped to a PlmEEVM, and this is how the two configurations are put into relation. The question is whether any mapping between the AMF Nodes and the PLM EEs is acceptable? We hereafter address this question through some examples explaining the specific property of the system that should be targeted in the integration of configuration fragments.

Hardware Disjointness of Service Providers for Enabling Availability

Fig. 5.2 shows a simple example in which an AMF configuration is put into relation with a PLM configuration by mapping AMFNode1 and AMFNode2 to EEVM2 and EEVM1, respectively.

These two VMs are running on the same VMM (EEVMM1) and PLM HE (HEHost1). At this point the HEHost1 as well as the EEVMM1 represent single points of failure. If this HEHost1 crashes both service providers, SU1 and SU2 will be lost and a service outage will be inevitable. Even if in the initial PLM configuration the VMs (EEVM1 and EEVM2) are hosted on different EEVMMs, at runtime the VMs may migrate and end up on the same VMM and HE at the same time. So, if the goal is to avoid any single point of failure due to the hosting hardware, we need to make sure that the service providers (SUs) of an SG will never be hosted on the same host.

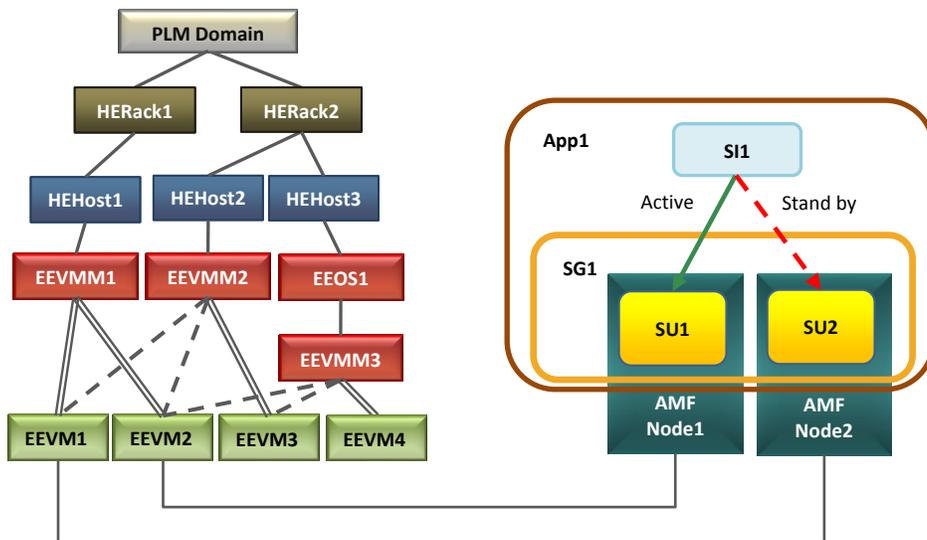


Figure 5.2. Host failure problem because of the relation between the AMF and PLM configuration fragments

Hardware Affinity of Service Providers for Fast Communication

As mentioned earlier in Chapter 2, AMF manages redundant service providers (SUs) to avoid service outage due to SU failure. When the SU with the active assignment fails, AMF shifts the active assignment to the standby SU. To be able to use the standby SUs, the state of the active and the standby providers need to be synchronized so that in case of service failure the assign-

ment of the service can be shifted without any service interruption. The active and standby SUs of an SG need to synchronize continuously and this state synchronization introduces some communication overhead causing latency in the normal behavior. The latency increases when the hosts of the SUs are farther from each other. E.g. in Fig. 5.3 SU1 is eventually hosted on HEHost2 and HERack2 while SU2 is eventually hosted on HEHost1 and HERack1. As the two SUs are residing on different HERacks, the latency is higher compared to the configuration in which the SUs are on the same HERack. Therefore, to assure an efficient communication (state synchronization) among the SUs of an SG and reduce this latency, the SUs should be placed closely together.

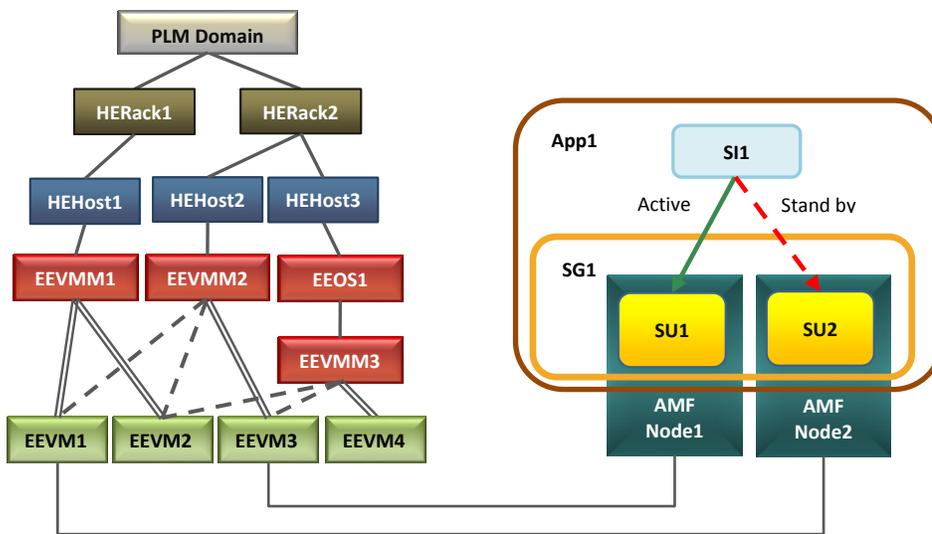


Figure 5.3. Latency problem because of the relation between the AMF and PLM configuration fragments

A combination of hardware availability and affinity

Each of the previous examples (hardware disjointness or affinity) shows an example of a property that may be targeted by a particular approach of integration of the configuration fragments. Moreover, more complex properties may be required such as the conjunction of both hardware

affinity and disjointness, i.e. the SUs should be hosted on different hosts but the hosts should also keep certain proximity such as being in the same rack or site to assure the fast synchronization among the redundant SUs.

The hardware disjointness and affinity relations are examples of the properties which need to be described and enforced by the integration to ensure properties like availability or lower latency for the system. The relations between the configuration fragments need to be defined properly and according to the required properties. These relations should be enforced by the integration to define a consistent system configuration that exhibits the targeted properties.

5.3 The Overall Approach

To integrate configuration fragments we use and extend the model weaving technique. In this technique a model called the *weaving model* is used to capture the mappings between the entities of the metamodels. As any model in the model driven paradigm the weaving model conforms to a metamodel, i.e. the *weaving metamodel*. The weaving metamodel describes the types of mappings that can be used in the weaving model. It also describes the types of entities which can be connected through these mapping types, i.e. the link end types. The instances of the mapping types (or link types) are used in the weaving model to connect the models'/metamodels' entities.

As discussed earlier, for the integration of configuration fragments we need to capture more complicated relations among the fragments than just the entity mappings. Therefore, we extend the weaving concept in order to capture the semantics of the relations among the configuration fragment entities and use this semantics for the integration of the fragment models. In our integration approach the configuration fragments and their metamodels are represented as the source models and source metamodels. For example, the AMF and PLM configuration models are the

source models and their UML profiles are the source metamodels. We also use a system configuration metamodel that is called the target metamodel and at this stage it is a union of the source metamodels without any relationship between them. Through the weaving we integrate the source models and generate a system configuration, i.e. the target model. We extend the weaving metamodel with special link types and we create a weaving model by defining the links between the configuration entities of the source and target metamodels. The weaving model is a static representation of the relations among the entities; therefore it is translated to an executable format using a Higher Order Transformation (HOT) [66]. The result of the HOT transformation is another transformation called the Final Transformation which takes some source configuration models (e.g. the AMF and PLM configuration models) as input and generates a target configuration model (i.e. the system configuration model) as output. The overall process of the configuration integration through model weaving is shown in Fig. 5.4. In the following we summarize this process.

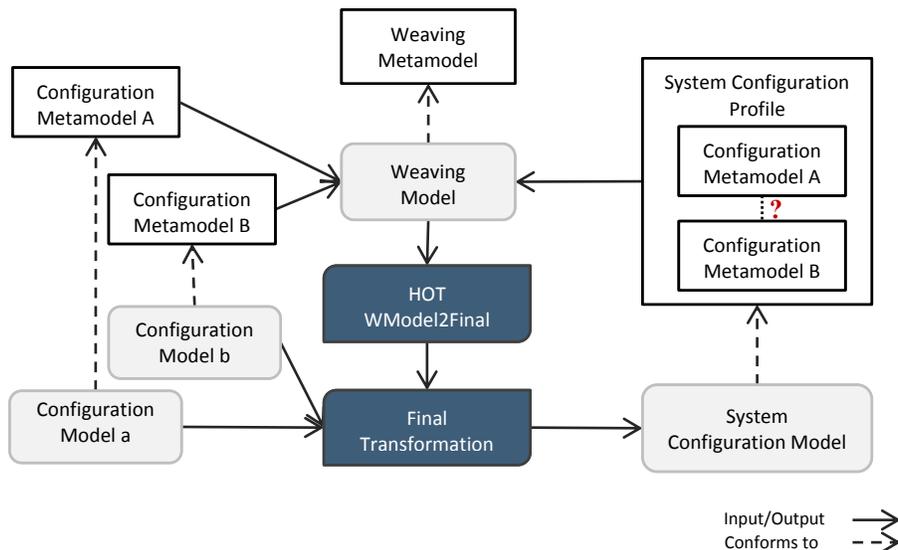


Figure 5.4. The system configuration generation through model weaving

5.3.1 Extending the Generic Weaving Metamodel

As mentioned earlier a weaving metamodel defines the link types and the link end types that can be used in the weaving model. Fig. 5.5 shows part of a generic weaving metamodel [14, 63] represented with lighter color elements. We extended this metamodel in order to capture the special relations between the configuration fragments. The elements extending the metamodel are shown in darker color in Fig. 5.5. In the following we explain these extensions in more details.

5.3.1.1 WLinkEnd Specializations

SourceEnd and TargetEnd

In the generic weaving metamodel the WLink represents the generic link type which maps the WLinkEnds. For configuration integration we need to add a direction to the links and distinguish the source and target ends of the links as we have source models as input and we want to create the target model as output. Therefore, we consider the WLinkEnd as an abstract class and specialize it into the SourceEnd which is used to represent the configuration entities from the source models and the TargetEnd to represent the created/modified configuration entities which will appear in the target model (i.e. the system configuration). To make sure that in each link we have at least one SourceEnd and one TargetEnd constraint C1 is defined on the WLink. This constraint is expressed in OCL as:

```
Context WLink
Inv C1: Self.end->exists (e1, e2: WLinkEnd | e1.ocIsKindOf (SourceEnd) AND
e2.ocIsKindOf (TargetEnd))
```

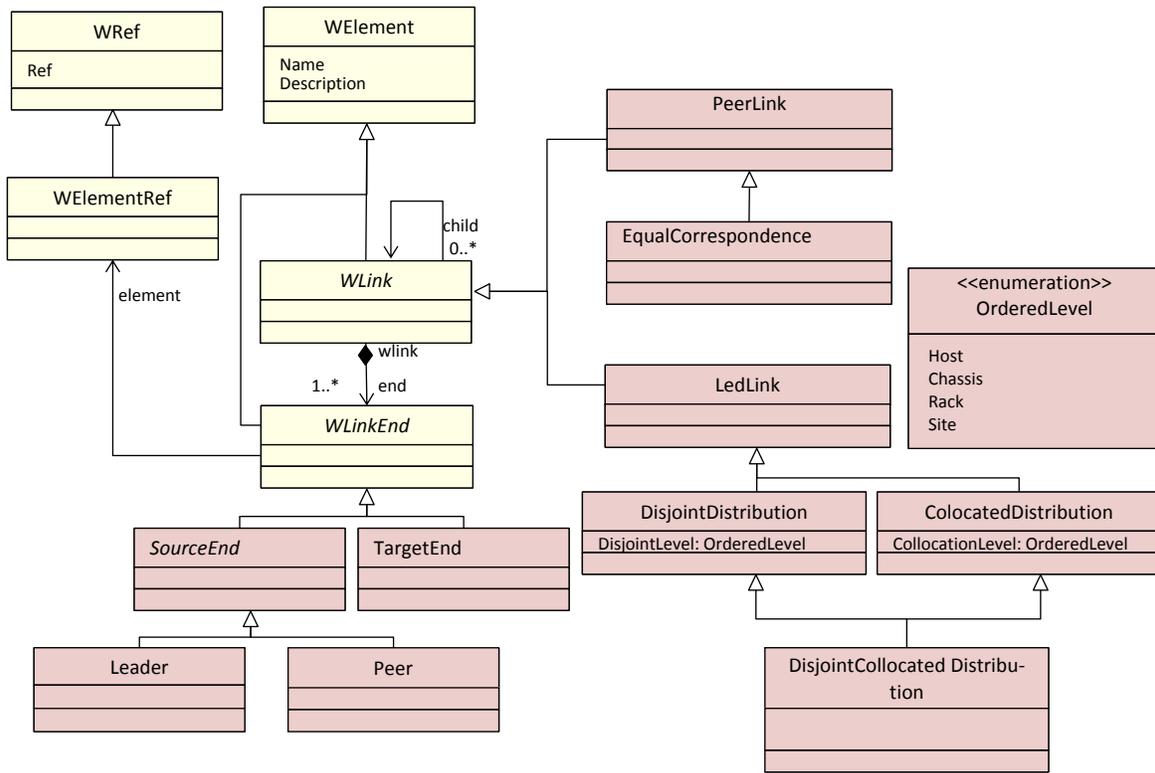


Figure 5.5. The generic weaving metamodel extended with new LinkTypes and LinkEnds

The entities of the source configuration metamodels that are specified as the SourceEnd are called the Source entities. The entities of the target metamodel appear in the TargetEnd and are called the Target entities. They are linked to the Source entities by the WLink.

Leader and Peer

The SourceEnd is specialized further into *Leader* and *Peer* link ends in the weaving metamodel to capture the influence of the configuration entities on each other. More specifically when a configuration entity is specified as a Peer and it is linked to a Target entity it means that the Target entity is created/modified with respect to the Peer Source entity (or Peer entity for short) however if the Peer entity also appears in the target model (i.e. created through the same or an-

other link), these entities (the Peer and the Target entities) would have equal influence on each other in the target model. In other words if either of them changes later in the target model, it can impact the other one.

Similar to the Peer link end type, the Leader link end is another specialization of the SourceEnd. Configuration entities specified as Leader Source entities (Leader entities for short) also create/modify the Target entities but in contrast to the Peer entities, if the Leader entities appear in the target model (i.e. created through other links), only the Leader entities can influence the Target entities in the target model and not the other way around. This means that later in the target model if the Leader entities change, this change impacts their created/modified Target entities. The Target entities can follow the changes of the Leader entities but if those Target entities change, they cannot impact the Leader entities.

5.3.1.2 WLink Specializations

PeerLink

The *PeerLink* represents the relation of the Peer Source entities and their Target entities. Defining a PeerLink among the Peer and Target entities means that even though the Peer entities are used to create/modify the Target entities the relation is not unidirectional. In the target model the relation is bidirectional, that is, the Target entities can have equal impact on the Source entities and vice versa. They are all in a Peer relation with respect to the constraints implied by the creation/modification rule.

A structural constraint, C2 is defined for the PeerLink to assure that the PeerLink has only Peer link end as its SourceEnd.

```
Context PeerLink
```

```
Inv C2: Self.end->forall (e: WLinkEnd | e.ocIsKindOf(SourceEnd) implies  
e.ocIsTypeOf(Peer))
```

EqualCorrespondence

In the configuration integration it happens that many entities from the source models are just copied to the target model. The *EqualCorrespondence* link type, inspired by [37], is defined to map the Source entities to their identical Target entities. EqualCorrespondence is a specialization of the PeerLink so the Peer link end is used as the SourceEnd for this link type and the TargetEnd is the other link end for this link type.

LedLink

The *LedLink* represents the relation of the Leader Source entities and the Target entities. It means that when a LedLink is defined among the Leader and the Target entities, the Leader entities create/modify the Target entities and such impact or affection among the entities (i.e. Leader entities impact the Target entities) needs to be maintained in the target model among the involved entities.

A structural constraint, C3 is defined for the LedLink to assure that the LedLink has only Leader link end as its SourceEnd. This constraint is expressed in OCL as:

```
Context LedLink
```

```
Inv C3: Self.end->forall (e: WLinkEnd | e.ocIsKindOf(SourceEnd) implies  
e.ocIsTypeOf(Leader))
```

DisjointDistribution

The *DisjointDistribution* link type is defined to capture the hardware-disjointness property for the target configuration. DisjointDistribution is an extension of the LedLink and therefore the

Leader link end and also the TargetEnd needs to be specified for the link. This link type has an attribute called *DisjointLevel* of an enumeration type *OrderedLevel*. The *OrderedLevel* enumeration has the items of Host, Chassis, Rack, Site, and Geographic which define the levels of disjointness that are required for the configuration entities. E.g. for the scenario we explained earlier, if the *DisjointLevel* attribute is set to Host, then the linked entities should be configured on different Hosts. If this attribute is set to Rack, for instance, the linked entities must be configured for different Racks. The values of the *OrderedLevel* type are defined according to the OVF specification [19].

CollocatedDistribution

The *CollocatedDistribution* link type is defined similarly to the *DisjointDistribution* but with another purpose; it is to capture the collocation requirement in the relations between the entities of the fragments. *CollocatedDistribution* is also specialized from *LedLink* and has a *CollocationLevel* attribute. This link guarantees that the target entities are configured for groups of collocated source entities. For example, in the usecase of Section 5.2.2 if the SUs are required to be configured on the HEs of the same Rack, the *CollocationLevel* is set to the required level, i.e. Rack.

DisjointCollocatedDistribution

In Section 5.2.2 we mentioned that both the availability and affinity of the service providers may be required. However, these properties can be conflicting and should not be considered independently if both are required. To capture such relation another link type is added which inherits from both *CollocatedDistribution* and *DisjointDistribution* and thus has the properties of both. To make sure that the two concepts do not introduce any conflict, we make sure that each con-

cept is applied at a different level. This means that the level of providing availability through DisjointDistribution should be different from the affinity level provided by the CollocatedDistribution. The DisjointLevel and CorelationLevel attributes allow us to make such a distinction. However, the levels cannot be selected arbitrarily and need to respect a rule. To define this rule we again followed the OVF specification [19] which indicates that the collocation property should be provided in a higher level than the disjointness. This means that for example if the disjointness is provided at Host level, then the collocation level can be Chassis, Rack, Site or Geographic. This rule can be specified with an OCL constraint in the weaving metamodel as follows:

```
Context DisjointCollocatedDistribution
Inv C4: OrderedLevel.allInstances()->indexOf(self .DisjointLevel) < Or-
deredLevel.allInstances()->indexOf (self.CollocationLevel)
```

5.3.2 Creating the Links in the Weaving Model

Once the required link types have been defined in the weaving metamodel, they can be used in the weaving model for relating entities of the source metamodels to the entities of the target (system configuration) metamodel. The weaving model includes instances of links (instances of link types) associated with their respective link ends. Examples of these links are described in the following for EqualCorrespondence and DisjointDistribution link types.

In the case of the integration of AMF and PLM configurations, if we assume a fixed hardware platform and accordingly the PLM configuration is fixed and cannot be changed as part of the integration, then the AMF entities (i.e. the Nodes, NGs and SUs) should be configured according to the entities of the relevant PLM configuration (i.e. VMs and HEs) to satisfy the hardware disjointness constraint. Thus, in the DisjointDistribution link the PlmEEVM and PlmHE entities of

the PLM configuration metamodel are the Leader SourceEnd and the AmfNode, AmfNG, and AmfSU are the TargetEnds. The application of this link type with Host disjointness is as follows:

```
<<WLink>> DisjointDistribution HEDisjointSUs
  <DisjointLevel>
    OrderedLevel  Host
  <Source>
    <<Leader>>    PlmEEVM
    <<Leader>>    PlmHE
  <Target>
    <<TargetEnd>> AmfNode
    <<TargetEnd>> AmfNG
    <<TargetEnd>> AmfSU
```

In more details this link indicates that the AmfNode, AmfNGs and AmfSU entities in the target model are created or modified with respect to the PlmEEVM and PlmHEHost. These creations/modifications should happen in such a way that Host disjointness is provided for the AmfSUs. The CollocatedDistribution is used in a similar manner.

An instance of the EqualCorrespondence link type is used to map an entity of a source metamodel to a similar entity of the target metamodel. Some semi-automated methods such as the technique introduced in [36] can be applied to automate the creation of the mappings based on the similarity (such as string or type similarity) of the entities. Such automation can be applied only after all other types of links have been defined in the weaving model.

```
<<WLink>> EqualCorrespondence EqualVMs
  <Source>  <<Peer>>      PlmEEVM
  <Target>  <<TargetEnd>> SystemEEVM
```

In the next section we explain how the links are translated to transformation rules to create the target model with respect to the semantics of the relations (links).

5.3.3 Generating the System Configuration from the Weaving Model

To be able to generate a system configuration model it is necessary to translate the weaving model into an executable format. This translation takes place using an HOT, which itself is a transformation. The HOT translates the links of the weaving model into transformation rules. An excerpt of the HOT code is demonstrated in Appendix A. The output of the HOT is the Final Transformation as shown in Fig. 5.4.

For instance the translation of the DisjointDistribution link results in several transformation rules (expressed in ATL) in the Final Transformation. This translation is done with respect to the algorithm which we introduced in [28] to create hardware disjoint groups of VMs and the respective AmfNodeGroups to configure the AmfSUs on the AmfNodeGroups. The high level overview of these ATL rules and a brief description of each are provided hereafter:

```
rule NodeVM_AssociationCreation(id: Sequence(Integer))  
rule VMG_Creation()  
rule NG_Creation(vmg: Sequence(OclAny))  
rule SUNG_AssociationCreation(su:AMF!AmfSU,index:Integer)
```

NodeVM_AssociationCreation Rule

This rule creates a relation (an association) between each distinct pair of PlmEEVM and AmfNode e.g. associating an AmfNode to the most similar PlmEEVM regarding the capacity of the two entities. The association of a PlmEEVM to an AmfNode entity can be seen as an attribute of the AmfNode in the target model. This relation is the base connection between the entities of the two configuration models.

