Generic and Adaptable Online Configuration Verification for Complex Networked Systems

Ludi Akue, Emmanuel Lavinal, Thierry Desprats, and Michelle Sibilla
IRIT, Université de Toulouse
118 route de Narbonne
F31062 Toulouse, France
Email: {akue, lavinal, desprats, sibilla}@irit.fr

***Abstract*—Dynamic reconfiguration is viewed as a promising solution for today's complex networked systems. However, considering the critical missions actual systems support, systematic dynamic reconfiguration cannot be achieved unless the accuracy and the safety of reconfiguration activities are guaranteed. In this paper, we describe a model-based approach for runtime configuration verification. Our approach uses model-driven engineering techniques to implement a platform-independent online configuration verification framework that can operate as a lightweight extension for networked systems management solutions. The framework includes a flexible and adaptable runtime verification service built upon a high-level language dedicated to the rigorous specification of configuration models and constraints guarding structural correctness and service behavior conformance. Experimental results with a real-life messaging platform show viable overhead demonstrating the feasibility of our approach.**

***Keywords*—Network and Service Management; dynamic reconfiguration; configuration verification; online verification; model-based approach.**

I. Introduction

Complex networked systems and services are a fundamental basis of today's life. They increasingly support critical services and usages, essential both to businesses and the society at large. The evident example is the Internet with all its services and usages in a variety of forms, architectures and media ranging from small mobile devices such as smartphones to large-scale critical systems such as clusters of servers and cloud infrastructures. Consequently, it is indispensable to ensure their proper and continuous operation.

Network and Service Management (NSM) is a research and technical discipline that deals with models, methods and techniques to ensure that managed networked systems and services operate optimally according to a given quality of service. To cope with the increasing complexity of managed systems, NSM has evolved into self-management, a vision that consists in endowing management solutions with a high degree of autonomy to allow them to dynamically and continuously reconfigure managed systems in order to maintain a desired state of operation in the face of unstable and unpredictable operational conditions.

A main obstacle to the diffusion of dynamic reconfiguration solutions is the lack of standard methods and means to ensure the effectiveness of subsequent configuration changes and prevent erroneous behaviors from compromising the system's operation. This issue is particularly significant in today's mission critical systems management like cloud infrastructures, avionics, healthcare systems or mobile multimedia networks. This will also help increase users' confidence in the automation of reconfiguration, thus ease the adoption of ongoing autonomic solutions.

This article extends our recent work [1] on a model-based approach for online configuration verification with a running prototype architecture. It also provides additional concepts, methods and tools forming an online configuration verification framework. In particular, we describe how we use the framework to enrich a management system for a message oriented middleware platform with online configuration checking capabilities. Following the same process, the verification framework could be integrated with other existing management solutions.

Our approach to build this framework was first to define MeCSV (Metamodel for Configuration Specification and Validation), a high-level language, dedicated to the specification and verification of configurations. MeCSV allows operators to specify at design time, a platform-neutral configuration schema of their managed system with constraints guarding the desired service architecture and operation. One novelty of the metamodel is to include the capability to express both *offline* and *online* constraints. Offline constraints are typically structural integrity rules, that is, rules that govern a system's configuration structure. Online constraints concern service operation, they consist of rules to be enforced with regards to runtime conditions to avoid committing inconsistent configurations. An earlier version of the metamodel has been presented in [1].

Second, we have designed a runtime verification service, able to manipulate the concepts defined in this language. This service offers two interfaces, a verification interface for invoking configuration verification and an edition interface for managing specifications at runtime. The verification interface is flexible as it provides different operations to tailor configuration verifications to the usage scenarios, e.g., verifying only a subset of constraints regarding their severity or importance. The edition interface enables constraints updates at runtime to cope with changing management requirements.

These two phases allow our framework to support a verification process that starts at design time with a rigorous specification of verification models and continues at runtime through an automatic checking of configurations based on these models.

The rest of the paper is organized as follows: Section

II identifies runtime configuration verification requirements and positions existing works. We give an overview of the framework in Section III, and through a case study included in Section IV, elaborate on its building blocks in Section V and Section VI. We describe the integration of the framework with a real-life messaging platform in Section VII along with experimental results, proving the feasibility of the approach. Section VIII concludes the paper and identifies future work.

II. Background and Related Work

We begin this section by introducing the terminology used throughout the paper, then we expose the runtime configuration verification requirements in the current context of autonomous management approaches like in [2] and [3]. Finally, we present how those requirements have been addressed in related works.

A. Terminology

This section recalls definitions of key terms used throughout the paper.

1) Configuration: A configuration of a system is a collection of specific functional and non-functional parameters (also known as configuration parameters or configuration data) whose values determine the expected functionalities that the system should deliver.