VMG_Creation Rule

This rule is used to generate the Host hardware disjoint VM Groups (VMGs) based on the input PLM configuration model. As explained in Chapter 2, through the PLM dependency object we know that each VM has a dependency on a number of VMMs, and each VMM is hosted on an HE, so we can identify the HEs where each VM can be hosted. We represent this information in a hardware dependency table. Fig. 5.6 shows an example of a hardware dependency table extracted from the PLM configuration of the same figure that we simplified by connecting the VMs directly to the VMMs they depend on. For instance for an arbitrary VM such as VM3 it is specified through dependency object that it can be hosted on VMM2 and VMM3 (or migrate between these VMMs). As VMM2 and VMM3 are installed on HE2 and HE3 respectively, we can say that VM3 eventually will be on HE2 or HE3.

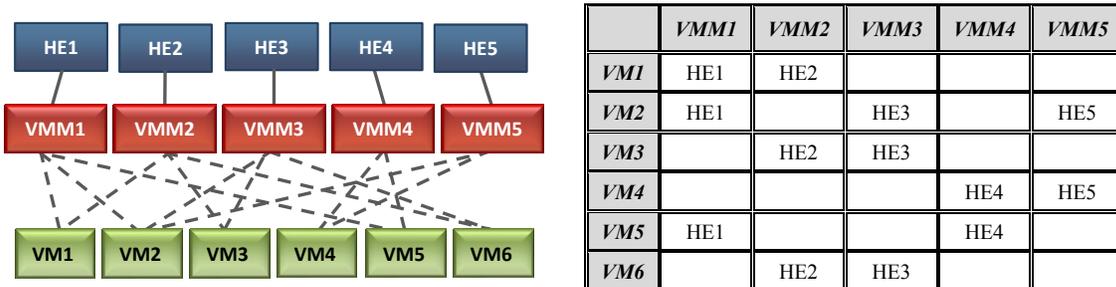


Figure 5.6. A partial PLM configuration and corresponding hardware dependency table

We defined an algorithm (Algorithm 5.1) to determine the HE-disjoint sets of VMs and their corresponding hardware-disjoint VMG set, for a given PLM configuration taking the hardware dependency table of the configuration as the input. The algorithm starts with initializing three sets: Set A containing all VMs and two empty sets named *Leftovers* and *VMGset*. The set of HEs that each VM i can be mapped to is defined as HE-VM i on Line 6. The rest of the algorithm is divided

in two parts. The first part (Line 8 to 26) determines the set of VMGs and their associated HEs (HE-VMG_i) by repeating the process of selecting the VM associated with the lowest number of HEs and the VMs that are associated with exactly the same set of HEs, removing them from A

Algorithm 5.1: Defining HE-disjoint groups of VMs

Input: HW_DependencyTable from the PLM configuration

Output: Set of HE-disjoint VMGs in VMGset and a set of unused VMs in leftovers

```

1: A := set of all VMs in PLM configuration
2: Leftovers := {}
3: VMGset := {}
4: // Identify the HEs related to each VMi based on HW_DependencyTable of the PLM configuration
5: for each VMi in A do
6:   HE-VMi := {HEs related to VMi in HW_DependencyTable}
7: end for
8: // Select among remaining VMs the VM associated with the lowest number of HEs in HW_DependencyTable
  and remove from A the VMs that are not HE-disjoint with it
9: // n is the counter of the VMGs
10: n := 0
11: repeat
12: select VMi from A such that |HE-VMi| ≤ |HE-VMj| for any VMj in A
13:   n := n+1
14:   VMGn := {VMi}
15:   HE-VMGn := HE-VMi
16:   A := A - {VMi}
17:   for each VMj in A
18:     if HE-VMj = HE-VMGn then
19:       VMGn = VMGn ∪ {VMj}
20:       A = A - {VMj}
21:     else if HE-VMj ∩ HE-VMGn ≠ {} then
22:       Leftovers := Leftovers ∪ {VMj}
23:       A = A - {VMj}
24:     end if
25: VMGset := VMGset ∪ {VMGn}
26: until A = {}
27: // Adding the leftover VMs to formed VMGs iff they intersect with only one VMG with respect to their HEs
28: for each VMi in Leftovers do
29: // k is the counter of VMGs with which VMi has common HEs
30: k := 0
31: for each VMGj in VMGset do
32:   if HE-VMi ∩ HE-VMGj ≠ {} then
33:     k++
34:     temp := j
35:   end if
36: end for
37: if k = 1 then
38:   VMGtemp = VMGtemp ∪ {VMi}
39:   HE-VMGtemp := HE-VMGtemp ∪ HE-VMi
40:   Leftovers := Leftovers - {VMi}
41: end if
42: end for

```

and moving to Leftovers any VM, which has any common HE with them (i.e. not HE-disjoint). The second part (Line 27 to 42) goes through the leftover VMs in the Leftovers set one by one and adds any of them that is HE-disjoint with all defined VMGs except one. The VM is added to the VMG with which it overlaps in terms of HEs. The other VMs remain in the leftovers.

Applying Algorithm 5.1 to the example of Fig. 5.6 will result in creating two hardware-disjoint VMGs, VMG1 and VMG2, determined following the steps of the algorithm as follows:

- $A = \{VM1, VM2, VM3, VM4, VM5, VM6\}$, Leftovers = {}, VMGset = {}

First VMG is created with VM1 that maps to two HEs:

$VMG1 = \{VM1\}$ with the set of HE-VMG1 = HE-VM1 = {HE1, HE2}

$A = \{VM2, VM3, VM4, VM5, VM6\}$

- VM2, VM3, VM5, VM6 are removed from A and put into Leftovers because of having HEs in common with VMG1

$VMG1 = \{VM1\}$ with HE-VMG1 = {HE1, HE2}, VMGset = {VMG1}

$A = \{VM4\}$

Leftovers= {VM2, VM3, VM5, VM6}

- The algorithm repeats previous steps and creates VMG2

$VMG2 = \{VM4\}$ with HE-VMG2 = {HE4, HE5}

VMGset = {VMG1, VMG2}

$A = \{\}$

Leftovers= {VM2, VM3, VM5, VM6}

- While handling the leftovers, VM3, VM6 can be added to VMG1 because they have HE2 in common with VMG1 and by adding them the HE-disjoint rule for the VMGs remains satisfied. This extends the set of HEs of VMG1 with HE3. VM2 cannot be added to neither VMG1 nor VMG2 because in either case the disjoint rule is violated. Similarly, VM5 also cannot be added to neither VMG1 nor VMG2. The algorithm terminates with:

$VMG1 = \{VM1, VM3, VM6\}$ with HE-VMG1 = {HE1, HE2, HE3}

$VMG2 = \{VM4\}$ with HE-VMG2 = {HE4, HE5}

VMGset = {VMG1, VMG2}

$A = \{\}$

Leftovers= {VM2, VM5}

The VMGs resulting from this algorithm (VMG1, VMG2) are hardware-disjoint and can be used in the configuration whenever hardware redundancy is required. The VMs remaining in the Left-overs set at the end of the algorithm cannot be used for the purpose of hardware redundancy.

The isolation of this calculation in a rule makes its modification or replacement by another algorithm easy and avoids touching the rest of the transformation model. This rule creates VMGs (each of which is a *sequence* of VMs) collected in a VMGSet (which is a *sequence* of *sequences* in ATL). The VMGSet and its VMGs do not appear in the target configuration but are used to create an NGSet and its NGs.

NG_Creation Rule

This rule creates the AmfNGs in the target model based on the previously created VMGs of the VMGSet and adds the relevant AmfNode entities to the created the AmfNG. In our translation we assume that we do not have the AmfNG entities in the AMF model, so we create them in the target model.

SUNG_AssociationCreation Rule

Finally this rule is used to establish the relation (association) between the AmfSUs of each AmfSG and an AmfNG entity which was created by the previous rule. In this work we do not consider any criteria for matching the AmfSUs to AmfNGs. This rule can be extended in the future by adding different heuristics for selecting the most appropriate AmfNG for each AmfSU based on some criteria (such as the number of AmfSUs in the AmfSG, etc.)

The Final Transformation generated from the HOT takes the configuration fragment models as input and generates a system configuration model as output. The generated configuration will

have all the entities of both input models and also the new entities and relations among the entities of the fragments capturing the targeted properties entailed by the weaving links.

5.4 Constraint Generation from the Integration

The transformation rules in the Final Transformation are generated by considering the special relations among the entities of the configuration fragments. These relations guarantee the targeted properties of the system configuration, i.e. the consistency of the system configuration with respect to the targeted properties such as availability and affinity.

Although this integration semantics is taken care of in the process of generating the system configuration model, it is not reflected in the system configuration profile. This integration semantics needs to be defined as integration constraints in the system configuration profile in order to guard the consistency of system configuration models against unsafe runtime modifications. The integration constraints (i.e. originating from the transformation rules and describing the semantics of the relation between the fragments) in addition to the union of the constraints of the fragments form the system configuration constraints. The transformation rules of the Final Transformation can be reused to generate automatically the integration constraints. The configuration designer does not have to define them manually as they are already embedded in the transformation rules. In this section we describe how the integration constraints can be extracted. We describe our approach for the generation of OCL constraints from the ATL transformation rules. Fig. 5.7 shows the constraint generation from the Final Transformation and the completion of the system configuration profile.

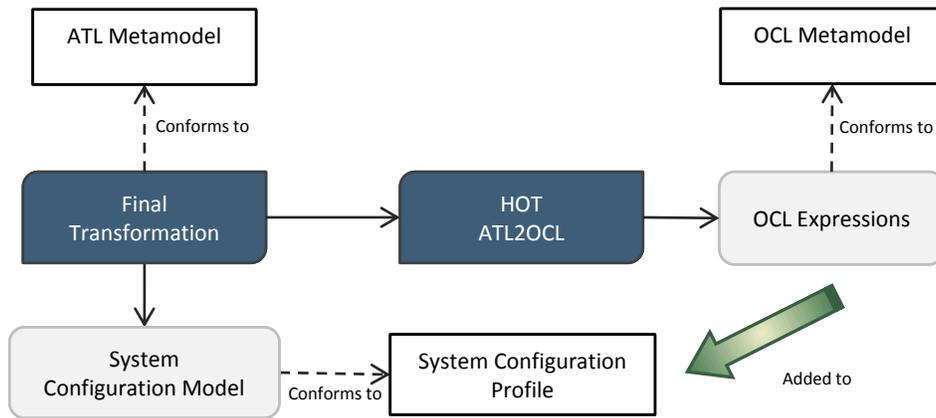


Figure 5.7. Generation of OCL constraints from the ATL transformation

An ATL transformation model consists of rules and helpers. There are three types of transformation rules in ATL: matched rules, lazy rules and called rules. The most commonly used rule type is the matched rule, which generates the target entities from the source entities defined by the source pattern of the rule. A matched rule is executed for all the occurrences of its source pattern. In contrast to the matched rule, a lazy rule is executed only when it is invoked. Finally, called rules are used to create target entities from imperative code. To be executed the called rules need to be invoked from an imperative code, which can be the action block of a matched rule, or from within another called rule. Helpers in the context of ATL are similar to methods. The helpers can be called from different points of an ATL program.

5.4.1 Entity Derivation Tree

Each target entity or its attribute is created by some transformation rules and helpers. If the entity is created in a rule and its attributes are created in some other rules, we consider the attribute creation as an entity modification. By following the transformation rules and helper invocations for the creation/modification of each target entity we can specify the process of its crea-

tion/modification as a *derivation tree*. At the root of the tree there is a target entity and at each level of the tree the nodes are the entities which are used to create their parent node and the edges are the operations that are applied on the nodes to create the parent node. An operation can be a rule/helper invocation, a filter or guard expression, or a piece of imperative code in the rules. At the last level of the tree are the leaves (entities) which already exist in the target model or the source models. Although the derivation tree can be created for each target entity, we are interested only in the target entities that are created/modified using some operations.

Traversing this tree from the root to the leaves helps us identifying the entities and operations that are used to create/modify a target entity. On the other hand going over the tree from the leaves to the root describes how a target entity is created/modified, from the operations the effect of which need to be captured as constraints between the entities of the system configuration.

Fig. 5.8 shows a very simple example of a derivation tree for creating the SystemEEVM from the PlmEEVM. A simple transformation rule called VM_Transformation is used to copy the PlmEEVM entities from the PLM configuration fragments to the system configuration and create the SystemEEVM entities. This rule is the translation of the *EqualVMs* link in Section 5.3.2 (i.e. an EqualCorrespondence weaving link). Let assume that we want only the VMs with Memory of 512MB or higher to be used for the target entity creation. Therefore the source pattern used in the VM_Transformation rule uses a filter on the entities of the source. Starting from the target entity and following the transformation rule creating it, we reach the source entity to which the filter operation was applied. This is shown as *Traversing direction* in Fig. 5.8. The target entity node (SystemEEVM) is created from the filtered source node (PlmEEVM) which is shown as *Entity creation direction* in the figure. The filter is an example of an operation that can be ap-

plied to source entities; it is shown on the edge connecting the nodes. Other operations can be helpers, called rules, or lazy rules.

```
rule VM_Transformation {
  from source:PLM!PlmEEVM(source.Memory>512)
  to target: System! SystemEEVM(
    Memory<- source.Memory ) }
```

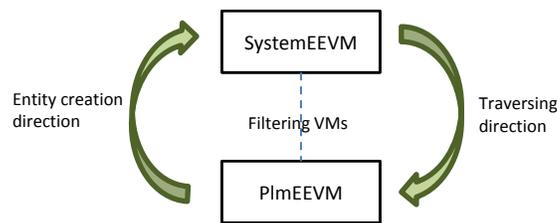


Figure 5.8. An example of derivation tree for SystemEEVM

More examples of derivation trees are shown in Fig. 5.9, which is based on the DisjointDistribution link and its respective transformation rules. Fig. 5.9 (a) shows the derivation tree of the modification of the AmfNode entity to map it to a VM in the PLM configuration fragment (the association between the AmfNode and PlmEEVM is considered as an attribute of the AmfNode). This tree has only one level and the operation on the edge between the root (AmfNode) and the leaf (PlmEEVM) is the rule NodeVM_AssociationCreation rule, i.e. a called rule which selects a distinct PlmEEVM for the AmfNode possibly based on some other criteria such as the capacity of the VM.

Fig. 5.9 (b) is the derivation tree for the creation of the AmfNG. This tree has two levels: level 1 includes the AmfNode and the VMG tree nodes on which the NG_creation operation (i.e. a called rule for creating NGs from the VMGs) was applied at this level. As the AmfNode exists in

the system model, it is a leaf node of the tree. On the other hand no VMG entity exists in the source or the target models. It is an entity which is only created and used in the transformation rules as an auxiliary entity. The VMG entity represented by the VMG node of the tree is created from the PlmHe, PlmDependency and PlmEEVM entities of the system

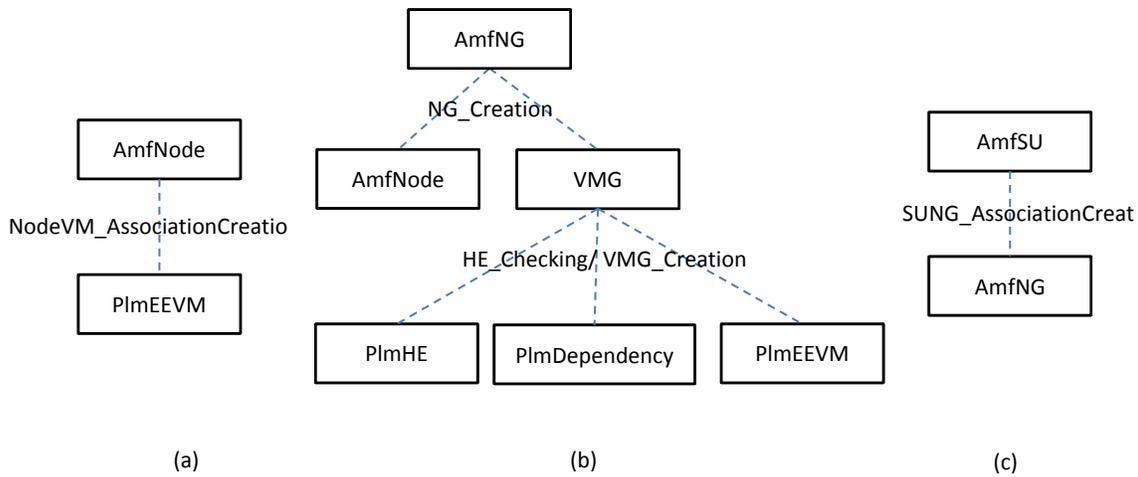


Figure 5.9. The derivation trees for the AmfNode, AmfNG, and AmfSU

model. To preserve any constraint implied by these operations in relation to the AmfNG, we need to include in our derivation tree these as well. Thus these entities are shown as the tree nodes in level 2 of the derivation tree. The VMG_Creation (i.e. a called rule) and the HE_Checking (i.e. a helper) are the operations applied on the nodes of level 2 to create their parent which is the VMG.

Fig. 5.9 (c) shows the derivation tree for the modification of the AmfSU entity (the association between the AmfSU and AmfNG is considered as an attribute of the AmfSU). This tree has only one level and the operation on the edge between the root (AmfSU) and the leaf (AmfNG) is the rule SUNG_AssociationCreation rule (i.e. a called rule which selects a distinct NG for the AmfSUs).

5.4.2 Translation of the ATL Operations to the OCL Expressions

Once a derivation tree is created, from the operations applied on the nodes of each level we need to derive an appropriate OCL expression. The context of a generated OCL expression at each level is the parent entity if this entity exists in the target model. E.g. for tree (a) of Fig. 5.9, AmfNode is the parent node and it is a target entity, which exists in the target model, therefore the context of the generated OCL expression from this tree is AmfNode.

However at any level of the tree if a parent does not exist in the target model (i.e. the parent is an auxiliary entity which is only used in the transformation) then the parent entity cannot be the context of the OCL expression generated for its subtree. (Note that this cannot happen to the root of the tree, which is a target entity and therefore it is always in the target model.) In such cases the context is the same as for the level above, e.g. the parent of this parent. An example of this case is the VMG entity in tree (b) of Fig. 5.9, which is created from the PlmHE, PlmDependency and PlmEEVM entities, but the VMG entity does not exist in the target model. So the context of the OCL expression created from the VMG_Creation and HE_Checking cannot be the VMG and is defined as for the level above, i.e. the parent of the VMG entity in the tree, which is the AmfNG.

To derive the OCL expression, we have categorized the ATL operations and the OCL expression respectively into types and define the types mapping. Table 5.1 summarizes the ATL operation types we identified for common ATL operations and the mappings of these ATL operation types to OCL expression types. These mappings are then defined as an HOT transformation (i.e. the ATL2OCL transformation of Fig. 5.7). Thus, we reuse the mappings for similar operations.

Table 5.1. The mapping of ATL operations to OCL expressions

ATL Operation Type	OCL Expression Type
Type operations in the filters (operations on primitive or collection types (e.g. select, iterate, so on)	Type operations as the invariant of constraint
Matched rules with iterative binding of entities' attributes (e.g. for loop)	Defined by <i>allInstances</i> or <i>forAll</i> expressions
Variables in the <i>Using</i> section	Defined by <i>let</i> expression
Helpers, Lazy rules, Called rules	Defined as the Body of Query operations

The OCL expressions resulting from applying this mapping to the derivation trees of Fig.5.9 are shown in Table 5.2.

Table 5.2. The OCL expressions resulting from the derivation trees of Fig. 5.9

Tree	ATL Operation	Operation Description	OCL Expression
a	NodeVM_Association Creation	-It maps each VM to a distinct Node -It requires PlmEEVM	Context AmfNode Inv: self.allInstances->forAll(N1,N2 N1 <> N2 implies N1.vm <>N2.vm)
b	Applied on Level2: VMG_Creation HE_Checking	-They are called to create the VMG -They require the PlmHE, the PlmDependency and the PlmEEVM	Context AmfNG::Disjointness(Ng1,Ng2):Boolean Body : If (Ng1.node -> iterate (N; VMM1: PlmEEVMM VMM1->including (N.vm.dependency.supplier))-> iterate (VMM; HE1: PlmHE HE1->including (VMM.he)) ->intersection(Ng2.node -> iterate (N; VMM2: PlmEEVMM VMM2-> including (N.vm.dependency.supplier))->iterate

			(VMM; HE2 : PlmHE HE2->including (VMM.he)) -> isEmpty()) then return True else return False endIf
	Applied on Level1: NG_Creation	-It is called to create the NGs -It requires the VMG and the AmfNode	Context AmfNG Inv: self.allInstances->forAll(Ng1,Ng2 Disjointness(Ng1,Ng2)=True)
c	SUNG_Associat ionCreation	-It modifies the SUs. -It iterates over the SUs of each SG to associate each SU with a distinct NG	Context AmfSU Inv: self.allInstances->forAll(Su1,Su2 Su1.sg=Su2.sg implies Su1.ng <>Su2.ng)

Note that for tree (b) Level 2 defines the Disjointness method in the context of the AmfNG, which is referenced at Level 1.

5.4.3 Role Definition for the Constrained Entities

In addition to the generation of the OCL expressions we can also capture the role of the constrained entities in the constraints. We use the leadership concept explained in Chapter 4 to capture the Leader/Follower/Peer role of constrained entities in the integration constraints and obtain these roles from the weaving process: The entities specified as the Leader SourceEnd of the LedLink take the Leader role and entities specified as the TargetEnd of the LedLink have the Follower role in the constraint generated from the LedLink and its transformation rules.

In the other link types, the SourceEnd is Peer and therefore, both the Peer Source entities and the Target entities of the link will have the Peer role in the LeadershipInfo of the generated constraint as they have equal influence over each other in the target model.

In the DisjointDistribution weaving link between the AMF and PLM configurations, the PlmEEVM and the PlmHE (the Leader Source entities) have an influence on the AmfNode, AmfNG and AmfSU (the Target entities). Accordingly in the generated constraint the PlmEEVM and the PlmHE entities have the Leader role and can affect the AmfNodes, AmfNGs and the AmfSUs which have the Follower role in the LeadershipInfo of the corresponding constraints.

5.5 Implementation and Discussion

We have implemented our approach using the Atlas Model Weaver (AMW) [8, 63, 64] and the Eclipse Modeling Framework (EMF) [10]. We extended the generic weaving metamodel of the AMW defined with the extensions discussed in Section 5.3.1. We used ATL [9] as our model transformation language for the implementation of the transformations. The algorithm defined in [28] is used as an instance for the implementation of the DisjointDistribution link, however it can be replaced with alternative algorithms.

Although we explained our approach in the context of the SA Forum middleware configuration fragments, we believe it is applicable to other domains where the integration of configuration fragments is required. Our integration approach is based on the model weaving technique and focuses on the semantics of the relations in the weaving. Examples of such semantics are the HW-disjointness property (to ensure hardware redundancy for redundant software entities to increase the availability of the system), the HW-collocation property (to decrease the communication latency due to state synchronization) or the combination of the two.

Defining special link types in the weaving metamodel allows for the development of more abstract mappings. Abstracting concepts is an intrinsic feature of metamodeling, which is discussed

widely in the literature. This advantage becomes bolder in the case of configuration integration from two perspectives: First it increases the reusability of link types since the defined link types can be used in future mappings when other configuration fragments need to be added with the same relations, they can use same link types for their mappings (e.g. using the “EqualCorrespondence” link type). The second advantage of the abstract definition of link types is that it allows for the selection of the desirable interpretation and implementation for the mapping. This means that the declarative definition of the link types can be translated according to the features of the system. Let us consider the “DisjointDistribution” link type. We interpreted this link with the assumption that we have a predefined PLM configuration with specific entities that are fixed and cannot be modified; on the other hand we forced the AMF to use the newly defined VM groups by changing the AMF configuration. While another interpretation of the “DisjointDistribution” may consider the AMF configuration as fixed and unchangeable model and use other heuristics to change the PLM configuration in a way to still provide hardware redundancy for redundant software entities.

The fact that model transformation is used to translate the weaving model into an executable format enhances the benefits of the weaving model. The reason is that the links of the weaving model help us to capture the transformation patterns and reuse them rather than defining all the rules manually. Another reason for selecting the model weaving technique over the direct model transformation is the extensibility of the weaving for integrating additional models with less manual effort. With model weaving we can simply add more models as input into the weaving process and the respective transformation rules will be generated automatically, while adding

more models directly into a transformation requires considerable time and effort to develop the new transformations rules.

The relations between the entities of the fragments need to be preserved at runtime, i.e. the targeted system properties also called system configuration consistency need to be preserved at runtime. However, because this integration semantics is not defined initially in the system configuration profile, the configuration designer has to define them manually. Reusing the transformation rules for the generation of the integration constraints is another advantage of our work. This automated constraint generation reduces the risk of miss-interpretation by different configuration designers of the integration relations.

The fact that we use the Final Transformation instead of the WModel2Final HOT for constraint generation implies that this technique can be used for other ATL transformations as well and it is not restricted only to weaving and integration transformations.

5.6 Summary

As systems can be developed by the integration of different aspects, services, or components developed separately, the system configuration can be obtained from the integration of the configurations of these different aspects developed separately as well. Although developed independently these configuration fragments are interrelated as they represent and potentially act on the same system entities. Moreover, the integration may target specific properties, such as availability or affinity, etc. Therefore, the configuration fragments need to be integrated carefully to form together a consistent system configuration with respect to fragment internal properties and targeted system configuration properties.

We tackled this problem with a model-based approach using model weaving. We used configuration samples from the SA Forum middleware for illustration purposes. Our approach to integrate configuration fragments takes into account the properties of the target system configuration. We used the weaving model to capture the mapping between the entities of the different configuration profiles. Model weaving has been widely used for model integration, model transformation, model merging, etc. [6, 35, 36, 37], however so far it has focused primarily on the static mapping of entities without considering the semantics of these relations. We introduced new link types to capture the special relations, i.e. integration semantics, between the entities of these profiles in a weaving model, i.e. they are added to the weaving metamodel. Using a set of ATL transformations we generate a consistent system configuration from the weaving. Although this integration semantics is taken care of in the process of generating the system configuration model, it is not reflected in the system configuration profile. This integration semantics needs to be defined as integration constraints in the system configuration profile in order to guard the consistency of system configuration models against unsafe runtime modifications. This is achieved automatically in our approach.

Our approach for integrating configuration fragments allows for the reuse and the extension of the system configuration generation process as the link types were defined once and reused for the mapping of different entities of the configuration fragments. More profiles can also be added to the process using the same or new link types.

Chapter 6

Partial Validation of Configurations at Runtime

In this chapter we explain our partial validation technique for checking the system configuration consistency at runtime. Moreover, we discuss how the output of the partial validation is envisioned to serve as input for an adjustment function which is also part of our configuration management framework.

The contents of this chapter have been published in [18].

6.1 Introduction

At runtime a system configuration may be modified for instance in response to changes in the system environment, for security/performance or fine-tuning purposes. These changes may jeopardize the configuration consistency as some constraints may be violated. Thus, any reconfiguration request needs to be checked against these constraints. A violation of a constraint means that the changes are not safe and/or incomplete. Runtime validation is a prerequisite for dynamic re-

configuration as detection and correction of potential inconsistencies are required. The capabilities of dynamic (or runtime) reconfiguration and runtime validation are also needed in high availability (HA) systems as they cannot be shut down or restarted for reconfiguration.

A system configuration may consist of hundreds of entities, with complex relations and constraints. Runtime reconfigurations (changes to a single system entity or a bundle of changes to a number of entities) often target only parts of the configuration. In such cases an exhaustive validation which checks all the consistency rules is not always required and can be substituted by a partial validation, in which the number of consistency rules (constraints) to be checked are reduced and this results in reduction of the validation time and overhead. This means that in partial validation only the constraints that are impacted by the changes are selected and will be checked because the other ones remain valid and do not need to be rechecked. In this chapter we discuss a partial validation technique for the configuration validation.

6.2 Partial Validation Technique

To validate a configuration model, its conformance to the configuration profile is checked. The profile defines the stereotypes, their relations (the structure of the model) along with a set of constraints over these stereotypes and relations to assure well-formedness. When a request for changing some entities of the configuration model is received, the modified model needs to be checked to make sure that it conforms to its profile. This full validation can be time and resource consuming. Such overhead is not desirable in live systems, especially in real-time and highly available systems. A solution to reduce the overhead and improve performance is to reduce the number of constraints to be checked, i.e. check only what needs to be checked again. We refer to this as partial validation.

In our approach we minimize the number of constraints to be checked based on the requested changes. This of course also leads to the reduction of the number of configuration entities to be checked. We check only the entities whose stereotypes are involved in the selected constraints. This new set of configuration entities includes at least the changed entities and the ones related to them through their constraints. We also provide a semi-formal proof to show that the results of our partial validation approach is equivalent to the results of the full validation where all the constraints are checked for every change request.

Fig. 6.1 represents an example in which changes in the model affects only some of the constraints. In this example a model and its profile is shown. The model entities (e.g. A1, B2, D3) conform to their respective stereotypes of the profile e.g. A1, A2 entities of the model conform to stereotype A and B2, B3 entities conform to stereotype B and so on. The constraints of the profile are shown as blue ovals (i.e. C1, C2, etc.).

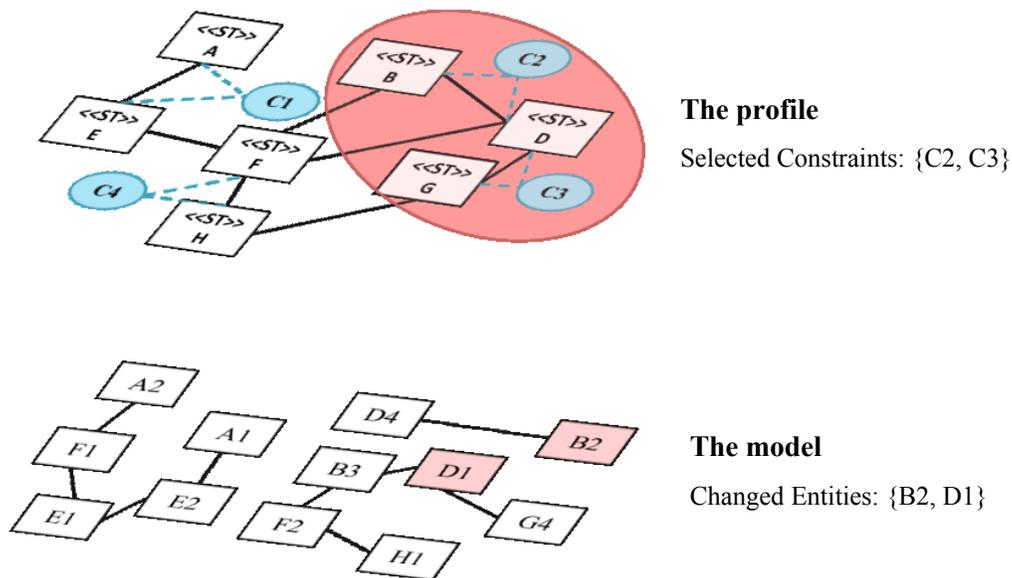


Figure 6.1. Model changes and affected constraints in the profile

Assuming that the change set includes model entities B2 and D1. To validate the model instead of checking all the constraints of the profile, it is enough to select only the ones that are affected by these changes which are C2 and C3 constraints.

To reduce the validation overhead the time to select this new set should be negligible compared to the time saving we achieve by the partial validation, i.e. validating the new set. Note that in cases where the modification request includes entities of many different stereotypes, the number of constraints that need to be selected is considerable and the selection may not be worthwhile anymore. The case is similar when although only a few constraints are selected but they apply and should be checked for a large number of configuration entities.

6.2.1 Filtering the Constraints

To identify the reduced set of constraints we filter the constraints based on the modification request. We assume that a request may consist of many changes each of which applies to one or more entities of the configuration model – we call them the change set.

We represent the configuration entities of the change set as a model, which conforms to a change profile. Fig. 6.2 shows the change profile and an example change set. The change profile has a stereotype called CEntity which extends the NamedElement metaclass of UML. The CEntity stereotype represents the configuration model entities to be changed – referred to as changed entity. In the Petstore, for example, we may need to reconfigure the Web Server, i.e. have it as the changed entity. The operation requested on the model entities is represented by Operation stereotype in the change profile. It is specialized as the Add, Update, and Delete stereotypes. Regardless of the requested operation, the constraints in which the entity is involved should be checked.

For the time being we are not concerned about the operation type and will not elaborate it further. Such a change model is one of the inputs to the constraint filtering process.

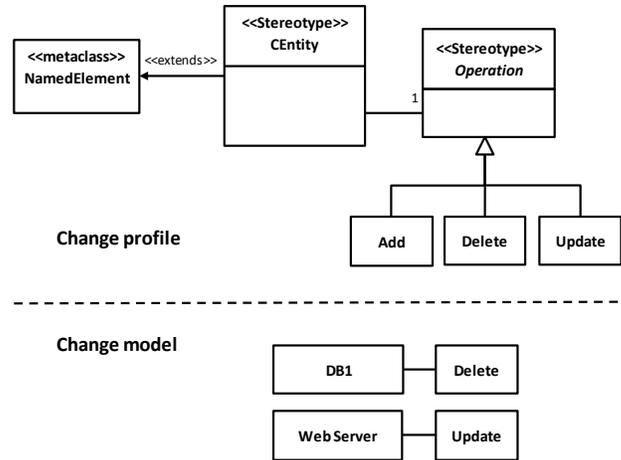


Figure 6.2. Change profile and a simple change model

Having the change model and the change profile as well as the configuration profile, the stereotypes applied on each changed entity can be identified. These are the stereotypes of the configuration profile and also the CEntity stereotype. For the validation the stereotypes of the configuration profile are considered. The constraints of the configuration profile are defined over these stereotypes and we also captured their roles through the leadership concept. By looking up the stereotype of each changed entity we can select from the constraints of the configuration profile that have the same stereotype as the stereotype applied to the entity of the change model. The role/leadership information determines the relevance of the constraint. Algorithm 6.1 describes the filtering process and the categorization of the filtered constraints.

The algorithm starts with an empty set of stereotypes, set A , and three empty models, LConstraint, FConstraint, PConstraint. In set A we collect the stereotypes of all changed entities (lines 6 to 8). The constraints of the configuration profile are captured in the ConstraintModel which

conforms to the constraint profile that we have previously defined. For each constraint in the ConstraintModel we consider its LeadershipInfo and compare the Leader/Follower/Peer stereotypes with the stereotypes of set A. If a common stereotype is found, then the constraint is added to one of the sets of LConstraint, FConstraint, PConstraint while making sure that each constraint will be added only to one of these output constraint sets (line 10 to 19). Thus we also categorize them during the process. The filtering process of Algorithm 6.1 is implemented using the transformation model shown in Fig. 6.3.

Algorithm 6.1 Filtering and Categorizing the Constraints

Input: ConfigurationProfile, ChangeProfile, ConstraintProfile, ConstraintModel, ChangeModel

Output: LConstraintModel, FConstraintModel, PConstraintModel

```

1: A := {}
2: LConstraintModel := {}
3: FConstraintModel := {}
4: PConstraintModel := {}
5: // Find all the stereotypes applied to the entities of the ChangeModel
6: for each ENTITYj in ChangeModel do
7:   A := A  $\cup$  {ENTITYj.getAppliedStereotypes()}
8: end for
9: // Filtering and categorizing constraints of the ConstraintModel
10: for each CONSTRAINTi in ConstraintModel do
11:   K := CONSTRAINTi->LeadershipInfo
12:   if {K.Leader}  $\cap$  A  $\neq$  {} then
13:     LConstraintModel := LConstraintModel  $\cup$  {CONSTRAINTi}
14:   else if {K.Follower}  $\cap$  A  $\neq$  {} then
15:     FConstraintModel := FConstraintModel  $\cup$  {CONSTRAINTi}
16:   else if {K.Peer}  $\cap$  A  $\neq$  {} then
17:     PConstraintModel := PConstraintModel  $\cup$  {CONSTRAINTi}
18:   end if
19: end for

```

6.2.2 Categorizing the Constraints

We categorize each selected constraint based on the roles of the changed entities in the constraints. We distinguish three constraint categories:

- FConstraint
- LConstraint
- PConstraint

If the stereotype applied on a changed entity has a leader role in the constraint, we add the constraint to the LConstraint set. Similarly, if the stereotype of a changed entity plays the follower role, the constraint is added to the FConstraint set. The PConstraint category is for the constraints whose entities have a peer role in the constraint and also appear in the change model. This categorization is also shown in Fig. 6.3.

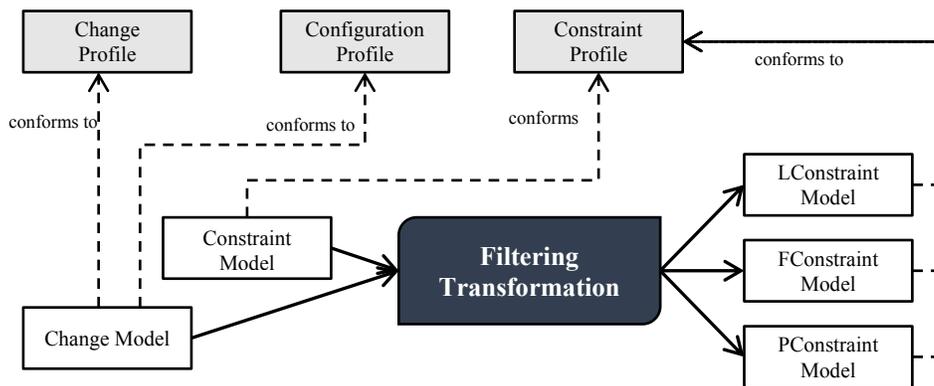


Figure 6.3. Filtering and categorizing the constraints based on the change model

It is possible that a constraint can be categorized in more than one category. For example, when both the leader and follower entities of a constraint are changed within the same change set the constraint can be categorized as FConstraint and as LConstraint and added to both. Considering the Petstore again a request may for example change DB1, an instance of the DB Tier (leader) and Host1, an instance of the Host (follower) in the same change set. In such a case, we have to make a choice. We add such constraints to the least restrictive LConstraint set to allow for potential adjustments. Since at least one leader entity is involved in case of constraint violation its fol-

lower(s) (except the ones that are in the change request) can be adjusted to satisfy the constraint. Similarly, the PConstraint category is preferred over the FConstraint category.

6.2.3 Validation of the Constraints

Once the constraints are filtered and categorized, the validation starts with the least flexible constraints and moves towards those allowing for more flexibility. The constraints in the FConstraint category are the least flexible ones because if they are violated, no adjustment can be made within the context of the change set to resolve the inconsistency as the follower entities cannot affect the leader entities. Thus in case of detecting a violation of the FConstraints, the requested change is rejected and the validation process stops. Next the LConstraint and the PConstraint sets are checked. If the validation fails in these cases, we consider this as a potential violation of configuration consistency because we may be able to resolve it with additional changes as leader entities can affect their followers and similarly peer entities can affect their peers. The adjustment module will try to resolve the inconsistency through additional modifications. The detailed adjustment process is explained in the next chapter.

6.3 A Semi-formal Proof for the Partial Validation Technique

In this section we provide a semi-formal proof of our validation approach which shows that the reduced set of constraints contain all the required ones to guarantee the validity of the whole configuration model.

Note that we do not distinguish the leader/follower/peer roles of the constrained entities for the proof. Because to prove the correctness of the validation we are only concerned about the sufficiency of the selected (filtered) constraints rather than their categorization.

6.3.1 Definitions

Profile: A Profile (P) is defined as a set of Stereotypes (ST_P), the set of Relations between them (R_P) and their set of Constraints ($Cons_P$).

$$P (ST_P, R_P, Cons_P)$$

For referring to the sets of stereotypes, relations and constraints of a given Profile we use the Profile's name as the index of the set e.g. ST_{P1} is the set of stereotypes of Profile P1.

Each relation Rl in the set R_P consists of a source stereotype (Rl.SrcST) and a destination stereotype (Rl.DstST). They specify the two ends of the relation Rl. A tuple of Lowerbound and Upperbound also specifies the minimum and maximum number of instances of the DstST in relation with a SrcST.

$$Rl (Rl.SrcST, Rl.DstST, (L, U))$$

For simplicity the relations in this definition are considered to be associations. Other types of relations (generalization, dependency, etc.) can be added with appropriate modifications of the definition.

Constraint: Each constraint ConsX in the set of $Cons_{P1}$ consists of an *Invariant* (a Boolean expression) and a set of stereotypes based on which the invariant is defined i.e. the constrained stereotypes. The set of constrained stereotypes is shown as ST_{ConsX} .

$$ConsX (Invariant, ST_{ConsX})$$

Model: A Model is defined as a set of entities (en_M) and a set of relations (r_M).

$$M (en_M, r_M)$$

Each relation rl ($rl \in r_M$) has a source entity represented as $rl.SrcEn$ and a destination entity represented as $rl.DstEn$.

$$rl(rl.SrcEn, rl.DstEn)$$

In order to be valid, a model should conform to its profile. This means that each entity of the model should respect the stereotype(s) of the profile that is applied to and also all the constraints of the profile should be valid in the model. These functions are defined as follows:

Let us assume profile P is applied on model M . The function AST^1 for the input of a model entity (that belongs to the entity set of a model M) returns as the output the stereotype (that belongs to the stereotype set of the profile P), which is applied on the entity.

$$s = AST(e), x \in en_M, s \in ST_P$$

AtomicValid is defined over a constraint (x) and a subset of entities and relations (K) that belong to a model (M). The result of this function is a Boolean value which shows whether x is satisfied with the values of the entities and relations in K or not. Thus K is a subset of M and contains the entities and relations that are related to constraint x . The entities in K are defined as those entities of the model on which the stereotypes of the constraint x are applied. If the stereotypes of constraint x are applied on the member ends of a relation, the relation is included in K . (en_M and r_M represent the set of entities and relations in model M and ST_x represents the set of stereotypes of the constraint x).

$$AtomicValid(K, x)$$

$$K(en_K, r_K) = \{(e, r) \mid e \in en_M, AST(e) \in ST_x, r \in r_M, (AST(r.SrcEn) \in ST_x \text{ AND } AST(r.DstEn) \in ST_x)\}$$

¹ Applied StereoType

If the result of the *AtomicValid* is true, it means that the constraint *x* is satisfied over a subset of entities and relations of the model *M* (entities on which the stereotypes of the constraint *x* are applied). So we can conclude that the constraint *x* is satisfied in model *M* or in other words, the validity of the constraint *x* over model *M* is true. Another function *Valid* is used to represent this statement.

$$\text{AtomicValid}(K,x) \leftrightarrow \text{Valid}(M,x)$$

Conformance of a model $M(en_M, r_M)$ to a profile $P(ST_P, R_P, Cons_P)$ is defined through the *Conform* function which returns true if all the constraints of the *P* are valid over model *M* and also all the entities and relations of the model respect the stereotypes and relations of the *P*. $Cons_P$ and ST_P are the sets of constraints and stereotypes of the profile *P* respectively. en_M and r_M are the set of entities and relations of model *M*. The *Respect* function is used to check if the entities and relations of the model respect the stereotypes and relations of the profile.

$$\begin{aligned} \text{Conform}(M, P) \leftrightarrow & (\forall x \in Cons_P, \text{Valid}(M, x)) \text{ AND } (\forall e \in en_M, \text{AST}(e) \in ST_P, \text{Respect}(e, \\ & \text{AST}(e)) \text{ AND } (\forall t \in r_M, \exists z \in R_P, \text{AST}(t.\text{SrcEn}) = z.\text{SrcST}, \text{AST}(t.\text{DstEn}) = z.\text{DstST}, \text{Re-} \\ & \text{spect}(t, z)) \end{aligned}$$

6.3.2 Modifying the Model

We assume that we have an initial model *M1* which is valid according to the profile *Pr* i.e. $\text{Conform}(M1, Pr)$. The *Change* function takes the *changeSet* model and *M1* as input and results in a new model *M2* with the modified entities and relations, i.e. applies the *changeSet* to *M1*.

$$\text{changeSet}(en_{\text{changeSet}}, r_{\text{changeSet}})$$

$$M2 = \text{Change}(M1, \text{changeSet})$$

To verify whether the changed model (M2) is also valid, we need to validate it by checking its conformance to the reference profile (Pr). To do so instead of performing a full validation and using Pr, we consider a second profile Pv which is created from the reference profile Pr with the same stereotypes and relations as Pr but with a reduced set of constraints. A filtering reduces the constraints of Pr based on the entities of the changeSet. As a result Pv is a subset of Pr.

$$Pv = \text{Filter} (Pr, \text{changeSet}) \quad , \quad Pv \subseteq Pr$$

According to the filtering function:

$$\forall y \in \text{en}_{\text{changeSet}}, (\text{if } \exists x \in \text{Cons}_{Pr}, \text{AST}(y) \in \text{ST}_x) \rightarrow x \in \text{Cons}_{Pv}$$

$$\text{AND } \forall z \in \text{r}_{\text{changeSet}}, \text{if } \exists g \in \text{Cons}_{Pr}, (\text{AST}(z.\text{SrcEn}) \in \text{ST}_g \text{ AND } \text{AST}(z.\text{DstEn}) \in \text{ST}_g) \rightarrow$$

$$x \in \text{Cons}_{Pv}$$

$$\text{AND } \forall s \in \text{ST}_{Pr} \rightarrow s \in \text{ST}_{Pv}$$

$$\text{AND } \forall r \in \text{R}_{Pr} \rightarrow r \in \text{R}_{Pv}$$

6.3.3 The Proof of Partial Validation

We prove by contradiction that the partial validation has the same result as the full (or exhaustive) validation. We make the assumption that the initial configuration model (to which the changes should be applied) is valid i.e. it conforms to its profile. Using the mentioned definitions we prove that if a modified model (M2) conforms to the filtered profile (Pv) then it also conforms to Pr. It means:

$$\text{Conform} (M2, Pv) \rightarrow \text{Conform} (M2, Pr)$$

Prove by contradiction technique is used, which means that we assume that the above statement is not true and show considering the other assumptions a contradiction.