2) Execution context: The execution context of a system comprises every element that can influence the system's operation. It includes both the system's technical environments, i.e., its interactions with other systems, its supported services and usages and the users' expectations, i.e., management and service objectives.

3) Operational state: The operational state of a system qualifies its observed operation in terms of the current values of its state parameters (or state data). The operational state is issued by a monitoring or a supervisory system. It reflects the behavior of the system relevant to the context at hand.

4) Dynamic Reconfiguration: Reconfiguration is the modification of an existing and already deployed configuration. Reconfigurations can be static, that is configurations are modified offline, when the system is not running. They can also be dynamic, that is configurations are modified online, while the system is running. We are especially interested in managed systems that are reconfigured dynamically.

5) Configuration verification: Configuration verification is the process of examining configuration instances against a set of defined requirements according to system architecture and service objectives. Configuration verification checks the correctness of proposed configuration instances, thus detect misconfigurations prior to changing the productive system.

B. Configuration Verification Requirements

Verification has always been critical to check that a given configuration meets its functional as well as non-functional requirements. When considering the lifecycle of a configurable system, in contrary to software verification that occurs mainly during the development phase of a system, configuration verification rather occurs in the use phase of a system to enforce its operation and maintenance (Fig. 1).

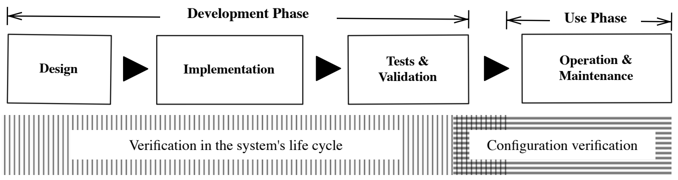


Fig. 1. Configuration verification in a simplified system's lifecycle

Configuration verification is traditionally done offline, either at design time, for example in test environments, or in production environments to enforce static reconfigurations. This verification is limited to structural sanity checks, typically testing the correct structure and composition of configuration parameters in terms of authorized values, consistent cross-components dependencies and syntactical correctness. This type of verification involves configuration data only. We have called it *structural integrity verification* [4].

By definition, dynamic reconfiguration implies configuration verifications should be carried out automatically at runtime for checking live configuration changes. Therefore, the verification process should take into account running operational conditions (1). It should also be flexible and adaptable to cope with the changing execution context in terms of architectural dynamics and changes of service objectives (2). Besides, it should accommodate the heterogeneity of management domains, representations and tools (3).

1) Operational Applicability Verification: Configuration verification should go beyond structural checks to assess the operational applicability of proposed configurations regarding runtime conditions at hand.

As systems adapt dynamically, ongoing operational states can invalidate the suitability of a produced configuration despite its structural correctness. For instance, one of the most common causes of a failed live virtual machine migration is not checking that the current physical resources of the destination host are sufficient before performing the live migration.

A runtime configuration verification should include *operational applicability verification*, that is, checking if a proposed configuration fulfills some running conditions [4]. This type of verification requires the knowledge of monitored operational state data. In other words, a runtime configuration verification should cover both *structural integrity verification* and *operational applicability verification.*

2) Flexible and Adaptable Verification: Configuration verification should be flexible and adaptable to cope with the changing execution context in terms of architectural dynamics and changes of service objectives.

The execution context of actual complex systems is highly dynamic in terms of operational conditions variations and dynamic usages where they are added, removed, migrated according to changing management and service objectives. This dynamicity implies configuration changes of different spatial and temporal scopes, e.g., local changes to system-wide changes, planned or spontaneous changes, punctual or life-long changes [5], [6].

A runtime configuration verification solution should thus accommodate these new characteristics by being flexible and adaptable in order to be tailored to reconfiguration needs and usage scenarios, e.g., adapting the verification scope and perimeter.

3) Platform-independent Verification: Configuration verification should be platform-neutral to accommodate the heterogeneity of management domains, representations and tools.

Management applications domains are highly heterogenous in terms of the nature and importance of configuration data, (e.g., configuration data can be functional, non-functional, static or dynamic), their different representations due to different management standards and protocols, as well as the diverse nature of properties to be checked (e.g., hard constraints, soft constraint) according to diverse usage scenarios (mobility, performance, functional, non-functional) [7], [8].

In consequence, a configuration verification solution should be high-level and capable to integrate this heterogeneity.

C. Related Work

This section discusses existing works regarding the identified configuration verification requirements.

The need for configuration representation standards and configuration automation are growing concerns regarding the complexity of the configuration management of today's large-scale systems [9], [7]. Our work is at the junction of these two topics as the verification framework we provide enables a platform independent online configuration verification that is a prerequisite for configuration automation as well as dynamic reconfiguration.