We add the negation of this statement to our assumptions:

Conform (M2, Pv) and \neg Conform (M2, Pr)

Based on the definition of the conform function we can say that there is at least one constraint of Pr that is not valid in M2 or at least one of the entities or relations of M2 does not Respect the profile Pr.

\neg Conform(M2, Pr) \rightarrow ($\exists e \in en_{M2}$, $AST(e) \in ST_{Pr}$, \neg Respect(e, AST(e)) or ($\exists t \in r_{M2}$, $\exists z \in R_{Pr}$, $AST(t.SrcEn) = z.SrcST$, $AST(t.DstEn) = z.DstST$, $\text{Respect}(t, z)$)) or ($\exists x \in Cons_{Pr}$, \neg Valid (M2, x))

At first we show that if the first part of the “or” statement would be true, we face a contradiction:

$\exists e \in en_{M2}$, $AST(e) \in ST_{Pr}$, \neg Respect(e, AST (e))

From the definition of the Pv:

$\forall s \in ST_{Pr} \rightarrow s \in ST_{Pv}$

As the $ST_{Pr} = ST_{Pv}$, the ST_{Pr} in the first statement can be replaced with ST_{Pv} thus:

$\exists e \in en_{M2}$, $AST(e) \in ST_{Pv}$, \neg Respect (e, AST (e))

This is in contradiction with the assumption that Conform (M2, Pv) is true, because :

Conform (M2, Pv) \leftrightarrow ($\forall x \in Cons_{Pv}$, Valid (M2, x)) AND ($\forall e \in en_{M2}$, $AST(e) \in ST_{Pv}$, Respect (e, AST(e)))

Similarly it can be shown that if a relation of model M2 does not respect Pr, a contradiction is encountered.

In the next step we show that if there is a constraint in Pr which is violated by M2, it would contradict to our initial assumptions. Three cases are possible:

First: The constraint x already belongs to Pv:

$$x \in \text{Cons}_{Pv}$$

Which is in contradiction to the assumption that M2 conforms to Pv because:

$$\exists x \in \text{Cons}_{Pv}, \neg \text{Valid}(M2, x) \leftrightarrow \neg \text{Conform}(M2, Pv)$$

Second: The constraint x does not belong to Pv (i.e. $x \notin \text{Cons}_{Pv}$), and constraint x involves the changeSet entities that means the stereotype set of constraint x has at least one stereotype which is applied to at least one of the entities of the change set or constraint x has stereotypes that are applied to the member ends (entities) of a changed relation in the change set:

$$\exists y \in \text{en}_{\text{changeSet}}, \text{AST}(y) \in \text{ST}_x \text{ OR}$$

$$\exists z \in \text{r}_{\text{changeSet}}, (\text{AST}(z.\text{SrcEn}) \in \text{ST}_x \text{ AND } \text{AST}(z.\text{DstEn}) \in \text{ST}_x)$$

According to the Filter function:

$$\forall y \in \text{en}_{\text{changeSet}}, (\text{if } \exists x \in \text{Cons}_{Pr}, \text{AST}(y) \in \text{ST}_x) \rightarrow x \in \text{Cons}_{Pv}$$

$$\forall z \in \text{r}_{\text{changeSet}}, \text{if } \exists x \in \text{Cons}_{Pr}, (\text{AST}(z.\text{SrcEn}) \in \text{ST}_x \text{ AND } \text{AST}(z.\text{DstEn}) \in \text{ST}_x) \rightarrow x \in \text{Cons}_{Pv}$$

And as $x \in \text{Cons}_{Pr}$, it can be concluded that x should also belong to Cons_{Pv} , that is:

$$x \in \text{Cons}_{Pr}, \exists y \in \text{en}_{\text{changeSet}}, \text{AST}(y) \in \text{ST}_x \rightarrow x \in \text{Cons}_{Pv} \text{ OR}$$

$$x \in \text{Cons}_{Pr}, \exists z \in \text{r}_{\text{changeSet}}, (\text{AST}(z.\text{SrcEn}) \in \text{ST}_x \text{ AND } \text{AST}(z.\text{DstEn}) \in \text{ST}_x) \rightarrow x \in \text{Cons}_{Pv}$$

This means that if such constraint exists in $Cons_{Pr}$, it should have been already added in the $Cons_{Pv}$ too because all the constraints that are relevant to the change set should be in the $Cons_{Pv}$.

Third: The constraint x does not belong to the constraint set of Pv (i.e. $x \notin Cons_{Pv}$), and constraint x does not involve the change set entities that means the stereotype set of constraint x does not have any stereotype which is applied to at the entities of the change set or the member ends (entities) of the changed relations in change set:

$$x \notin Cons_{Pv}, \nexists y \in en_{changeSet}, AST(y) \in ST_x$$

$$x \notin Cons_{Pv}, \nexists z \in r_{changeSet}, (AST(z.SrcEn) \in ST_x \text{ AND } AST(z.DstEn) \in ST_x)$$

Based on our assumption:

$$\neg Valid(M2, x) \leftrightarrow \neg AtomicValid(K, x)$$

$$K(en_K, r_K) = \{(e, r) \mid e \in en_{M2}, AST(e) \in ST_x, r \in r_{M2}, (AST(r.SrcEn) \in ST_x \text{ AND } AST(r.DstEn) \in ST_x)\}$$

The constraint x does not have any stereotypes which is applied to the entities or member ends of relations in the change set, so the intersection of the two sets $changeSet$ and K (set of entities and member ends of the relations of the model $M2$ on which stereotypes of constraint x is applied) is empty:

$$K \cap changeSet = \emptyset$$

When none of the entities of K belongs to the $changeSet$, it can be deducted that all the entities of K are in $M1$ model:

$$K \subseteq M1$$

Thus $M2$ model can be replaced with $M1$ in the previous assumption and state that:

$$K(en_K, r_K) = \{(e, r) \mid c \in en_{M1}, AST(e) \in ST_x, r \in r_{M1}, (AST(r.SrcEn) \in ST_x \text{ AND } AST(r.DstEn) \in ST_x)\}$$

And because K is in common between M1, M2, then:

$$\neg \text{Valid}(M2, x) \leftrightarrow \neg \text{AtomicValid}(K, x) \leftrightarrow \neg \text{Valid}(M1, x)$$

And this is a contradiction to our first assumption because:

$$\neg \text{Valid}(M1, x) \leftrightarrow \neg \text{Conform}(M1, Pr)$$

Thus we can conclude that the filtered constraints are sufficient for validating the model.

6.4 Prototype Implementation and Evaluation

In this section we present a preliminary evaluation of our partial validation approach using a prototype implementation and discuss the results.

6.4.1 Implementation Setup

We used the UML profile of the ETF [21] as a configuration profile and applied our partial validation approach to its instances. The ETF UML profile used in our evaluation experiments has 28 stereotypes and 24 OCL constraints defined over these stereotypes.

We implemented the partial validation method in the EMF [10], using the ATL [9] for constraint selection and also the EMF OCL APIs for constraint validation in a standalone java application. The experiments were performed on a machine with an Intel® Core™ i7 with 2.7 GHz and 8 Gigabytes RAM and a Windows 7 operating system.

6.4.2 Evaluation Scenarios

We created an initial ETF model that conforms to the ETF UML profile. The ETF model has 50 entities. We considered three change sets to be applied to this model. In each case a certain number of model entities are randomly selected and changed. This number is 10, 20, and 30 for the three cases, respectively. The selections were made independently from each other. For these cases we compared the number of constraints selected in the constraints model and the total number of constraint checks performed during the partial validation. We also measured the execution time of the full and each of the cases of the partial validation. Each validation test was executed five times and the average was considered as the validation time.

6.4.3 Results and Discussions

Table 6.1 presents the results for the different cases of partial validation in comparison with the full validation. The first row of the table represents the result of the full validation of the model in which all entities are checked for all applicable constraints, i.e. as if all entities have changed. The second, third and fourth rows present the results for the partial validations for 10, 20 and 30 changed entities. As the number of changed entities increases, more constraints are selected, more times they are checked and the validation time increases.

Table 6.1. Partial validation performance results

	Number of Changed Entities	Number of Selected Constraints	Total Number of Constraint Checks Performed	Partial Validation Time (ms)
Initial Model	50	24	70	6933
Test CASE 1	10	8	41	4432
Test CASE 2	20	15	55	5413
Test CASE 3	30	18	57	5845

As it was expected the validation time is proportional to the number of selected constraints which in turn depends on the number of changed entities. However the number of selected constraints is not proportional to the number of changed entities, which is explicable by the characteristics of the ETF profile:

In the ETF profile some entity types have only a few tagged values and constraints while others have a relatively large number of each. Also, the frequency of use of different entity types in an ETF model is different. In a given ETF model the number of component types is typically higher than the number of other entity types. This means that in a random selection of changes the probability is higher to select a change in a component type and with that more constraints are selected. This is the reason that with only 10 changed entities the number of selected constraints is already 8 and these constraints are checked 41 times. The high ratio of checks is further explicable by the fact that the component type is specialized into several specific component types (using the UML generalization in the profile). Thus each child component type inherits the tagged values and the constraints of its parent component types. I.e. if the constraint of a parent stereotype is in the selected constraint set, then that constraint should be checked over all the child entities of that parent.

This shows that the stereotype of the changed entity has a determining role whether using partial validation results in the expected time gain. The characterization of the configuration profile is necessary to determine whether partial validation is beneficial and for which kind of change set.

In this evaluation we did not include the constraint categorization as in these preliminary measurements we focused on the time gain resulting from the partial validation. In this respect we

have shown that there is a time gain, however the results are also showing that further analysis is needed to determine the circumstances. This is important as the constraint selection process itself takes some time and it becomes an additional overhead. Another improvement to our work is to trim down the configuration model and check only the changed entities and those related to them as opposed to all entities of the affected stereotypes. This will reduce the size of the configuration model and therefore improve further the validation time.

6.5 Summary

Runtime configuration validation is required to assure the consistency of the configuration during dynamic reconfigurations of the system. The consistency of the configuration is defined as the correctness of the attribute values and relations of the configuration entities and satisfying the constraints defined over them. A runtime validation should be efficient in the sense that it should impose minimal overhead on the system and it should be completed within a required time frame especially for real-time and highly available systems.

Some existing model-based approaches for re-validation after the changes try to reduce the number of constraints and/or entities that need to be checked [46, 47, 48, 49, 50]. Some of these works use a list of events that can violate the constraints along with the affected entities and add the list to the configuration schema to be able to check the affected constraints [46, 47]. However they cannot handle complex constraints and the approach is not efficient for large systems with high number of constraints and events. Others store queries permanently in memory and update the values of the partial matches used in queries after each model change [49] or they store the validation log of checking each constraint over the model entities and re-validate the model when

the stored parts are changed [48,50]. Thus their approaches have considerable memory consumption.

We proposed a model-based approach of partial validation of configurations. The constraints of the configuration schema have been extended so that they describe the roles of the configuration entities in the constraints. Namely, the leader/follower/peer roles define which entity can impact the other ones in the constraint. For the partial validation we filter the constraints based on the modification request and this role information. This means that we select only those constraints which have at least one constrained entity changed by the modification request. Thus reducing the number of constraints to be checked in a partial validation, which therefore can achieve better performance compared to a full validation. We have also categorized the filtered constraints according to the roles of the changed entities in the constraint to distinguish the cases when there may be a possibility of resolving potential inconsistencies.

Chapter 7

Runtime Adjustment of System Configuration

In this chapter we describe our model-based configuration adjustment as part of the configuration change management framework. The content of this chapter has been published in [22, 72].

7.1 Introduction

Runtime modification requests can cause inconsistencies in the configuration. Such violations or inconsistencies might be resolvable by adding complementary modifications that complete the initial partial set of changes. This is specially the case for a self-adaptive system that needs to adjust itself to changes during runtime [23]. Finding the proper set of complementary modifications is not always straightforward. The reason is that these modifications can affect other configuration entities, which are also involved in other constraints that should not be violated. The initial changes can propagate throughout the configuration and affect other configuration entities up to the point where the constraints are all satisfied or all entities have been considered and no solution is found. Such a change propagation process may result in changing a large number of

entities. This is not desirable because the configuration is a representation of a real system and any changes in the configuration need to be applied on real system entities. Thus, the modifications need to be minimal not to destabilize the system.

In this chapter we define our adjustment resolution with respect to the system constraints and the impact of the system entities on each other. To reduce the number of complementary modifications we determine a propagation scope for solving the violated constraint. The scope is defined with respect to the impact that entities may have on each other (i.e. the leadership information). As we mentioned in previous chapters, the leadership concept is defined based on the relations and dependencies between the configuration entities and their attributes. It reflects that some entities/attributes have dominant or leader role toward others, which should follow them. Using the leadership concept we can direct the propagation. We devise a depth-first incremental change propagation method to reduce the number of modified entities and avoid affecting entities unnecessarily. Determining the propagation scopes and identifying the entities to be modified enable us to formulate the problem as a CSP and use a constraint solver to find the valid modifications for solving the constraints.

7.2 Consistency Preservation through Adjustment

The consistency of a configuration should be preserved throughout the system lifecycle to avoid any mal-functioning of the services and applications deploying this configuration. A configuration model is consistent when it satisfies all the structural and semantic rules, i.e. all the constraints of its metamodel (profile). To ensure consistency, any reconfiguration is checked against the consistency rules. Thus if a constraint is violated because of changes in one or more of its constrained entities, the other entities involved in the constraint can be modified to satisfy the

constraint, i.e. to complement the proposed changes. In turn, these complementary modifications can cause other inconsistencies as the modified entities may be subject to other constraints. Thus, these newly violated constraints need to be handled as well. The modifications propagate in the configuration model to the point where all the constraints are satisfied i.e. the adjustment is successful, or no further modification is possible while still some constraints are not satisfied, i.e. no successful adjustment is possible and the change is rejected.

The adjustment process is defined as a set of complementary modifications and (if necessary) the propagation of these modifications in the model to find a solution which satisfies all the constraints. Thus, a solution includes some entities of the configuration model with new values that along with the other entities of the model satisfy all the constraints. The adjustment process has two steps: first step is to identify the scope of the changes i.e. *what* entities of the model may need to be modified and which constraints may be impacted; and the second step is to modify as few entities as possible in the scope in such a way that all the constraints are satisfied, i.e. address *how* the modifications should be done. In our approach first we identify the propagation scope for each incomplete change by collecting all the entities and constraints that can be affected. Then we try to modify a minimum subset of these scopes to satisfy the violated constraints. We propagate the modifications based on some heuristics to reduce the complementary modifications.

7.2.1 Adjustment Challenges

The configurations of large systems consist of thousands of interrelated entities. In such models, an attempt to resolve the violation of a single constraint can result in changes of multiple entities which in turn may violate other constraints. The changes may propagate in an exponential man-

ner and finally result in changing a large number of entities (the whole configuration model in the worst case). This is not desirable as more changes results in more constraints to solve which requires more time and computation. Moreover more changes in the configuration mean the re-configuration of more system entities in the running system (as the configuration changes need to be applied on system entities). More reconfigurations in the system in turn risk more the system stability - especially undesirable in highly available systems. Thus, it is desirable to keep the changes to the minimum possible. The issue is how to limit the change propagation to reduce the number of changes and reduce the cost of changes.

7.2.2 Preliminary Definitions

In our model-based framework we define the constraints over the stereotypes of the configuration profile. By applying the stereotypes of the profile to the entities of the configuration models, we ensure that the constraints are also applied to all the instances of those stereotypes. As we have both the configuration model and the profile we can figure out the constraints that are applied to the entities of the model. For the sake of simplicity, we use the constraints as part of the configuration model. An example of a model with constraints for the OVF Petstore example is shown on the right hand side of Fig. 7.1. In this figure constraints are shown as ovals and the entities as rectangles. The participation of each entity in a constraint is represented by an edge between the constraint and the constrained entity. The role of the entity in the constraint is shown as a label on this edge (e.g. label “L” represents the Leader role). We use this representation in the rest of this chapter as it focuses on the role of entities in the constraints and depicts how the constrained entities can affect each other. Although the constraints are applied on all the entities of the same context (e.g. constraint C1 is applicable on both DB Tier and Web Tier), for the sake

of simplicity we only show the relation of the constraints to the entities where the constraint can be checked (e.g. constraint C1 is not shown for Web Tier as the PG attribute of the Web Tier does not have a value so there is no need to check C1 for Web Tier).

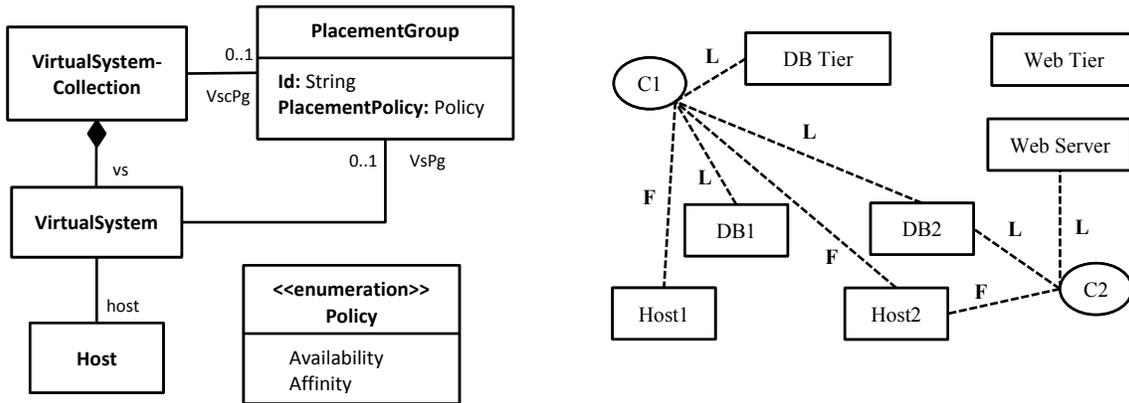


Figure 7.1. Representation of OVF model with constraints and the role of entities in constraints

Formally, we define a configuration model as a tuple $G = \langle En, C, Role, f \rangle$, where

- En is a set of configuration entities,
- C is a set of configuration constraints, e.g. the constraints for assuring the affinity or availability for Virtual Systems in the OVF configuration example,
- $Role$ is a set of leadership roles for the constrained entities, $Role = \{leader, follower, peer\}$,
- f is a function defined over the cross-product of entities and constraints and which associates a role with an entity in a constraint

$f: En \times C \rightarrow Role$, with the following constraints

- For any constraint if there is a leader entity then there is at least one follower and there is no peer entity (note that we may have more than one leader and/or follower in a constraint):

$\forall c \in C$, if $\exists en_x \in En$ with $f(en_x, c) = \text{leader}$ then $\exists en_y \in En$ with $f(en_y, c) = \text{follower} \wedge \nexists en_z \in En$ with $f(en_z, c) = \text{peer}$.

- For any constraint if there is a peer entity all entities involved in the constraint are peers:

$\forall c \in C$, if $\exists en_x \in En$ with $f(en_x, c) = \text{peer}$ then $\forall en_y \in En$ with $f(en_y, c) \neq \text{Nil}$, $f(en_y, c) = \text{peer}$.