Existing management standards like the Distributed Management Task Force Common Information Model (CIM) [10] and the YANG data modeling language [11] include constructs for configuration verification, yet their enforcement is left to implementors and solutions developers. Furthermore, those standards do not propose any mechanism for flexible and adaptable configuration verification as well as the management of the resulting verification lifecycle.

Beyond management standards, related works can fall into two groups: constraint checking approaches [12], [13], [14], [15] and valid configuration generation approaches [16], [17], [18].

In the first group, configuration experts are given a specification language to express some constraints that a verification engine checks on proposed configuration instances. In the second group, configuration decision is modeled as a Constraint Satisfaction Problem, adequate SAT solvers generate valid configurations or prove insatisfiability.

Both groups present common limitations, they do not address online configuration checking with regards to operational conditions: they provide only structural integrity verification that relies purely on configuration data or confines configuration verification to design time. They do not support a flexible and adaptable verification (e.g., only check a subset of constraints, modify and manage constraints during the reconfiguration life cycle). In addition, they mainly propose domain-specific tools with use-case specific verification (networks, distributed applications, JAVA applications, virtual machines).

Our work in contrast proposes a generic configuration verification approach that targets specifically online configuration checking, considering the influence of ongoing execution conditions on the verification process. It is thus profitable for complementing existing verification approaches.

In particularly, our work shares common foundations with SANChk [14], a SAN (Software Area Networks) configuration verification tool. They both use formal constraint checking techniques and enable a flexible and adaptable configuration verification. However, SANChk is specific to SAN configurations and does not target online configuration verification.

III. Online Configuration verification Framework

This section presents our verification approach and subsequent assumptions and concepts.

A. Assumptions

The following assumptions characterize the class of managed systems that we currently consider:

- The system is supposed known, observable, it is under a supervisory control that collects measures about its operating states and environment.
- The system is dynamically configurable, that is to say its current configuration can be altered at runtime if needed.
- The system's execution context is highly dynamic, hence is subject to sudden and often unpredictable variations.
- Either the supported management goals are clearly specified in order to derive properties to validate, or these properties are already defined.

B. Vision and Design Principles

The verification framework aims at offering an online configuration checking service that can be used by management solutions without changing existing tools. It specifically targets current autonomous and self-management approaches. As such, it purposefully addresses the runtime management of the systems' dynamics and the rapidly changing service and architecture requirements. The framework supports these new requirements through three main design principles according to the requirements exposed in Section II:

- Enabling an operational verification of configurations that takes into account their dependency on running execution states: in the context of self-adaptation, configurations are highly dependent on the operational conditions that can invalidate the suitability of a candidate configuration.
- Allowing modification of validity properties at runtime: management systems are likely to have their requirements evolve at runtime, and these evolutions are to be translated at runtime into the creation or modification of properties on configurations.

- Supporting existing management systems in order to enhance their reliability with a verification functionality. This can notably be achieved by integrating existing management standards such as CIM and YANG.

C. Integration within the Management Control Loop

Self-management is generally performed through a control loop called the MAPE (Monitor-Analyze-Plan-Execute) loop: the managed system is monitored (Monitor) to produce metrics that are analyzed (Analyze) to detect or prevent any undesirable behavior. Corrective changes are then planned (Plan) – either in the form of a new configuration or as a sequence of actions – then effected on the system (Execute) [2].

Runtime configurations decision is normally the responsibility of the *Plan* function. Consequently, a runtime configuration verification function that handles the two types of configuration verification is worthy to extend the MAPE loop to assure the validity of proposed configurations.

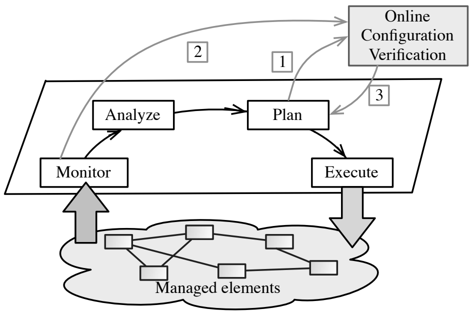


Fig. 2. The vision of online configuration checking

Fig. 2 illustrates our vision of a standard online configuration checking. An external and protocol-independent verification solution interacts with a management system (represented through the MAPE loop) through the Plan block (1), center of runtime configuration decisions, while relying on the Monitor block (2) for the retrieval of the required ongoing execution states, and thus processes an online verification of configurations (3).

As a result, the verification solution we propose, can extend any self-configurable system that does not have built-in online configuration verification. The only requirement for the self-configuring system is to allow access to its configurations and monitored data at runtime. Furthermore, this verification solution can be independently tuned and managed giving ongoing usage needs.

D. Overview of the Framework's Building Blocks

This section describes the building blocks of our verification framework. The framework supports a model-based approach for runtime configuration verification relying on a high-level specification language MeCSV and a verification service able to manipulate the concepts defined in this language.