We use the term *ChangeBundle* to denote the initial set of changed entities. We cannot change these entities as part of the adjustment process because the user requested to change them. The subset of entities of the *ChangeBundle* that causes violation is referred to as *IncompleteChangeSet* and it is obtained from the validation phase. We refer to an entity in the *IncompleteChangeSet* as an *infringing* entity. An infringing entity is either a leader or a peer entity in the violated constraint because if the entity was a follower, then the change would have been rejected in the validation phase. As mentioned earlier follower entities cannot affect leader entities and thus there is no possibility for any adjustment.

$\text{IncompleteChangeSet} = \{en_x \in \text{ChangeBundle} \mid \exists c \in C \text{ with } f(en_x, c) = (\text{leader or peer}) \wedge c \text{ is not satisfied}\}$.

The *SinkSet* contains entities which have only follower or peer role in all the constraints they are involved in.

$\text{SinkSet} = \{en_x \in En \mid \forall c_y \in C, \text{ if } f(en_x, c_y) \neq \text{Nil} \text{ then } f(en_x, c_y) = (\text{follower or peer})\}$.

The subset of constraints that are violated by the entities of the *IncompleteChangeSet* is called the *ViolatedConstraintSet* and it is also obtained from the validation phase.

$$\text{ViolatedConstraintSet} = \{c \in C \mid \exists en_x \in \text{ChangeBundle with } f(en_x, c) = (\text{leader or peer}) \wedge c \text{ is not satisfied}\}$$

We define a binary relation \triangleright , named *Compulsion*, on the model entities ($en_i, en_j \in En$) as follows:

- $\forall en_i, en_j \in En, en_i \triangleright en_j \Leftrightarrow (\exists c \in C \mid f(en_i, c) = \text{leader} \wedge f(en_j, c) = \text{follower}) \vee (\exists c \in C \mid f(en_i, c) = \text{peer} \wedge f(en_j, c) = \text{peer})$

or

- $\forall en_i, en_j \in En, en_i \triangleright en_j \Leftrightarrow \exists en_k \in En \mid en_i \triangleright en_k \wedge en_k \triangleright en_j$

The *compulsion* relation is transitive by definition.

Propagation Scope

A *propagation scope* is a slice of a configuration model which contains all the entities and constraints that can be affected by the propagation of a change. The propagation scope for an infringing entity is defined by the set of entities (and their associated constraints) that are in a compulsion relation with the infringing entity. A violated constraint may have many infringing entities, but their propagation scopes are equal because they are leaders/peers in the same violated constraint and as such they may equally impact all the followers/peers of this violated constraint. Therefore for a violated constraint we consider the propagation scope of one infringing entity only.

For an infringing entity (en_i) in a violated constraint (c_x), a propagation scope PS_i is a tuple

$\langle E_i, C_i, \text{Role}, f_i \rangle$ where:

- $E_i = \{en_i\} \cup \{en_j \in En \mid f(en_j, c_x) = (\text{follower or peer})\} \cup \{en_j \in En \mid \exists en_k \in E_i \text{ with } f(en_k, c_x) = (\text{follower or peer}) \wedge en_k \triangleright en_j\}$,
- $C_i = \{c \in C \mid \exists en_x \in E_i \setminus \{en_i\} \wedge f(en_x, c) \neq \text{Nil}\}$,
- $\text{Role} = \{\text{leader, follower, peer}\}$, and
- f_i is the project of f on $E_i \times C_i$

Fig. 7.2 shows an example of a model with an infringing entity E1 which violates constraint C1.

The propagation scope PS_1 for entity E1 is shown as the enclosed part in the figure.

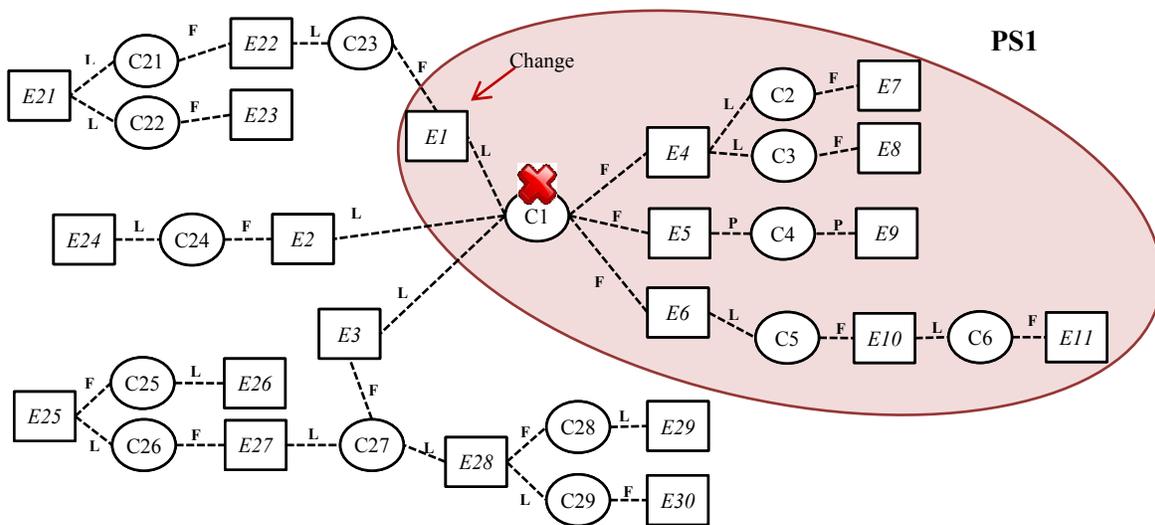


Figure 7.2. An example of the propagation scope for an infringing entity

Propagation Path

A *path* in a propagation scope is defined as an ordered set of entities that starts with the infringing entity. The path is created following the compulsion relation between the entities. It ends if

the entity is a sink entity in the scope or if it is only leader/peer for entities of the path itself - to avoid cycles. The paths are used for change propagation within each scope.

For an infringing entity en_i , a $Path_x$ is an ordered set of entities of the scope PS_i (in which en_i is the head of the path) and is defined as follows:

- $en_i \in Path_x$,
- $\forall en_j \in E_i, en_j \in Path_x$ iff
 - $en_i \triangleright en_j$, and
 - $\forall en_k \in Path_x, (en_k \triangleright en_j) \vee (en_j \triangleright en_k)$
- $\exists en_k \in Path_x$ such that
 - $en_k \in SinkSet$, or
 - $\exists en_j \in E_i, en_k \triangleright en_j \Rightarrow en_j \in Path_x$

The collection of all paths in a propagation scope is called a PathCollection. In addition to the infringing entity, which is common to all the paths in a PathCollection, different paths in the PathCollection may have other entities in common as well.

Fig. 7.3 shows the propagation scope for the infringing entity (E1) which violates constraint C1. As E1 is a leader entity in C1, its change can propagate to the follower entities of C1 which results in multiple paths (i.e. Path A, Path B, Path C, Path D). The paths start with E1 and end with an entity with only a follower role or they end with an entity with only a peer role whose other peers have already been visited in the path. E.g. Path A ends with E7 which is a follower role in C2 and C6, and Path C ends with E9 which is peer in C4 and its other peer (E5) already exists in the Path C, Note that the propagation scopes and propagation paths can be determined using depth-first search algorithms from the graph theory [24].

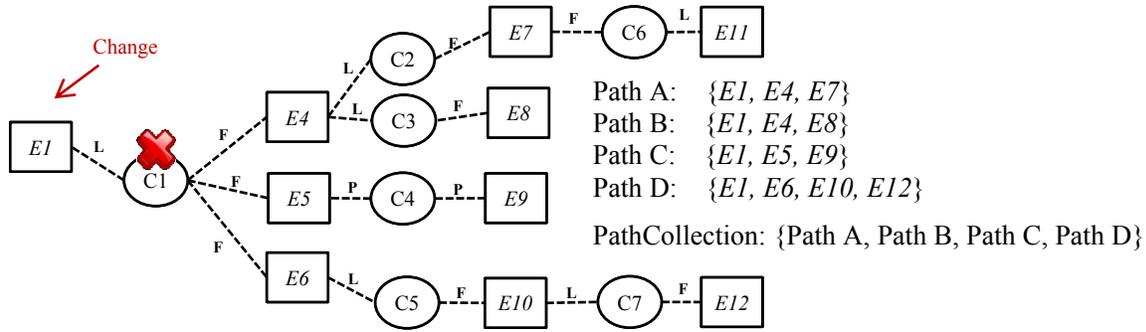
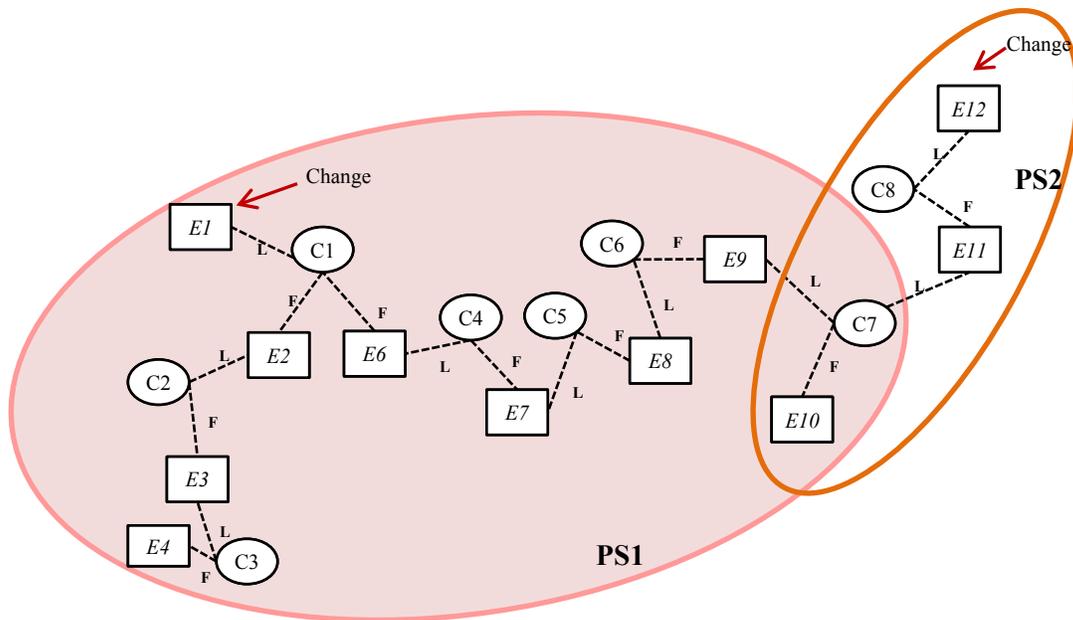


Figure 7.3. A PathCollection with multiple paths for an infringing entity

7.3 The Adjustment Process

Determining the propagation scope allows us to isolate the problem and ensures that all the entities that can possibly be impacted during change propagation are already gathered in the scope. Multiple changes that are requested as a change bundle may cause multiple constraint violation. In this case for each infringing entity of a violated constraint, a propagation scope and PathCollection need to be calculated. If these propagation scopes are disjoint (no common entity between them), then we try to solve each scope individually. On the other hand if the propagation scopes have an intersection, the scopes should not be solved separately. Our assumption is that the change bundle contains related changes and if the scopes are overlapped, we may not find a solution for each scope separately and a solution is possible only when they are considered together as a single problem. To further explain this, consider the model of Fig. 7.4 which shows an example of overlapping propagation scopes and their intersection. In this model two entities E1, E12 are infringing entities and for each the propagation scope (PS1, PS2 respectively) and PathCollection are calculated. Now let us assume for the PS1 of E1, Path A is selected and satisfiable. For the PS2 we have only one path, path X which is selected but not satisfiable which results in rejecting the changes. However if we selected Path B of the PS1 then the PS2 would also be-

come satisfiable because the changes in the PS1 affect the entities of the intersection of the two scopes.



Path A : {E1, E2, E3, E4}
 Path B : {E1, E6, E7, E8, E9, E10}

Path X : {E12, E11, E10}

Figure 7.4. Multiple constraint violations with overlapping propagation scopes

7.3.1 Grouping the Overlapping Scopes

Not to miss a solution (such as the one described for Fig. 7.4), overlapping scopes need to be solved together. For this purpose we group overlapping scopes together into a Group. For each Group an *Intersect* captures the common entities of the overlapping scopes. The Groups are disjoint (i.e. they have no common entities) as the overlapping scopes are supposed to be collected in the same Group. An example of a Group and its Intersect formed from three propagation scopes PS1, PS2, and PS3 is shown in Fig. 7.5. In this figure E30 is the common entity between PS1 and PS2 and on the other hand E22, E23 are the common entities between PS2 and PS3. The three scopes form Group1 with the Intersect of E30, E22, and E23.

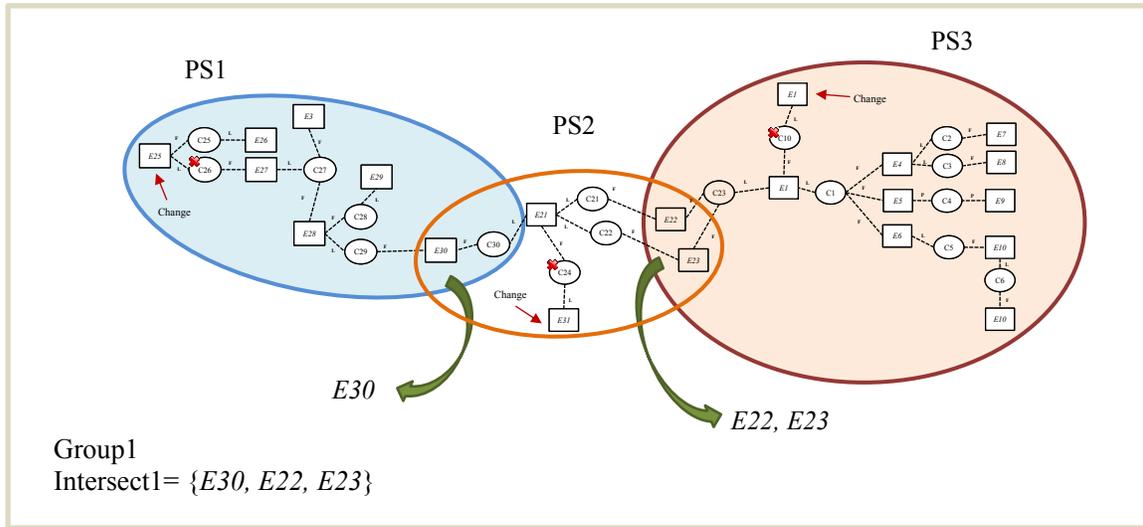


Figure 7.5. The formation of a group and its Intersect from overlapping scopes

The GroupSet is the collection of the Groups and the IntersectSet is the collection of the Intersects of the Groups. The process of forming the Groups of a GroupSet is the following:

Each of the scopes is compared with the Groups of the GroupSet and if the scope has common entities with a Group, it is added to the Group and the Intersect of the Group is updated accordingly with the common entities. If a scope cannot be added to any of the existing Groups, a new Group is created with an empty Intersect (i.e. how the first Group is formed). Algorithm 7.1 describes how the Groups are formed with overlapping scopes. The input of the algorithm is the collection of all propagation scopes (i.e. PropagationScopeCollection) and the outputs are the GroupSet and IntersectSet. At the beginning both GroupSet and IntersectSet are empty (lines 2-3). Each propagation scope is compared with the groups of the GroupSet to find out the Group with which it has common entities (lines 4-18). An integer variable K is used to keep track of the addition of the scope to a Group (K keeps the index of that Group). At the beginning K is set to value (-1) for the scope (line 5). If a Group with common entities with the scope is found and the

scope is not already added to any groups (i.e. $K=-1$), the scope is added to the Group and the common entities are added to the Intersect of the Group and K is set to the index of the Group (lines 6-12). After adding the scope to a Group, it should also be checked if the scope has common entities with the other groups of the GroupSet as well. This checking is required because the scope may have common entities with more than one group. In this case such groups should be merged with the first one (the Group in which the scope is already added), the Intersect of the Group is updated accordingly and the already merged groups are deleted. (lines 12-18).

Algorithm 7.1: Grouping the Overlapping Scopes

Input: PropagationScopeCollection,

Output: GroupSet, IntersectSet

```

1: // Grouping the overlapping scopes
2: GroupSet:={}
3: IntersectSet:={}
4: For each PSi in PropagationScopeCollection
5:     K:=-1
6:     For (j:=0; j<|GroupSet| ; j++)
7:         If (Epsi ∩ EGroupj ≠ ∅) then
8:             If (K == -1)
9:                 Intersectj := Intersectj ∪ (Epsi ∩ EGroupj)
10:                Groupj := Groupj ∪ PSi
11:                K:=j
12:            Else //PSi is already added to Groupk
13:                Groupk := Groupk ∪ Groupj
14:                Intersectk := Intersectk ∪ Intersectj ∪ (Epsi ∩ EGroupj)
15:                Delete Groupj
16:                Delete Intersectj
17:                j--
18:            End if
19:        End if
20:    End For
21: // If PSi has no intersection with the groups, create a new group with the PSi and an empty intersection for that
22:    If (K == -1) then
23:        GroupSet:= GroupSet ∪ {PSi}
24:        IntersectSet:= IntersectSet ∪ { }
25:    End if
26: End For
27: Return GroupSet, IntersectSet

```

After checking all the Groups, if the K is still (-1), i.e. no Group can be found which has common entities with the scope, then a new Group is created with the scope and with empty Intersect (lines 22-25). The group finding procedure is repeated for each scope, and finally the calculated GroupSet and IntersectSet are returned as the output (line 27).

Once Groups are created from the scopes, we try to solve each Group separately because all the related scopes are already gathered in a group. We recognize two types of Groups: (1) groups with a single scope (i.e. with an empty Intersect) and (2) groups with multiple scopes (i.e. with a not-empty Intersect). We use different methods for solving each type, i.e. *the Depth-first Incremental Change Propagation* for each Group with a single scope, and the *Path Bonding* for groups with multiple scopes. In the following we define the two methods.

7.3.2 Depth-first Incremental Change Propagation for Solving Groups with a Single Scope

The PathCollection of a propagation scope for an infringing entity may contain different paths. To find a solution (i.e. the complementary changes which satisfies the constraints of the scope), one path at a time is selected and tried by modifying the entities in the path. If no solution can be found in the selected path, another path from the PathCollection is selected. The path selection ends when either a solution is found or when all the paths are exhausted. If all the paths are exhausted and no solution is found, all the paths of the PathCollection are considered together to find a solution and consider changing multiple followers of different constraints. It is possible that no solution exists to the problem. However if there is a solution, we will not miss it as all the constraints of the scope are considered if no solution was found in any of the paths.

The paths in the PathCollection can be ordered according to their lengths in terms of entities. As we aim at changing the least number of entities we try to find a solution for the shortest path possible. Moreover, we use an incremental change propagation to select the minimum number of entities in the path for modifications. In each increment we try to find new values for a selected follower or peer entity of the violated constraint. In the first increment the selected entity is the entity of the path which is directly related to the infringing entity of the scope. In the second increment the third entity in the path is selected, which is directly related to the previously added entity, and so on. Each selected entity, which is not at the end of the path, participates in two sets of constraints. The set of constraints, in which the selected entity has the follower role, it is called MandatoryC as it contains the constraints, which are mandatory to satisfy by the change of the selected entity. The second set of constraints includes the constraints in which the selected entity has a leader or peer role. We call this set RelaxC. We try to find a change for the selected entity that satisfies the constraints in this set as well, but if we cannot we can relax the problem by dropping these constraints. This is because the selected entity has a leader/peer role in these constraints and if these constraints are violated we can try to resolve them in the next increment by selecting the next entity in the path (which is a follower or peer in RelaxC).

In each increment the purpose is to find a change for the selected entity which satisfies all the constraints in MandatoryC and RelaxC. If no solution can be found, the constraints in RelaxC are dropped to relax the problem. If by doing so the problem becomes solvable, we add a new increment and repeat the process for the selected entity. At any increment if a solution is found the propagation stops. If we cannot find a solution after removing the RelaxC constraints, the selected path is unsolvable. Note that in the last increment when the selected entity is the last entity of

the path, we only need to solve MandatoryC as RelaxC is empty. Fig. 7.6 shows an example of applying the depth-first incremental propagation. In this figure E1 is the infringing entity and C1 is the violated constraint. At first as shown in stage (a) of the figure, Path A which is one of the paths with the shortest length in the PathCollection is selected (Path B and Path C also have the same length and could have been selected as each has the length of three similar to Path A).

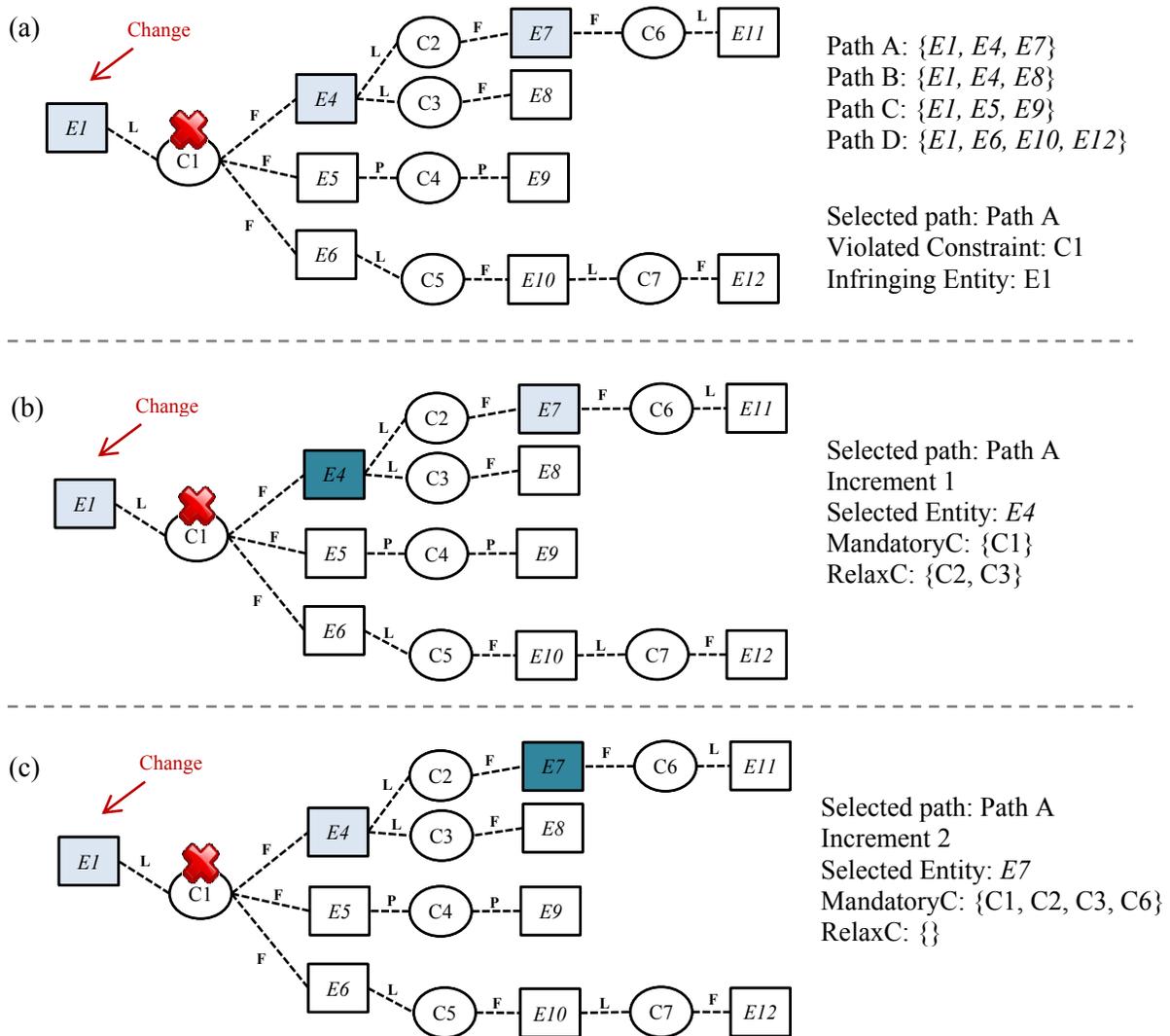


Figure 7.6. Depth-first incremental change propagation

In Increment 1 shown as stage (b) in the same figure, the second entity of Path A i.e. E4 is selected for the change propagation. The violated constraint C1 is added to the MandatoryC and because E4 has the leader role in C2, C3 these constraints are added to RelaxC. As we cannot find a new value for E4 to satisfy all constraints of both MandatoryC and RelaxC in the first increment, the problem is relaxed by disregarding RelaxC constraints. Assuming a new value for E4 can be found to satisfy the MandatoryC constraints therefore RelaxC is added to MandatoryC and RelaxC is emptied and we proceed to Increment 2 which is shown in stage (c) of Fig. 7.6. In this increment the next entity of Path A i.e. E7 is selected. E7 has a follower role in both C2, C6 (i.e. E7 is a sink entity), thus both constraints are added to Mandatory C and because E7 is the last entity of the Path, RelaxC remains empty. If new values can be found for E7 and E4 that satisfy all the constraints of MandatoryC (i.e. C1, C2, C3, C6), we have a solution, otherwise another path should be tried. Algorithm 7.2 describes our depth-first incremental change propagation process. The inputs of the algorithm are the propagation scope for the infringing entity, its PathCollection, the constraint violated by this infringing entity, and the ConstraintSet of the propagation scope. The output is the Solution obtained from the constraint solver. First the PathCollection is sorted based on the length of its paths (line 2). In the beginning MandatoryC consists of the initially violated constraint. The paths of the PathCollection are selected one at a time and in each selected path we follow the incremental propagation by selecting the next entity and considering its constraints from the ConstraintSet (lines 9-17). If there is a solution that satisfies the constraints in both MandatoryC and RelaxC, then the solution is returned and the algorithm terminates (lines 18-21). Otherwise, if the constraints in MandatoryC are not satisfiable (the Solution is empty), the path is unsolvable (lines 22-23). In the other case when the constraints in

MandatoryC are satisfiable but the constraints in RelaxC are not, we proceed to the next increment and select next entity of the path and repeat the same steps until we find a solution or we reach the end of the path. If the path is unsolvable then all the constraints except the Violated Constraint are removed from MandatoryC (line 29) and the next path of the PathCollection is selected. After exploring all the paths if no solution is found, we try to find a solution by considering all the paths together, which means giving all the constraints in ConstraintSet of the scope simultaneously to a constraint solver (line 31).

Algorithm 7.2: Depth-first Incremental Change Propagation

Input: PropagationScope, ViolatedConstraint, PathCollection, ConstraintSet

Output: Solution

```

1: //Sort the PathCollection based on the length of the paths
2: Sort(PathCollection[])
3: UnSolvablePath:= False
4: SolutionFound:=False
5: MandatoryC:={ViolatedConstraint}
6: RelaxC:={}
7: For (j:=0; j<|PathCollection|&SolutionFound==False; j++)
8:   SelectedPath:= PathCollection[j]
9:   For(i:=1; i<|SelectedPath| & UnSolvablePath==False; i++)
10:    entity:= SelectedPath[i]
11:    For each constraint in ConstraintSet
12:      If ( $f(\text{entity}, \text{constraint}) = \text{leader or peer}$ ) then
13:        RelaxC:= RelaxC  $\cup$  {constraint}
14:      Else if ( $f(\text{entity}, \text{constraint}) = \text{follower}$ ) then
15:        MandatoryC:= MandatoryC  $\cup$  {constraint}
16:      End if
17:    End for
18:    Solution:= Solve(MandatoryC  $\cup$  RelaxC)
19:    If (Solution $\neq$ {}) then
20:      SolutionFound:=True
21:      Return Solution
22:    Else if (Solve(MandatoryC)=={}) then
23:      UnSolvablePath:= True
24:    Else
25:      MandatoryC:=MandatoryC  $\cup$  RelaxC
26:      RelaxC:={}
27:    End if
28:  End for
29:  MandatoryC:={ViolatedConstraint}
30: End for
31: Solution:= Solve(ConstraintSet)
32: Return Solution

```

7.3.3 Path Bonding for Solving Groups with Multiple Scopes

The scopes of a group need to be solved together. To avoid changing all entities of a Group and reduce the number of changed entities, we proceed as follows: In each Group all the paths which have common entities with the Intersect of the Group are selected to form the *BondedPath* of the Group. In other words we bond the related paths and disregard the other paths which do not have common entities with the Intersect of the Group. The entities of the bonded paths of each Group are our primary candidates for the complementary changes.

Fig. 7.7 represents the path selection for Group1. For the sake of simplicity only the first and the last entity of each path is represented and the other entities of the paths are shown with an arrow from the first to the end of the path. In the path selection, all the ones which have an end in the Intersect of the Group are selected.

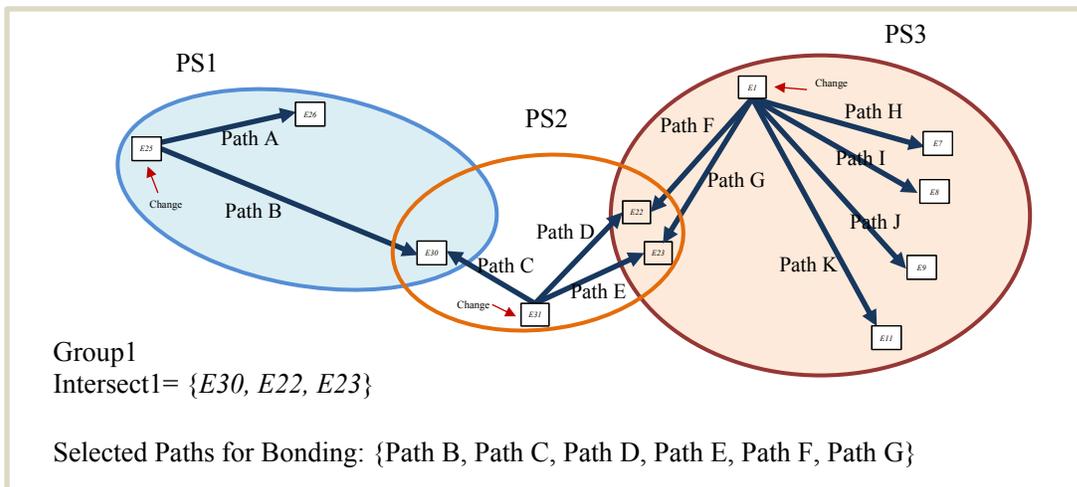


Figure 7.7. Selecting the paths of the Group for the bonding

By grouping the scopes and bonding their paths we address the fact that the initial changes that are related to each other are requested in the same change bundle. Thus when potential incon-

sistencies are detected for a requested change bundle, there is a high chance that the solution is possible only by considering the related scopes together.

Similarly to the depth-first incremental change propagation, for each entity in the BondedPath, the mandatory constraints (MandatoryC) need to be identified and satisfied by the complementary changes. The MandatoryC in this case contains all the constraints in which the entities of the BondedPath are participating (as leader, follower or peer). Unlike in case of the single constraint violation, we cannot perform an incremental propagation and thus there is no need for the RelaxC. We try to find a solution by considering the BondedPath of a Group. If we cannot find a solution that satisfies all the constraints in MandatoryC, then we need to consider the other paths of the Group as well. Algorithm 7.3 describes the path bonding and finding the solution for groups with multiple scopes. The Group, its PathCollectionSet, its Intersect, its ConstraintSet and the IncompleteChangeSet are the input of the algorithm and the output is the Solution for the group. For the given Group a BondedPath, a MandatoryC and a Solution are initialized at the beginning (lines 1-3). The BondedPath is calculated for the Group i.e. for each infringing entity of the IncompleteChangeSet, which also belongs to the Group (line 5). We select the paths from its PathCollectionSet that has common entities with the Intersect of the Group (line 6). The identified paths are added to the BondedPath of the Group (lines 6-10).

Next the constraints related to the entities of the BondedPath are added to MandatoryC (lines 13-19). A solution for the Group should satisfy all the MandatoryC constraints (line20). If such a solution exists (Solution is not empty), it is returned as the output i.e. the ultimate solution for the Group (lines 21-22). If the BondedPath is not satisfiable (Solution is empty) then all the paths of

the Group are considered and all the constraints of the Group are added to MandatoryC (lines 23-31). The result of solving the MandatoryC is the solution of the Group (line 32). If the constraints of the MandatoryC are satisfiable the Solution is not empty, otherwise the Solution is empty. And finally the Solution (empty or not) is returned as the output of the algorithm (line 34).

Algorithm 7.3: Path Bonding and Its Resolution

Input: PathCollectionSet, Group, Intersect, IncompleteChangeSet, ConstraintSet

Output: Solution

```

1: Solution:={}
2: BondedPath:={}
3: MandatoryC:={}
4: // Bonding the related paths of each Group
5: For each enj in IncompleteChangeSet && Groupi
6:   For each pathk in PathCollectionj
7:     If (pathk ∩ Intersecti ≠ {}) then
8:       BondedPath:=BondedPath ∪ pathk
9:     End if
10:   End For
11: End for
12: //Identifying the constraints related to the entities of the bonded path
13: For each enx in BondedPath
14:   For each cy in ConstraintSet
15:     If ( f(enx, cy) ≠ Nil) then
16:       MandatoryC:= MandatoryC ∪ {cy}
17:     End if
18:   End for
19: End for
20: Solution:=Solve(MandatoryC)
21: If (Solution≠{}) then
22:   Return Solution
23: Else
24:   MandatoryC:={}
25:   For each enx in Egroupi
26:     For each cy in ConstraintSet
27:       If ( f(enx, cy) ≠ Nil) then
28:         MandatoryC:= MandatoryC ∪ {cy}
29:       End if
30:     End for
31:   End for
32:   Solution:=Solve(MandatoryC)
33: End if
34: Return Solution

```

7.3.4 The Overall Adjustment

As mentioned earlier for each type of group a different method is used (i.e. depth-first incremental propagation for the groups with a single scope and path bonding for the groups with multiple scopes). Algorithm 7.4 describes the overall approach to the adjustments, which includes these two methods for solving the groups. The inputs of the algorithm are PathCollectionSet, GroupSet, IntersectSet, IncompleteChangeSet, ConstraintSet, and the ViolatedConstraintSet. The algorithm tries to solve each Group in the GroupSet resulting in a PartialSolution. The final Solution for the incomplete change set is the union of all these PartialSolutions or it is empty. Thus the output is empty if any of the Groups is unsolvable; otherwise it contains the complementary changes for adjusting the configuration model. The algorithm starts by initializing the Solution as an empty set (line 1). For each Group in the GroupSet a PartialSolution and a ViolatedConstraint set are initialized (line 3-4). If the Intersect of the Group is Null, i.e. there is only one scope in the Group then the incremental change propagation method is used for solving the Group (lines 5-8). If the Intersect of the Group is not Null, i.e. there is more than one scope in the Group then the path bonding method is called to solve the Group (lines 9-12). In either case the PartialSolution of the Group is obtained. If the PartialSolution is not empty (i.e. the Group is solvable) then the returned PartialSolution is added to the final Solution (lines 13-14) and the procedure repeats for the next Group. If the PartialSolution is empty it means that the Group is not solvable and thus there is no final Solution (lines 15-18). The reason is that a Solution is only complete when all the Groups of the GroupSet are solved. Even if one Group is unsolvable, we do not have a complete Solution and the adjustment is not possible which means the inconsistencies caused by the infringing entity cannot be resolved. At the end if all Groups in the GroupSet are solvable,

the ultimate Solution is returned (line 20) containing all the complementary changes needed for the adjustment.

Algorithm 7.4: Overall Approach to the Adjustment

Input: PathCollectionSet, GroupSet, IntersectSet, IncompleteChangeSet, ConstraintSet, ViolatedConstraintSet

Output: Solution

```

1: Solution:={ }
2: For each Groupi in GroupSet
3:   PartialSolution:={ }
4:   ViolatedConstraint:={ }
5:   // Depth-first Incremental ChangePropagation is called for a group with a single scope (Null Intersect)
6:   If (Intersecti≠Null) then
7:     ViolatedConstraint:= Select (scopej, ViolatedConstraintSet)
8:     PartialSolution = IncrementalPropagation (Propagation Scopej, ViolatedConstraint , PathCollectionj,
9:     ConstraintSetj)
9:   // Bonding Path is called for a group with multiple scopes
10:  Else
11:    PartialSolution = BondingPath (PathCollectionSet , Groupi , Intersecti , IncompleteChangeSet, Con-
12:    straintSeti)
12:  End if
13:  If (PartialSolution ≠ { }) then
14:    Solution=Solution ∪ PartialSolution
15:  Else
16:    Solution={ }
17:    Return Solution
18:  End if
19: End for
20: Return Solution

```

7.3.5 Complexity Analysis for the Overall Adjustment Process

In our proposed solution we use some heuristics to reduce the number of complementary modifications. However they introduce some overhead. As the overall solution consists of some pieces, we discuss the time complexity of each piece separately and the complexity of the overall solution is the aggregation of the complexity of the pieces.

The adjustment resolution starts by calculating the propagation scope and PathCollection for the infringing entities. The propagation scope and the PathCollection can be calculated simultaneously as they follow the same logic. The complexity is similar to the complexity of traversing a

graph with depth-first search and in worst case it is $O(b^d)$ where b is the branching factor (in our case number of constrained entities in the constraints) and d is the depth of the search (in our case the longest path length) [24]. If there are no cycles in the configuration model between the constrained entities (i.e. it is a tree), the complexity of the scope/path creation is $O(n)$ where n is number of constrained entities in the model; in the worst case all entities are visited once. This calculation is done m times where m is the number of violated constraints. Thus $O(m \times (b^d))$ is the complexity in worst case and where there is no cycle (tree-based structure) the complexity is $O(m \times n)$.

The second part is the grouping of overlapping scopes. In the grouping algorithm every scope is checked with the existing groups to find out if it has common entities with them. So the worst case scenario is when we have maximum number of groups. The maximum number of groups equals to the number of scopes (when each group has only one scope) and in worst case scenario the number of scopes equals to the number of violated constraints (i.e. infringing entities are violating different constraints and make distinct scopes). For the first scope the algorithm checks 0 groups and creates the first group, the second scope is checked with one group and creates the second group. This continues until it reaches to the m^{th} scope. For the m^{th} scope it checks $(m - 1)$ group. So in total the algorithm performs $(m - 1) + (m - 2) + (m - 3) + \dots + 1 + 0$ checks and we can say the complexity of the grouping is $O(m^2)$.

The third part of the solution is the solving of the groups; using the depth-first incremental propagation for groups with a single scope and the path bonding for the groups with multiple scopes. The complexity of the depth-first incremental propagation for each scope includes sorting the PathCollection and traversing the paths. If α is the number of groups with a single scope, the

complexity of solving all groups with a single scope is $O(\alpha \times p \times \log p) + O(\alpha \times p \times d)$ where p is the average number of paths in the groups, and d is the average path length.

On the other hand for solving each group with multiple scopes by path bonding, all paths of each scope are checked to select the ones which have common entities with the intersect of its group. Therefore the complexity of solving all groups with multiple scopes is $O((m - \alpha) \times p \times d)$ where p is the average number of paths in the groups, and d is the average path length and m is the number of scopes (i.e. the number of violated constraints).

Thus the total complexity of the adjustment is calculated as follows:

[*Propagation scope*] + [*Grouping*] + [*Depth first incremental propagation / BondedPath*]

$[O(m \times (b^d))]$ + $[O(m^2)]$ + $[O(\alpha \times p \times \log p) + O(\alpha \times p \times d) + O((m - \alpha) \times p \times d)]$

$n =$ *Total number of entities in the model*

$m =$ *Number of violated constraints = Number of scopes*

$b =$ *Branching factor (average number of constrained entities in the constraints)*

$d =$ *The average path length*

$p =$ *Average number of paths in the scope*

$\alpha =$ *Number of groups with a single scope (i.e. a portion of m)*

Although the calculations for determining the scopes and using the heuristics may seem imposing some overhead to the adjustment, the approach is still beneficial compared to the traditional constraint solving solutions, because:

1. We try to limit the size of the problem by calculating the propagation scopes and using the discussed adjustment heuristics to reduce the complexity of the problem. The complexity of SMT problems is generally exponential or even worse when it comes to the combination of different theories (e.g. linear integer arithmetic, theory of arrays, etc.)[69,70].
2. In general constraint solving solutions intend to find valid values for the variables of the constraints; however they do not consider any restriction to minimize the number of changes. As we do the adjustment for the runtime models, we are concerned about minimizing the changes.

7.4 Prototype Implementation and Experimental Evaluation

We implemented a prototype of the overall adjustment resolution in Microsoft visual studio and used Microsoft Z3 [25] as our constraint solver. We used an ETF model as our configuration model [21]. The UML profile of the ETF model consists of 26 stereotypes and 28 constraints. The initial model conforming to this profile contains 40 entities with a total number of 85 attributes.

To translate this model to a constraint satisfaction problem we follow the partial evaluation of constraints proposed by Song et al. in [26]. A variable is created for each entity or attribute of the model which is involved in a constraint. During the translation process the constraints are also partially evaluated and constraint instances are generated with the variables. For each constraint in the profile we may generate multiple constraint instances (depending on the number of model entities conforming to the context stereotype of the constraint). The created variables and constraint instances are the input to our prototype together with the IncompleteChangeSet and the

violated constraints (obtained from the validation phase). The tests are performed on a machine with an Intel® Core™ i7 with 2.7 GHz and 8 Gigabytes RAM and a Windows 7 operating system.

7.4.1 Evaluation Scenarios

We measure the execution time and number of necessary complementary changes with our approach, and compare them to the measurements for runs for the “total change” resolution where the changes are given to the solver with the subset of the configuration model that they can impact directly or indirectly (i.e. without considering any leadership information). We consider our adjustment resolution in two cases: in first case, i.e. the overall adjustment resolution, we use the created paths and solve each group with the paths (i.e. use of depth-first incremental propagation for the groups with one scope and use of path bonding for the groups with multiple scopes in the group) and in the second case, which we call it “group-based” resolution, the complete set of calculated groups of the propagation scopes are given to the solver. We should indicate that the second case (group-based resolution) is actually the worst case of the overall adjustment; this means that if no solution can be found by depth-first incremental propagation or path bonding for a group, then the whole group is considered for the modifications.

Three scenarios are considered: (1) Solving a group with a single scope, (2) Solving multiple groups, and (3) Detection of the not-adjustable changes for multiple groups. For the first two scenarios we measure the number of complementary changes and the time needed for calculating them. For the last scenario we only do the comparison of the execution time of the overall adjustment resolution with the total change resolution.