1) MeCSV language: MeCSV is a metamodel dedicated to the formal modeling of configuration information for runtime verification. It offers platform-neutral configuration specification constructs including innovative features that enable

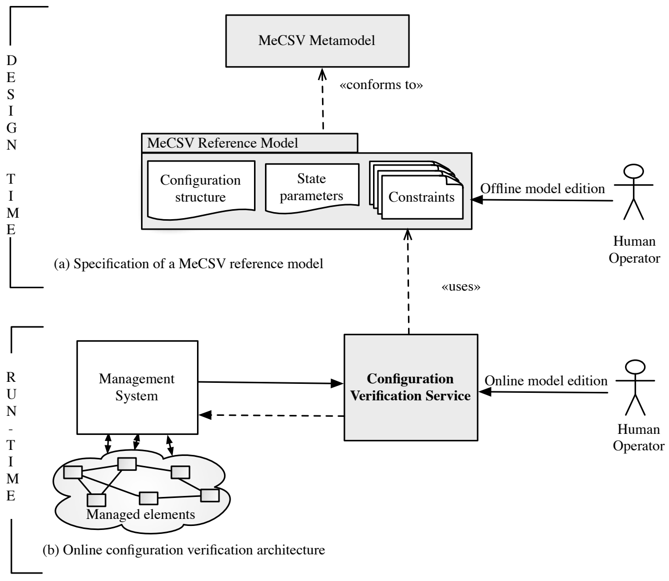


Fig. 3. Framework's approach and architecture

verification against runtime execution conditions. Even though MeCSV allows to model a system's configuration, it is intended for verification purposes only and is not suitable neither to model exhaustive management information nor to handle the management of the configuration's lifecycle (configuration data stores, configuration deployment etc.).

2) Target Domain Reference Model: The central objective of MeCSV is to allow the definition of a *Reference Model* that every possible configuration of the target system should conform to (Fig. 3 (a)). Operators or vendors can thus use the MeCSV metamodel at design time to define the reference model of a given managed application domain (e.g., an application server, a messaging middleware). Defined reference models are instances of the MeCSV metamodel, they are dedicated to a target application domain but do not rely on platform-specific representations. This reference model is to be defined only once, it will be processed at each reconfiguration decision, to dynamically evaluate configuration instances.

3) Verification Service: The framework includes a runtime architecture, Fig. 3 (b), with a verification service. This service needs to be initialized with the defined reference model. A related management system assuring monitoring and reconfiguration capabilities can then invoke specific operations at runtime to perform online verifications of decided configuration instances. This verification service also supports online modification of reference models to cope with the evolution of management and system requirements.

E. General Life Cycle of the Framework

The framework's building blocks support the following verification process:

1) At design time: A human operator, (e.g., an administrator or a configuration expert) uses the MeCSV metamodel to formally specify the *MeCSV Reference Model* of a given application domain (cf. Fig. 3 (a)). This reference model is made of a configuration schema of the domain, a relevant set of operational state parameters to monitor and the offline and online constraints, necessary to enforce the structural integrity and operational applicability of configuration instances.

2) At runtime: this reference model will be used by the Verification Service for the automatic evaluation of proposed configuration instances (cf. Fig. 3 (b)). This online verification can evolve along with management objectives as the deployed reference model can be updated by human operators.

These two phases will be detailed subsequently in Sections V and VI, respectively.

IV. Use Case

This section illustrates a Message-oriented Middleware (MOM) case study on which the examples given in the following sections will be based.

A. Introduction to MOM

A MOM system is a specific class of middleware that supports loosely-coupled communications among distributed applications via asynchronous message passing, as opposed to a request/response metaphor. They are at the core of a vast majority of financial services. Client applications interact through a series of servers where messages are forwarded, filtered and exchanged.

The middleware's operation involves the proper configuration of numerous elements such as message servers, message destinations and directory services. Involved configuration and reconfiguration tasks can be classified into two categories: first, setup operations that include defining the number of servers, where they will run and the messaging services each will provide. Second, maintenance operations that use the platform's monitoring metrics to adjust initial setups such as memory resources, message thresholds and users access.

By adding a management interface, an operator can monitor and tune the system's performance, reliability and scalability according to the monitored metrics (e.g., memory resources and users access) and management objectives.

B. Joram's platform

Joram (Java Open Reliable Asynchronous Messaging) [19] is an open source MOM implementation in Java. Joram provides access to a MOM platform that can be dynamically managed and adapted, i.e., monitored and configured for the purpose of performance, reliability and scalability thanks to JMX (Java Management eXtensions) management interfaces.