7.4.2 Solving a Group with a Single Scope

In this scenario we consider 10 test cases. In each test case one entity is changed randomly to violate a constraint. For each test case we measure the execution time and the number of necessary complementary changes of the adjustment resolution (more specifically the depth-first incremental propagation method) and compare them to the execution time and number of complementary changes for the group-based resolution and for the total change resolution. The results are shown in the diagrams of Fig. 7.8 and Fig. 7.9. The results reported in Fig. 7.8 show that the number of complementary changes with the overall adjustment resolution is always less than the total change resolution and it is also less or in some test cases equal to the number of changes for group-based resolution. The number of changes are equal for the two resolutions (overall adjustment and group-based) when the depth-first incremental propagation is unable to solve the

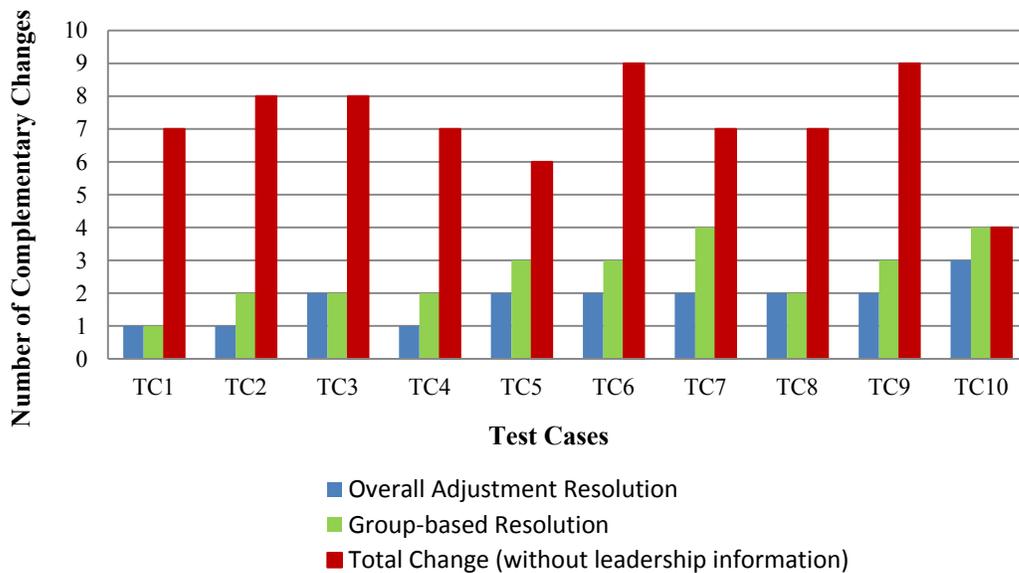


Figure 7.8. Comparison of the number of complementary changes using the overall adjustment resolution versus the group-based and total change resolutions

scope by considering a single path and has to consider all the paths of the scope together or when there is only a single path in the scope and the number of increments is equal to the path length (both cases make the incremental and group-based resolution the same).

Fig. 7.9 shows the comparison of the execution times of the overall adjustment resolution with the execution times of the group-based resolution and measurements of the total change resolution. The measured times indicate that the overall adjustment resolution (i.e. the depth-first incremental propagation in this scenario) usually takes less time than the group-based and total change resolutions. However, in test case TC10 the execution time of the overall adjustment is higher than the execution time of the group-based resolution. The reason is that in this test case the depth-first incremental propagation was unable to find a solution in one path and had to consider the group (containing one scope) for the change (similar to the group-based resolution) but as it has already tried each path individually, the total time for the overall adjustment is summed up and becomes higher.

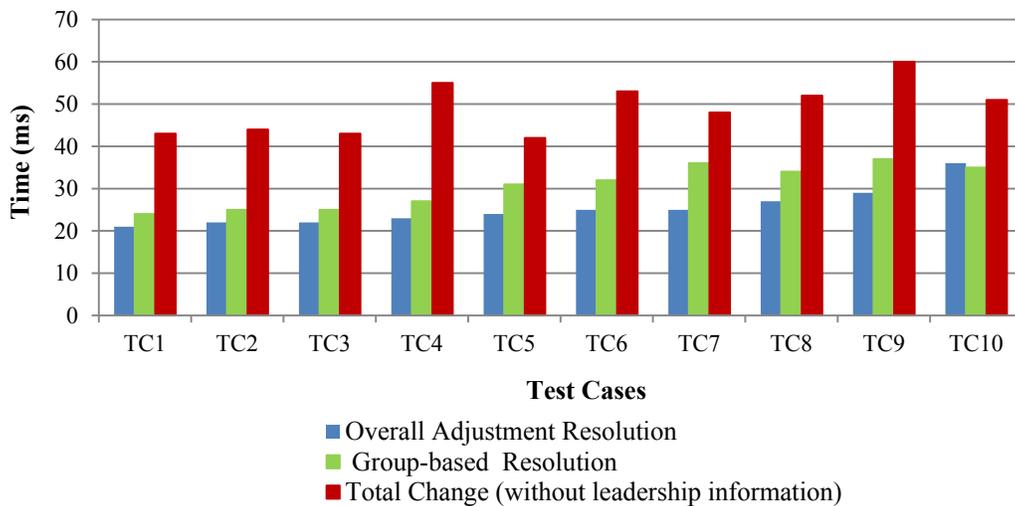


Figure 7.9. Comparison of the execution time using the overall adjustment resolution versus the group-based and total change resolutions

7.4.3 Solving Multiple Groups

In this scenario we consider 14 test cases each with a random `IncompleteChangeSet`. Similar to the previous scenario we measure the execution times and the number of necessary complementary changes when using our adjustment and compare them to the measurements of group-based and total change resolutions. Fig. 7.10 and Fig. 7.11 show the results of our experiments. We should indicate that measurements for the overall adjustment resolution includes the group creation and solving the groups one by one with either depth-first incremental propagation or by path bonding and, if no solution was found then try to solve the group by considering the whole group (similar to the group-based resolution).

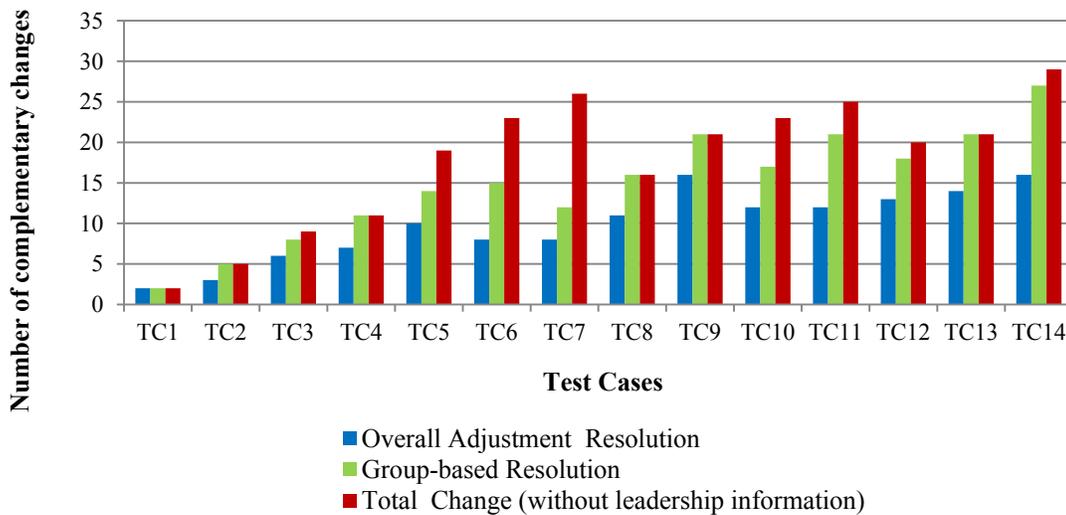


Figure 7.10. Comparison of the number of complementary changes using the overall adjustment resolution versus, the group-based and the total change resolutions

As the chart in Fig. 7.10 shows, the number of complementary changes of the overall adjustment is always less than the number of changes which are considered by the group-based or total change resolutions. In our tests it happened that one or two groups would require the resolution by group-based resolution but as the other groups could be solved by the depth-first incremental

propagation or by path bonding, the overall adjustment resolution have a better overall outcome and reduces the number of complementary changes in each test case.

The comparison of the execution times of the overall adjustment, group-based and the total change resolutions shown in Fig. 7.11 also indicates that our resolution is faster than the total change in most of the cases. The test cases in which our resolution has a similar or higher execution times compared to the total change resolution are the situations that at least one group could not be solved with the depth-first incremental propagation or the path bonding, thus the whole group is considered to be changed similar to the group-based resolution.

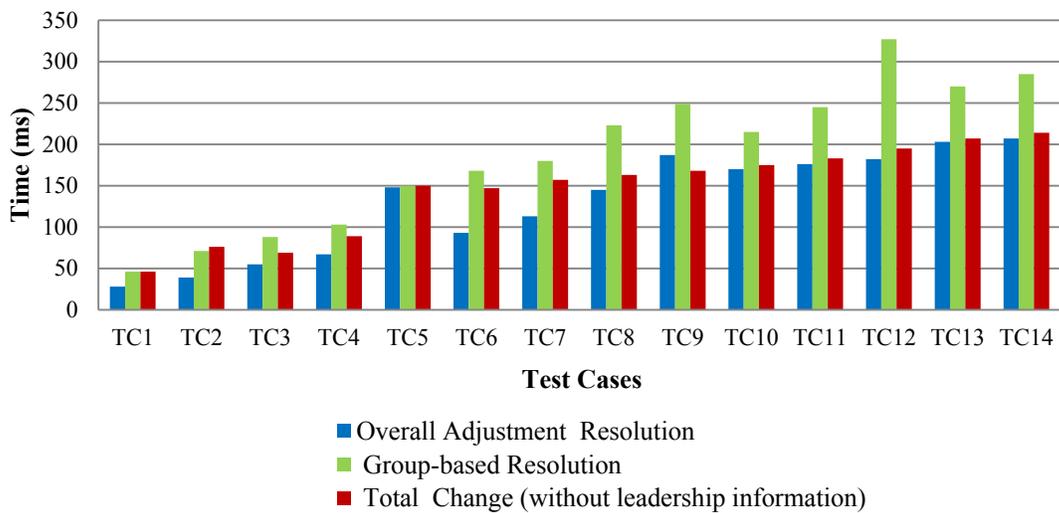


Figure 7.11. Comparison of the execution times using the overall adjustment resolution versus, the group-based and the total change resolutions

Overall consideration of the measurements and the comparisons show that the overall adjustment resolution reduces the number of complementary changes but this is achieved by doing some further calculations (for creating the paths and groups) that can be time consuming specially when the constraint violations increases. Although the execution times of the adjustment are still

lower than the total change for most of the test cases, it should be used with cautious in time critical applications.

7.4.4 Detection of the Not-adjustable Changes for Multiple Groups

Another scenario which we considered for evaluating our work is to find out how fast it can detect the not-adjustable changes. For this scenario we compare the execution times of the overall adjustment resolution with the execution time of the total change resolution and disregard the group-based resolution. The reason is that in this scenario the overall adjustment includes the group-based resolution and if no solution can be found with the paths, the whole group is considered for change. Six test cases are considered, each with different number of constraint violations. Fig. 7.12 shows the measurements of the execution times for the overall adjustment and the total change resolution. As the measurements indicate the overall adjustment resolution can detect the unsolvable cases faster and this is because in the overall adjustment each group is tried to be solved independently from the other groups and if a group is unsolvable we can stop the process (because a solution is complete only when all the groups are solved). On the other hand the total change resolution considers all the changes and their respective constraints all together which requires more execution time.

The experiments indicate that despite of the overhead of the propagation scope/path creation, the overall adjustment resolution still outperforms the total change resolution in most of the cases by reducing the execution time and reducing the number of complementary changes and also detects the unsolvable cases faster. However the execution time in some cases is increased if the depth-first incremental propagation or the path bonding methods cannot find a solution and the whole group is required to be considered for change. Therefore the adjustment resolution should be

used with caution in time critical applications and more analytical considerations can make it more suitable for the time-critical systems.

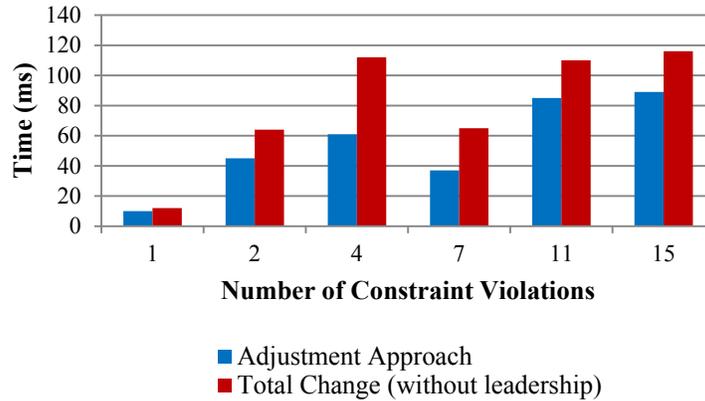


Figure 7.12. Comparison of the execution times of the overall adjustment resolution versus the total change resolution

7.5 Summary

To assure the consistency of the system configuration during a dynamic reconfiguration, the changes need to be checked. Inconsistency often happens because of incomplete reconfiguration changes that need to be completed by adding complementary modifications, i.e. configuration adjustments. Configuration adjustment requires a comprehensive knowledge of the configuration entities/attributes, their relations and system constraints. Not all users (admin or a management application that requests the reconfiguration) have such knowledge or the information may not be exposed to the user e.g. for security concerns. As we explained in Chapter 5, each user manages the system resources using a configuration fragment. However the user is not aware of the relations between the fragments because these relations are only defined in the system configuration (that is generated through the weaving of the fragments). Thus the users may not have a complete set of changes in a change request and for such requests they rely on

the adjustment module to complete the request with respect to the relations of the fragments. Moreover, the complementary modifications of the configuration should be minimized to reduce the time and computational cost of changes and not to destabilize the system at runtime.

Some approaches proposed in the literature for consistency preservation define fixes for constraint violation at design time [59, 60, 61]. Such solutions require extensive effort and can be very complicated and error-prone especially for large systems consisting of many interrelated constraints. Other solutions propose to use pre-defined valid choices for entities for inconsistency resolution but they rely on the users' intervention to select from the choices [26, 57, 58]. Others only focus on satisfying the constraints without concerns about minimization of the changes which can affect the system performance and stability [55, 56].

We proposed a model-based approach for the adjustment of configurations to address the aforementioned challenges. The structure of the configuration - including the entities and their relations - is captured in a configuration profile. The constraints of the configuration are expressed through extended constraints, which also describe the roles of the configuration entities in the constraints. The leader/follower/peer roles define which entity can impact the other ones in the constraint. At runtime a validator detects potentially incomplete changes that violate the configuration constraints. The result of the validation is the input for the adjustment engine. In our adjustment resolution, a propagation scope is identified with respect to any one of the infringing entities of a violated constraint. This scope consists of the entities, and related constraints, that may be affected through change propagation. Different change propagation paths are defined within the scope based on the impact of the entities on each other.

If the propagation scopes are disjoint, each scope can be handled independently following the depth-first incremental propagation method. This let us reduce the side-effects of the change propagation and avoid changing entities unnecessarily. The overlapping propagation scopes need to be solved together assuming that the reconfiguration changes are introduced as a bundle, because of the relation between the changed entities. Thus these scopes are grouped and we try to connect them by bonding the change propagation paths. This means that for each infringing changed entity we select the path which has entities in common with other scopes of its group. Then the paths of the overlapping scopes are considered as a single problem to be solved. The defined problem is then given to a constraint solver to determine the new values that satisfy the constraints.

As we formulate the adjustment by determining the scope and selecting the entities that are required to be modified, we attempt to minimize the modifications in the configuration and the resolution time which are important factors to consider for highly available systems. We implemented a prototype of our work and the results of our prototype evaluation also indicates that the overall adjustment resolution reduces the number of complementary changes and the execution time compared to the total change resolution in which all the entities that are directly and indirectly affected by the initial changes are considered for complementary changes.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this thesis, we presented a model-based framework for configuration management to integrate configuration fragments in a consistent manner at design time and to preserve the consistency of the generated system configuration at runtime by validating and adjusting the configuration after modifications whenever necessary. In our proposed framework configuration profiles are defined (using the UML profiling mechanism) to capture the concepts of each configuration domain, relation and constraints between the concepts. We extended OCL by defining roles for the constrained entities to represent the impact of the entities on each other in a constraint (i.e. the leadership concept). To be valid and consistent a configuration needs to conform to its profile.

In our model-based configuration management framework, we define the relations between the configuration profiles in a weaving model which is then used to generate the system configuration. We extended the weaving technique by adding semantics to capture certain aspects/properties of the system such as availability. We also extract these semantics from the con-

figuration integration process and define them as system constraints. These constraints in addition to the constraints of configuration profiles are used for configuration validation at runtime. We explained our work in the context of the SA Forum middleware and OVF standard; however it is applicable in more general settings.

At runtime the system may be reconfigured to meet certain/new requirements or respond to performance degradations. These changes may lead to the inconsistency of the system configuration as some relations between entities from different fragments may not hold anymore. We proposed a partial validation approach for the validation of the configuration. In our partial validation only a subset of the constraints that are affected by the modifications are selected and checked as the other constraints remain valid. Constraints are also categorized to specify the order in which they should be checked. For evaluating the partial validation approach we semi-formally proved its result is equivalent to the exhaustive validation which checks all the constraints. Our qualitative evaluation demonstrates reduction of the validation time compared to the exhaustive validation. The validation result determines the violated constraints and the entities that cause the inconsistency. In order to correct the potential inconsistencies we proposed an adjustment mechanism to add complementary modifications to the configuration to neutralize the potential inconsistencies. The goal is to minimize the adjustment time and the number of complementary changes because any changes in the configuration need to be applied at runtime on real HA system entities. Thus, the modifications need to be minimal not to destabilize the system. To perform the adjustment we identify the propagation scope for the modifications with respect to the impact of the entities on each other (using the leadership concept). The scope determines which entities can be affected by propagating the changes in the configuration. The scopes which overlap (have

common entities) are grouped and solved together. A final adjustment solution is possible only when all the scopes are solved. We also proposed a depth-first incremental change propagation method to reduce the number of modified entities and avoid affecting other entities unnecessarily. The problem then is formulated as a CSP and a constraint solver is used to find the valid modifications for solving the constraints. The experimental results show that our adjustment solution reduces the execution time and the number of complementary modifications compared to the traditional solution that propagates the changes without considering the leadership concept.

8.2 Future Research

In this section we briefly discuss the research activities which can be considered to complete the work reported in this thesis, or to be considered as future research directions.

8.2.1 The Integration of Configuration Fragments

To integrate the configuration fragments we extended and used model weaving technique to capture the relations among the fragments with respect to specific aspects of the system such as availability. We also discussed that these aspects may conflict or impact each other such as the case of enabling availability and affinity for the integration of AMF and PLM configurations. In our proposed approach such relations are defined once by the configuration designer during the creation of the weaving model and then used for generation of configuration with different input models. To improve the current approach, the detection and resolution of such conflicts can be automated.

We also mentioned that the semantics of the relations between the fragments are captured in the weaving model which is later translated into an executable ATL transformation. However this

translation is now performed by the configuration designer and based on his knowledge of the domain. Automating this process can not only increase the reusability of the weaving links but will also help the verification of the integration process. In the current approach we assume that the created transformations are correct by construction. This is mainly because for verifying the transformations we require the integration constraints which do not exist in the initial configurations and they are only created during the integration process.

We also express the semantics of the relations between the fragments as OCL constraints from the integration process by identifying the derivation tree for the created/modified entities and mapping the operations into OCL expressions. We defined some of these mappings which are used in the context of integrating the SA Forum configurations in Chapter 5. The list of these mappings can be extended to cover a wider range of operations and expressions. In addition there are cases in which more than one mapping is possible and although either of those mappings can result in a valid OCL expression but further analysis can help to select the optimized mappings in different cases.

8.2.2 Partial Validation of the Configuration

In our partial validation approach we select only the impacted constraints by the changes and only validate them as the other ones remain valid. Reduction in the number of constraints results in reduction in validation time which is an important factor in real-time and HA systems. The selected constraints might be applied on many entities of the configuration and thus the constraint should be validated for all those entities either if they are changed or not. A more efficient way is to select the impacted parts of the model by the changes and validate the selected constraints only for the entities of the selected parts of the model. In other words not only to select

the impacted constraints but also to validate them on a projection (view) of the model which contains the entities impacted by the changes.

8.2.3 Runtime Adjustment of the System Configuration

As mentioned in Chapter 7, our goal for adjustment of the configuration at runtime is to reduce the execution time and number of complementary modifications required to keep all the system constraints satisfied. We introduced the depth-first incremental change propagation and path bonding as some heuristics to accomplish this goal. Although we will not miss a solution (if any exists), however it is not guaranteed that we find the best solution with these heuristics and some other heuristics e.g. breadth-first change propagation may outperform our proposed approach in certain cases. The efficiency of such propagation heuristics depends on the current values of the configuration entities and the structure of the configuration model (i.e. the relations and constraints of the entities in the configuration profile). Future work can involve analyzing the performance of different heuristics based on the structure of the configuration and selecting the best propagation solution based on the result of these analyses.

In our adjustment approach we focused on finding the complementary modifications that satisfies the system constraints; however applying these modifications at runtime introduces other challenges. E.g. in what order the changes should be applied in the system to minimize the service outage. Thus, another potential extension to the configuration adjustment approach can be the definition of the change application with respect to the special aspects of the system such as availability.

References

- [1] K. Moazami-Goudarzi, “Consistency preserving dynamic reconfiguration of distributed systems”, PhD thesis, Imperial College London, March 1999.
- [2] J.O. Aagedal, J. Bezivin, and P.F. Linington, “Model-driven development”, In J. Malenfant, and B.M. Ostvold, (Eds.), Object-Oriented Technology. ECOOP 2004 Workshop Reader, LNCS, vol. 3344, Springer-Verlag, pp. 148-157, 2005.
- [3] Object Management Group: Unified Modeling Language – Superstructure Version 2.4.1, formal/2011-08-05, <http://www.omg.org/spec/UML/2.4.1/>.
- [4] L. Fuentes-Fernández, and A. Vallecillo-Moreno, “An introduction to UML profiles”, UPGRADE, The European Journal for the Informatics Professional, vol. 5, no. 2, pp. 5-13, 2004.
- [5] Object Management Group, UML 2.0 OCL Specification, Version 2.4, formal/2014-02-03, 2014, <http://www.omg.org/spec/OCL/2.4/>.
- [6] P.A. Bernstein, “Applying model management to classical meta data problems”, In the 1st Conference on Innovative Data Systems Research (CIDR03), pp. 209-220, USA, 2003.
- [7] Service Availability Forum, <http://www.saforum.org>.
- [8] Atlas Model Weaver (AMW) website, <http://www.eclipse.org/gmt/amw/>.
- [9] Atlas Transformation Language (ATL) website, <http://www.eclipse.org/atl/>.
- [10] Eclipse Modeling Framework, EMF, <http://www.eclipse.org/emf>.
- [11] Service Availability Forum, Application Interface Specification, Availability Management Framework, SAI-AIS-AMF-B.04.01, <http://www.saforum.org/HOA/assn16627/images/SAI-AIS-AMF-B.04.01.pdf>.
- [12] Service Availability Forum, Application Interface Specification, Platform Management Service, SAI-AIS-PLM-A.01.02, <http://www.saforum.org/HOA/assn16627/images/SAI-AIS-PLM-A.01.02.pdf>.
- [13] M. Didonet del Fabro, and P. Valduriez, “Towards the efficient development of model transformations using model weaving and matching transformations”, Journal of Software and Systems Modeling (SoSyM), vol. 8, no. 3, pp. 305–324, 2009.
- [14] M. Didonet del Fabro, J. Bezivin, F. Jouault, and P. Valduriez, “Applying generic model management to data mapping”, In Proc. of the Journées Bases de Données Avancées (BDA05), Saint-Malo, France, 2005.
- [15] Atlantic Modeling group (AtlanMod group) website, http://www.emn.fr/z-info/atlanmod/index.php/Main_Page.