Principal managed elements are message servers that offer the messaging functionalities such as connection services and message routing and message destinations that are physical storages supporting either queue-based (i.e., point-to-point) or topic-based (i.e., publish/subscribe) communications.

A Joram platform can be configured in a centralized fashion where the platform is made of a single message server and a distributed fashion, the platform is made of two or more servers running on given hosts. A Joram platform can be dynamically reconfigured, message servers can be added and removed at runtime. A platform configuration is described by an XML configuration file according to a provided DTD (Document Type Definition).

Fig. 4 shows a centralized configuration example, that will be further experimented in Section VII (Test Case 1). This configuration is made of one server, several middleware services (connection manager, naming service, etc.), two message queues and a user's permissions (note that due to space limitations, some configuration elements have been discarded).

```
<?xml version="1.0"?>
<config name="Simple_Config">
 <server id="0" name="S0" hostname="localhost">
  <service class="org.[…].ConnectionManager" args="root root"/>
  <service class="org.[…].TcpProxyService" args="16010"/>
  <service class="fr.[…].JndiServer" args="16400"/>
 </server>
</config>
<JoramAdmin>
  <InitialContext> […] </InitialContext>
  <ConnectionFactory> […] </ConnectionFactory>
  <Queue name="myQueue" serverId= "0"
         nbMaxMsg="200" dmq="dmqueue">
    <freeReader/> <freeWriter/>
    <jndi name="myQueue"/>
  </Queue>
  <User name="anonymous" password="passwd" serverId="0"/>
  <DMQueue name="dmqueue" serverId = "0">
     <freeReader /><freeWriter />
  </DMQueue>
</JoramAdmin>
```

Fig. 4. A Joram's configuration example

C. Verification Requirements

The following requirements are considered for the purpose of the case study, they encompass the manufacturer's set of configuration constraints and custom configuration constraints that guarantee memory performance. A valid configuration of the platform should provide the necessary messaging features in order for client applications to communicate efficiently. More precisely,

- Correct configuration structure: it should respect the platform's architecture and the relationships between the configuration parameters. (`Req1`)
- Object discovery and lookup: connection factories and destinations should be accessible via a naming service, i.e., the platform should provide an accessible JNDI service where the reference of the administered objects should be stored. (`Req2`)
- Memory optimization: message queues should not run low in memory, i.e., queues should not be loaded at more than 80% of their maximum capacity. (`Req3`)

V. MeCSV Metamodel

This section presents the salient features of the metamodel depicted in Fig. 5. MeCSV is organized in three categories of constructors: the first category is dedicated to configuration description, the second to operational state data description and the last for constraint expression.

A. Configuration Data Description

This part of the metamodel, depicted in Fig. 5 - Configuration, represents concepts to describe configuration data.

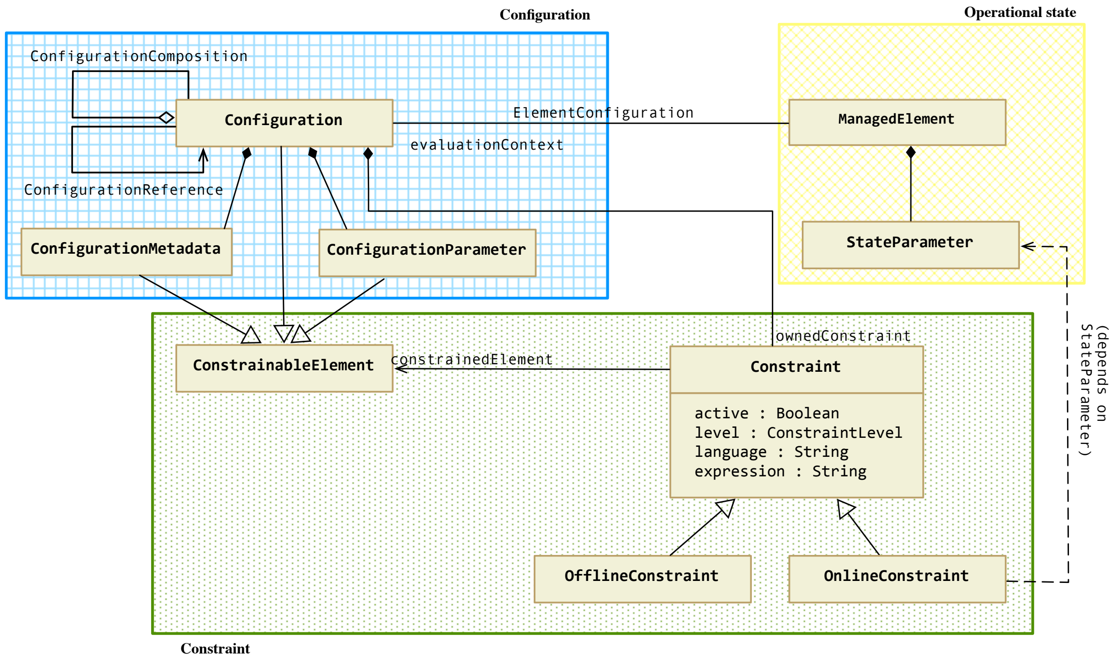


Fig. 5. Overview of the MeCSV metamodel

Configuration data are generally described in a set of configuration files where their structure is specified through the setting of some configuration properties with appropriate values and options.

There are a lot of bindings between the system's elements that should be reflected in their configuration: for example, the dependency of a server on its host machine should be specified by the coordination of the server's hostname value with the machine's hostname value.

These needs are addressed by the following constructs that are protocol and tool independent.

1) Configuration Parameter: represents quantifiable configuration parameters of managed elements; their expression defines the configuration data structure. A message server's identifier or hostname information are examples of its *configuration parameters*.

2) Configuration: allows to coordinate configuration parameters and group them in categories. Configuration elements act as containers for configuration parameters. For example, a configuration file can be modeled as a single *Configuration*, or for more flexibility, divided into multiple *Configurations*.

3) Configuration Dependency: to model bindings between two configuration elements, a *configuration dependency* should be defined between them. It means that a configuration parameter of some configuration references a whole or a part of another configuration. Typically, a server's hostname references its host machine's name information. It is an example of a *configuration dependency* between the server and its host.

4) Configuration Composition: this relationship allows to divide a main configuration into partial configurations. For example, a message server's configuration is logically split into message services, connection factories and destinations sub-configurations. These sub-configurations are linked to their parent configuration through a *configuration composition*.

5) Configuration Metadata: provides a means to specify metadata for configuration lifecycle management. For instance, one could want to tag specific configurations as default or initial. Another example is the `visited` metadata used in the JORAM platform to mark deployed configurations.

B. Operational State Data Description

As our work targets a global management environment where the managed system is both observable and reconfigurable, we provide constructs to represent information about managed elements as well as their monitored state. A knowledge of the monitored state is required to guide reconfigurations and to assert the operational compliance of proposed configurations. The following concepts allow to describe operational state data (Fig. 5 - Operational state).

1) Managed element: represents the notion of managed element like it is similarly defined in several management information models. A common pattern is to separate managed elements representation from configuration modeling; managed element representations containing monitoring-oriented information. In the case study, the message server is an example of *managed element*.

2) State Parameter: models the traditional operational state attributes like operational status, statistical data, in sum, any collected metrics about the system's operation. The current queue's load or the number of active TCP connections, are examples of state parameters.

In our approach, *Managed Element* and *State Parameter* are the necessary management building blocks for configurations and runtime constraints definition. Their values are supposed to be provided by the existing monitoring framework. They are read-only elements contrary to configuration data.

C. Constraint Specification and Management

The following elements allow to define the constraints that configuration instances should respect as shown in Fig. 5 - Constraint.

1) Constraint: represents the restrictions that must be satisfied by a correct specification of configurations according to the system's architecture and management strategies. Constraints are boolean expressions in a given executable language. The *Constraint* element is subtyped into *offline* and *online constraints* to support the specificities of the two types of configuration validation.

2) Offline Constraint: represents structural integrity rules that a correct configuration data structure and composition of the system should respect. They can be checked either beforehand at design time or during runtime; they do not involve any check against operational conditions. For example, each message destination should have a JNDI name (in order to be looked up by client applications).

3) Online Constraint: defines rules for the operational applicability enforcement. *Online constraints* use *state parameters* values, their evaluation tests the configuration data against convenient state parameters. For instance, a queue's maximum capacity should be kept greater than the current number of pending messages.

4) Constraint Lifecycle Management: Constraints also have additional attributes for their life cycle management: they have a "constraint level" attribute to modulate their strictness. In particular, this allows to assign a severity level to the different constraints (e.g., high, medium, low) and an "active" attribute to activate or deactivate them depending on the operational context and management strategies (e.g., critical vs non-critical).

MeCSV has been formally specified as a UML profile [20]. UML constructs have been tailored to the MeCSV concepts to enable its usage in available UML modelers and to ease the adoption of the MeCSV language. Indeed, UML is well supported by many modeling tools and widely accepted as a standard modeling language.

D. Reference Model Specification Process

The specification process is a two-step process that occurs at design time: first the representation of the reference model structural classes, that is the representation of the configuration data and state data structure, and second, the expression of offline and online constraints.

The completion of these two steps provides the MeCSV reference model of a given management domain that is to be registered into the verification service. It will be used at runtime to check decided configurations.

1) Direct Modeling: Direct modeling is the general process for a MeCSV Reference Model specification. Operators install the MeCSV metamodel into a compliant model editor such as Eclipse Model Development tool (ECLIPSE MDT) and use MeCSV constructs to represent each part of the subsequent reference model.