- [16] Institut National de Recherche en Informatique et en Automatique, INRIA, <http://www.inria.fr/en/>.
- [17] I. Warren, J. Sun, S. Krishnamohan, and T. Weerasinghe, “An automated formal approach to managing dynamic reconfiguration”, In Proc. of 21st IEEE/ACM Int. Conference on Automated Software Engineering, pp. 37-46, Washington, USA, 2006.
- [18] A. Jahanbanifar, F. Khendek, and M. Toeroe, “Partial validation of configuration at runtime”, In Proc. of 18th Int. Symposium on Real-Time Distributed Computing (ISORC), pp. 288-291, Auckland, New Zealand, 2015.
- [19] Open Virtualization Format Specification, DMTF Standard, Ver: 2.1.0, December 2013: http://www.dmtf.org/sites/default/files/standards/documents/DSP0243_2.1.0.pdf.
- [20] Distributed Management Task Force: <http://dmtof.org/>.
- [21] Service Availability Forum, Application Interface Specification, Software Management Framework, SAI-AIS-SMF-A.01.02, <http://www.saforum.org/HOA/assn16627/images/SAI-AIS-SMF-A.01.02.AL.pdf>.
- [22] A. Jahanbanifar, F. Khendek, and M. Toeroe, “Runtime adjustment of configuration models for consistency preservation”, In Proc. of 17th Int. Symposium on High Assurance Systems Engineering (HASE), pp. 102-109, Orlando, USA, 2016.
- [23] M. Salehie and L. Tahvildari, “Self-adaptive software: landscape and research challenges”, In ACM Transactions on Autonomous and Adaptive Systems (TAAS), vol. 4, no. 2, pp. 14:1–14:42, 2009.
- [24] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, “Introduction to Algorithms”, Second Edition. MIT Press and McGraw-Hill, 2001.
- [25] Microsoft Research: Z3: a high-performance theorem prover, <http://z3.codeplex.com>.
- [26] H. Song, S. Barrett, A. Clarke, and S. Clarke, “Self-adaptation with end-user preferences: Using run-time models and constraint solving”, In Proc. of 16th Int. Conference on Model-Driven Engineering Languages and Systems (MODELS), pp. 555-571, 2013.
- [27] A. Jahanbanifar, F. Khendek, and M. Toeroe, “A Model-based approach for the integration of configuration fragments”, In Proc. of 11th European Conference on Modelling Foundations and Applications (ECMFA), pp. 125-136, Italy, 2015.
- [28] A. Jahanbanifar, F. Khendek, and M. Toeroe, “Providing hardware redundancy for highly available services in virtualized environments”, In Proc. of 8th Int. Conference on Software Security and Reliability (SERE), pp. 40-47, USA, 2014.
- [29] B. Selic, “A systematic approach to domain-specific language design using UML”, In Proc. of 10th Int. Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), Greece, pp. 2–9, 2007.
- [30] P.C. Clements, and L.M. Northrop, “Software Product Lines: Practices and Patterns”, Addison-Wesley, 2001.

- [31] B. Omelayenko, “A mapping meta-ontology for business integration”, In Proc. of the Workshop on Knowledge Transformation for the Semantic Web at the 15th European Conference on Artificial Intelligence, pp. 76-83, France, 2002.
- [32] A. Maedche, B. Motik, N. Silva, and R. Volz, “MAFRA—AMapping FRamework for Distributed Ontologies”, In Proc. of 13th Int. Conference on Knowledge Engineering and Knowledge Management (EKAW), pp. 235-50, 2002.
- [33] S. Spaccapietra, and C. Parent, “View integration: A step forward in solving structural conflicts”, In IEEE Transactions on Data and Knowledge Engineering, vol. 6, no. 2, pp. 258-274, 1994.
- [34] S. Melnik, E. Rahm, and P. Bernstein, “Rondo: A programming platform for generic model management”, In Proc. of the ACM International Conference on Management of Data (SIGMOD03), pp. 193–204, USA, 2003.
- [35] T. Reiter, E. Kapsammer, W. Retschitzegger, and M. Wimmer, “Model integration through mega operations”, In Proc. of the Int. Workshop on Model-driven Web Engineering (MDWE), Sydney, 2005.
- [36] M. Didonet del Fabro, and P. Valduriez, “Semi-automatic model integration using matching transformations and weaving models”, In Proc. of ACM symposium on Applied computing, pp. 963-970, USA, 2007.
- [37] A. Jossic, M. Del Fabro, J. P. Lerat, J. Bezivin, and F. Jouault, “Model integration with model weaving: A case study in system architecture”, In Proc. of Int. IEEE Conference on Systems Engineering and Modeling, pp. 79–84, Israel, 2007.
- [38] D.D. Ruscio, H. Muccini, A. Pierantonio, and P. Pelliccione, “Towards weaving software architecture models”, In Proc. of 4th Workshop on Model-Based Development of Computer-Based Systems and 3rd Int. Workshop on Model-Based Methodologies for Pervasive and Embedded Software, pp. 103-112, Germany, 2006.
- [39] T. Hinrich, N. Love, C. Petrie, L. Ramshaw, A. Sahai, and S. Singhal, “Using object-oriented constraint satisfaction for automated configuration generation”, In Proc. of 15th IFIP/IEEE Int. Workshop on Distributed Systems Operations and Management, LNCS vol. 3278, Springer, pp. 159-170, Davis, USA, 2004.
- [40] A. Sahai, S. Singhal, R. Joshi, and V. Machiraju, “Automated generation of resource configurations through policies”, In Proc. of 5th IEEE Int. Workshop on Policies for Distributed Systems and Networks(POLICY2004), pp. 107-110, 2004.
- [41] G. Friedrich, and M. Stumptner, “Consistency-based configuration”, In AAAI'99 Workshop on Configuration, AAAI Press Technical Report, WS-99-05, pp. 35–40, Orlando, USA, 1999.
- [42] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft, “The SmartFrog configuration management framework”, In ACM Journal of SIGOPS Operating System Rev., vol. 43, no. 1, pp. 16–25, 2009.

- [43] M. Burgess and A.L. Couch, “Modeling next generation configuration management tools”, In Proc. of the 20th Conference on Large Installation System Administration (LISA), pp. 131–147, Washington, USA, 2006.
- [44] L. Akue, E. Lavinal, T. Desprats, and M. Sibilla, “Runtime configuration validation for self-configurable systems” , In Proc. of IFIP/IEEE Int. Symposium on Integrated Network Management (IM 2013), pp. 712–715, Ghent, Belgium, 2013.
- [45] L. Akue, E. Lavinal, T. Desprats, and M. Sibilla, “Integrating an online configuration checker with existing management systems: application to cim/wbem environments”, In Proc. of 9th Int. Conference on Network and Service Management (CNSM), pp. 339-344, Zurich, Switzerland, 2013.
- [46] J. Cabot, and E. Teniente, “Incremental integrity checking of UML/OCL conceptual schemas”, In Journal of Systems and Software, vol. 82, no. 9, pp. 1459–1478, 2009.
- [47] J. Cabot, and E. Teniente, ”Computing the relevant instances that may violate an OCL constraint”, In Proc. of 17th Int. Conference on Advanced Information Systems Engineering (CAiSE’05), O. Pastor, J. Falcão e Cunha, (eds.), LNCS, vol. 3520, pp. 48–62, Springer, Heidelberg, 2005.
- [48] I. Groher, A. Reder, and A. Egyed, “Incremental consistency checking of dynamic constraints”, In Proc. of Fundamental Approaches to Software Engineering (FASE 2009), D.S. Rosenblum, G. Taentzer, (eds.). LNCS, vol. 6013, pp. 203–217, Springer, Heidelberg, 2010.
- [49] G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Ökrös, “Incremental evaluation of model queries over EMF models”, In Proc. of 13th Int. Conference on Model Driven Engineering Languages and Systems: Part I, pp. 76–90, Berlin, Springer-Verlag, Heidelberg, 2010.
- [50] A. Reder, and A. Egyed, “Incremental consistency checking for complex design rules and larger model changes”, In Proc. of the 15th Int. Conference on Model Driven Engineering Languages and Systems (MODELS), Berlin, Springer-Verlag, Heidelberg, vol. 7590, pp. 202-218, 2012.
- [51] A. Khalil and J. Dingel, “Supporting the evolution of UML models in model driven software development: A survey”, Technical Report, School of Computing, Queen’s University, Canada, 2013.
- [52] A. Egyed, “Consistent adaptation and evolution of class diagrams during refinement”, In Proc. of 7th Int. Conference on Fundamental Approaches to Software Engineering, LNCS, vol. 2984, Springer, Berlin, Heidelberg, pp. 37-53, Barcelona, Spain, 2005.
- [53] J.O. Kephart, “Research challenges of autonomic computing”, In Proc. of 27th Int. Conference on Software Engineering (ICSE 2005), pp. 15-22, 2005.
- [54] S. Neema, and A. Ledeczi, “Constraint-guided self-adaptation”, In Proc. of 2nd Int. Conference on Self-adaptive Software: Applications, LNCS, vol. 2614, pp. 39-51, 2003.

- [55] P. Sawyer, R. Mazo, D. Diaz, C. Salinesi, and D. Hughes, “Using constraint programming to manage configurations in self-adaptive systems”, *IEEE Computer*, vol. 45, no. 10, pp. 56-63, 2012.
- [56] J. White, B. Dougherty, D. C. Schmidt, and D. Benavides, “Automated reasoning for multi-step feature model configuration problems”, In *Proc. of the Software Product Line Conference (SPLC)*, pp. 11-20, San Francisco, USA, 2009.
- [57] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, “Generating range fixes for software configuration”, In *Proc. of the 34th Int. Conference on Software Engineering (ICSE)*, pp. 58-68, Piscataway, USA, 2012.
- [58] V. Poladian, J.P. Sousa, D. Garlan, and M. Shaw, “Dynamic configuration of resource-aware services”, In *Proc. of 26th Int. Conference on Software Engineering (ICSE)*, pp. 604-613, 2004.
- [59] A. Egyed, E. Letier, and A. Finkelstein, “Generating and evaluating choices for fixing inconsistencies in UML design models”, In *Proc. of the 23rd IEEE/ACM Int. Conference on Automated Software Engineering (ASE '08)*, pp. 99–108, 2008.
- [60] C. Nentwich, W. Emmerich, and A. Finkelstein, “Consistency management with repair actions”, In *Proc. of 25th Int. Conference on Software Engineering (ICSE)*, pp. 455–464, 2003.
- [61] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei, “Supporting automatic model inconsistency fixing”, In *Proc. of 7th Joint Meeting of the European Software Eng. Conference and the ACM SIGSOFT Symposium on the foundations of software engineering (ESEC/FSE '09)*, pp. 315-324, New York, USA, 2009.
- [62] J. Bézivin, and F. Jouault, “Using ATL for checking models”, In *Proc. of the 4th Int. Workshop on Graph and Model Transformation (GraMoT 2005)*, *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 69–81, 2006.
- [63] M. Didonet del Fabro, J. Bézivin, and P.Valduriez, “Weaving models with the eclipse AMW plugin”, In *Proc. of the Eclipse Modeling Symposium, Eclipse Summit Europe*, 2006.
- [64] M. Didonet del Fabro, J. Bézivin, F. Jouault, E. Breton, and G. Gueltas, “AMW: A generic model weaver”, In *the 1^{ère} Journée sur l’Ingénierie Dirigée par les Modèles*, 2005.
- [65] P. Salehi, A. Hamoud-Lhadj, P. Colombo, F. Khendek, and M. Toeroe, “A UML-based domain specific modeling language for the availability management framework”, In *Proc. of 12th IEEE Int. Conference on High Assurance Systems Engineering Symposium (HASE)*, pp. 35–44, San Jose, USA, 2010.
- [66] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, “On the use of higher-order model transformations”, In *Proc. of the 5th European Conference on Model Driven Architecture -Foundations and Applications (ECMDA-FA 2009)*, R.F. Paige, A. Hartman, A. Rensink, (eds.), LNCS, Springer, Berlin, vol. 5562, pp. 18-33, 2009.
- [67] E. Tsang, “Foundations of Constraint Satisfaction”, Books on Demand, 2014.
- [68] V. Kumar, “Algorithms for constraint satisfaction problems: A survey”, *AI Magazine* vol.13, no.1, pp. 32-44, 1992.

- [69] L. de Moura and N. Bjorner, “Model-based theory combination”, In Proc. of the 5th workshop on Satisfiability Modulo Theories (SMT’07), pp. 37–49, 2007.
- [70] G. Nelson, and D.C. Oppen, “Simplification by cooperating decision procedures”, ACM Transactions on Programming Languages and Systems, vol. 1, no.2, pp. 245–257, 1979.
- [71] A. Jahanbanifar, F. Khendek, and M. Toeroe, “Semantic weaving of configuration fragments into a consistent system configuration”, Submitted to the Information Systems Frontiers Journal in December 2015.
- [72] A. Jahanbanifar, F. Khendek, and M. Toeroe, “A runtime change propagation and adjustment approach for configuration models”, Submitted to the TAAS ACM Journal in April 2016.
- [73] A. Jahanbanifar, and M. Toeroe, “Defining disjoint node groups for virtual machines with pre-existing placement policies”, US patent, filed April 2013.
- [74] A. Jahanbanifar, M. Toeroe, and F. Khendek, “Generating consistent system configuration by model weaving”, US patent, filed April 2015.
- [75] A. Jahanbanifar, M. Toeroe, and F. Khendek, “Partial validation of configurations at runtime”, US patent, filed October 2015.
- [76] A. Jahanbanifar, M. Toeroe, and F. Khendek, “Adjustment of configuration models for consistency preservation”, US patent, filed September 2015.
- [77] A. Jahanbanifar, M. Toeroe, and F. Khendek, “Method of deriving system constraints from configuration integration rules”, US patent, filed December 2015.

Appendix A: An Excerpt of the Higher Order Transformation (HOT) for the Generation of System Configuration

This appendix provides part of the code used in the HOT (i.e. WModel2Final) for the generation of the final transformation used to create the system (introduced in Chapter 5).

```
module HOT3MM;

create OUT : ATL from IN : AMW, sourceModel1: MOF, sourceModel2: MOF, target-
Model: MOF ;

helper def : model1 : String = 'Model1';
helper def : metamodel1 : String = 'Profile1';

helper def : model2 : String = 'Model2';
helper def : metamodel2 : String = 'Profile2';

helper def : targetM : String = 'SystemModel';
helper def : targetMM : String = 'SystemProfile';

helper def : getSourceModel1Instance(classifierID : String) :
MOF!EModelElement =
    thisModule.getInstanceById('sourceModel1',classifierID);

helper def : getSourceModel2Instance(classifierID : String) :
MOF!EModelElement =
    thisModule.getInstanceById('sourceModel2',classifierID);

helper def : getTargetModelInstance(classifierID : String) :
MOF!EModelElement =
    thisModule.getInstanceById('targetModel',classifierID);

rule Module {
    from
        amw : AMW!Magic
    to
        atl : ATL!Module(
            isRefining <- false,
            name <- if(amw.name.ocliIsUndefined()) then 'ModelTransform'
                    else amw.name
                    endif,
            inModels <- Set {amw.sourceModel1, amw.sourceModel2},
            outModels <- amw.targetModel,
            elements <- Set{amw.correspondences},)
}
```

```

rule SourceModel1Ref {
  from
    amw : AMW!MagicModelRef (amw.name = 'sourceModel1')
  to
    atl : ATL!OclModel (
      metamodel <- ametamodel,
      name <- thisModule.model1),

    ametamodel : ATL!OclModel (
      name <- thisModule.metamodel1,
      elements <-
        MOF!EClassifier.allInstancesFrom('sourceModel1')
      )
    }

rule SourceModel2Ref {
  from
    amw : AMW!MagicModelRef (amw.name = 'sourceModel2')
  to
    atl : ATL!OclModel (
      metamodel <- ametamodel,
      name <- thisModule.model2),

    ametamodel : ATL!OclModel (
      name <- thisModule.metamodel2,
      elements <- MOF!EClassifier.allInstancesFrom('sourceModel2')
      )
    }

rule TargetModelRef {
  from
    amw : AMW!MagicModelRef (amw.name = 'targetModel')
  to
    atl : ATL!OclModel (
      metamodel <- ametamodel,
      name <- thisModule.targetM
      ),

    ametamodel : ATL!OclModel (
      name <- thisModule.targetMM,
      elements <- MOF!EClassifier.allInstancesFrom('targetModel')
      )
    }

rule CreateOutModel() {
  to
    target : ATL!MatchedRule (
      name <- 'modelRule',
      isAbstract <- false,
      isRefining <- false,
      inPattern <- modelInPattern,
      outPattern <- modelOutPattern,
      actionBlock <-doSection ),

```

```

modelInPattern : ATL!InPattern(
  elements <- element ),

element : ATL!SimpleInPatternElement(
  id <- 'source_' ,
  varName <- 'source',
  type <- aType
  ),

aType : ATL!OclModelElement (
  name <- 'Model',
  model <- ATL!OclModel.allInstances()->select ( e | e.name =
  thisModule.metamodel2)->first() ),

modelOutPattern : ATL!OutPattern(
  elements <- elementOut
  ),

elementOut : ATL!SimpleOutPatternElement(
  id <- 'source2_' ,
  varName <- 'target',
  type <- aType2,
  bindings <- aBinding4Model
  ),

aType2 : ATL!OclModelElement (
  name <- 'Model',
  model <- ATL!OclModel.allInstances()->select ( e | e.name =
  thisModule.targetMM)->first() ),

aBinding4Model: ATL!Binding (
  propertyName <- 'name',
  value <- stringExp
  ),

stringExp : ATL!StringExp (
  stringSymbol <- 'output'),

doSection : ATL!ActionBlock(
  statements <- st1
  ),

st1: ATL! ExpressionStat(
  expression <- value1 ),

value1 : ATL!OperationCallExp(
  operationName <- 'applyProfile',
  arguments <- argument1,
  source <- valSource
  ),

valSource : ATL!VariableExp(
  referredVariable <- elementOut ),

argument1 : ATL!CollectionOperationCallExp(
  operationName <- 'first',
  source <- selectCall),

selectCall : ATL!IteratorExp(
  name <- 'select',

```

```

        source <- allInstanceCall,
        body <- selectBody,
        iterators <- Sequence{iterator}),

selectBody : ATL!OperatorCallExp(
    operationName <- '=',
    arguments <- Sequence{argument3},
    source <- iteratorVar    ),

iterator : ATL!Iterator(
    varName <- 's',
    id <- 's2'    ),

iteratorVar : ATL!NavigationOrAttributeCallExp(
    name<-'name',
    source<- valSource4    ),

valSource4 : ATL!VariableExp(
    referredVariable <- iterator),

argument3 : ATL!StringExp(
    stringSymbol <- thisModule.targetMM),

allInstanceCall:ATL!OperationCallExp(
    operationName <- 'allInstancesFrom',
    source <- value2,
    arguments<- argument4),

argument4: ATL!StringExp(
    stringSymbol <- thisModule.targetMM ),

value2: ATL!OclModelElement (
    name <- 'Profile',
    model <- ATL!OclModel.allInstances()->select ( e | e.name =
    thisModule.targetMM)->first()
    )
}

```

```

rule equalCorrespondence {
  from
    amw : AMW!EqualCorrespondence
  to
    atl : ATL!MatchedRule (
      name <- amw.name
      isAbstract <- amw.isAbstract,
      isRefining <- false,
      inPattern <- (if (not amw.sourceModel1.oclIsUndefined())
        then
          amw.sourceModel1
        else
          amw.sourceModel2
        endif
      ),

```

```

        outPattern <- amw.targetModel,
        actionBlock <- doSection
                        ),
        doSection : ATL!ActionBlock()
    }

rule equalElementSourceModell {
  from
    amw : AMW!EqualElement(not thisMod-
ule.getSourceModellInstance(amw.element.ref).oclIsUndefined())
  to
    atl : ATL!InPattern(
      elements <- element,
      filter<- aFilter ),

    element : ATL!SimpleInPatternElement(
      varName <- amw.variableName,
      type <- aType ),

    aType : ATL!OclModelElement (
      name <- thisMod-
ule.getSourceModellInstance(amw.element.ref).ownedElement-
>first().type.name,
      model <- ATL!OclModel.allInstances()->select ( e | e.name =
thisModule.metamodel1)->first()
    ),

    aFilter: ATL!OperationCallExp(
      operationName <- 'isStereotypeApplied',
      arguments <- arg1,
      source <- valSource
    ),

    valSource : ATL!VariableExp(
      referredVariable <- thisMod-
ule.resolveTemp(amw.refImmediateComposite().sourceModel2, 'e-
lement') ),

    arg1 :ATL!StringExp (
      stringSymbol <- thisModule.metamodel1 + '::' + thisMod-
ule.getSourceModellInstance(amw.element.ref).name
    )
}

```