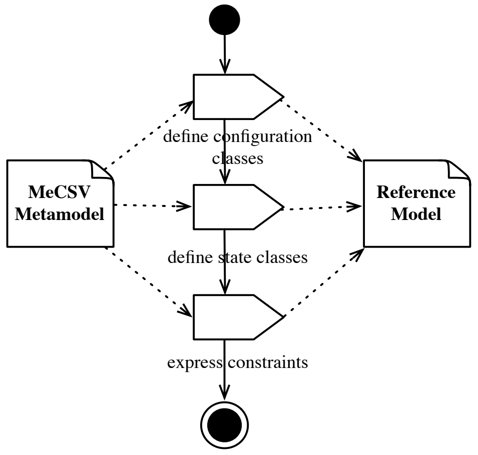


Fig. 6. Reference model design process: direct modeling

As it is shown in Fig. 6, they first describe the configuration parameters and state parameters organized into classes with convenient composition and dependency associations, then they specify the offline constraints that constrain the pure structure of configuration information and the online constraints that help evaluate the compliance of a given configuration information with the execution context at hand.

2) Model Transformation: This general specification process slightly differs when a management information model already exists (Fig. 7).

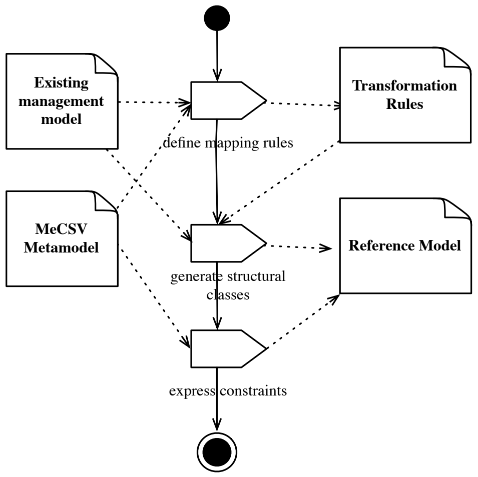


Fig. 7. Reference model design process: model transformation

Indeed, the first step can be automated, mapping rules can be directly defined between the specific management model and MeCSV, thus translating the legacy constructs into the related MeCSV ones. For example, one could use model-driven techniques such as model to model transformation or reflection for the implementation of such mapping rules. The second step of constraints expression remains identical.

VI. Verification Service Architecture

This section details the architecture of the runtime verification service and the resulting verification process.

A. Overview

Fig. 8 gives an overview of the main components of the verification service, a verification engine and a model repository, offering two interfaces, the verification and edition interfaces respectively for the usage of the service and the online edition of MeCSV reference models. The verification interface is supported by the verification engine and the edition interface enables modification of reference models stored in the model repository.

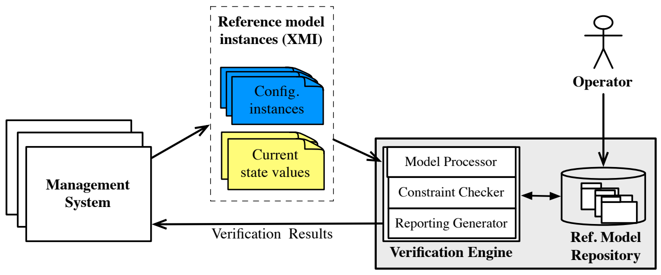


Fig. 8. Online Verification Service

1) Verification Interface: This interface is to be used by a management system to request the verification of configuration instances at runtime. It offers several functions, listed in Table I, that allow to trigger two types of verification: *i)* a complete verification where every existing constraint, in *active* state, is verified or *ii)* a selective verification where a specific subset of constraints are verified according to their type and severity level (e.g., online constraints with a highest severity level).

The verification engine is designed to process model instances conforming to an existing MeCSV reference model classes. Consequently, every API call must contain configuration instances and running operational state values described in a MeCSV-compliant format. Each call returns a verification result object including potential verification errors.

2) Edition Interface: This interface allows the registration of MeCSV reference models to the model repository. It also supports the online modification of registered reference models. Constraints can thus be added, removed, their status and severity can also be updated any time (Table I).

Note that the reference model is reloaded each time it is updated, allowing both the configuration structure and the set of constraints to be modified at runtime. This feature is particularly useful to add or remove constraints according to the management requirements that may change over time.

3) Reference Model Repository: The reference model repository stores MeCSV reference model classes and constraints to be processed during the verification. It also supports the usual creation, update, deletion and querying functions of a database, to adapt the model to evolving management requirements.

4) Verification Engine: The verification engine is the system component that checks provided configuration instances and reports inconsistencies. It provides three capabilities:

- A model processor, capable of analyzing and parsing model elements. It handles verification requests and ensures the existence of a related MeCSV reference model for received configuration instances.
- A constraint-checker, capable of checking dynamically received configuration instances against related reference model classes and available set of constraints. If a constraint is not satisfied, it notifies found errors to a reporting submodule.
- A reporting generator, capable of issueing an indication that contains flawed elements and violated constraints.

The verification engine is built-upon the open source Dresden OCL library [21]. Dresden OCL includes an OCL parser and interpreter that we have enriched with MeCSV specific features such as offline and online constraints differentiation and selective constraint checking, and with new capabilities like runtime modification of reference model constraints.

The *Verification Service* thus allows an existing management system to request verification of live configuration changes, It supports a single tenant as well as a multi tenant usage.

B. Online Verification Process

The following sequence diagram (Fig. 9) shows the interactions involved when a management system requests verification of some configuration instances. Two types of interactions can be identified: internal interactions between the decision and the monitoring modules of the management system and external interactions between the management system and the verification service.

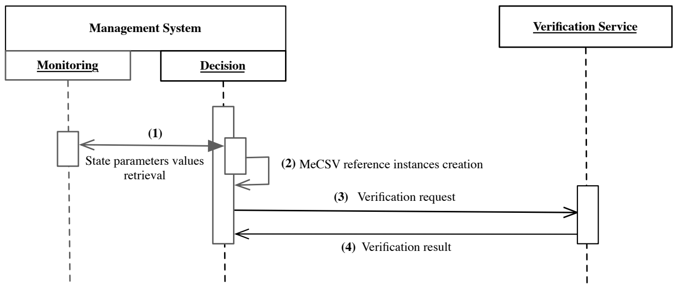


Fig. 9. Online Verification Process

When the decision module elects a configuration to verify, it first interacts with the monitoring module to retrieve current values of defined state parameters, Fig. 9 - (1), then it transforms those data into MeCSV-compliant instance models, Fig. 9 - (2), and sends them to the verification service, Fig. 9 - (3). The verification service checks received configuration instances both structurally and according to retrieved operational state instances of the step (2) and returns verification results, Fig. 9 - (4). It thus enriches management systems with a runtime configuration verification capability.

TABLE I. API CALLS SUPPORTED BY THE VERIFICATION AND THE EDITION INTERFACES

Interface	API call name	Functionality
Verification Interface (access to the Verification Engine)	`validateAll()`	Check given configurations against all existing constraints.
	`validateByConstraintType()`	Check given configurations against constraints of a certain type, e.g., online only.
	`validateByConstraintLevel()`	Check given configurations against constraints of a certain severity, e.g., fatal.
	`validateByConstraintFeatures()`	Check given configurations against constraints of a certain type and severity, e.g., online and fatal.
Edition Interface (access to the Reference Model Repository)	`registerReferenceModel()`	Register a MeCSV reference model.
	`updateConstraintStatus()`	Edit the status of a given constraint, e.g., deactivate a constraint.
	`updateConstraintLevel()`	Edit the severity level of a given constraint, e.g., decrease the severity.
	`updateConstraintFeatures()`	Edit both the status and the severity level of a given constraint.

VII. EXPERIMENT

This section describes the application of the verification framework to the JORAM case study presented in Section IV. Sections VII-A and VII-B describe how we have used the framework at design time to specify a reference model for the case study and how this reference model has been exploited at runtime to execute verification. Section VII-C evaluates this prototype experiment and Section VII-D discusses observations and results.

A. Design time: Reference Model Specification

Following the direct modeling specification methodology, exposed in Section V-D, we installed the MeCSV Eclipse Plugin in the ECLIPSE MDT model editor.

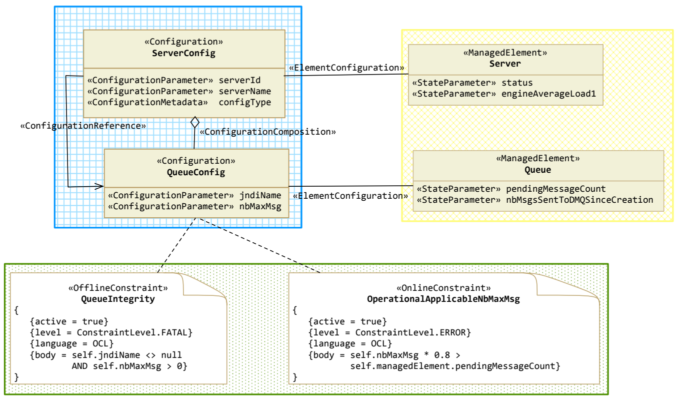
The different configuration concepts of Joram's DTD grammar have been modeled as adequate MeCSV-stereotyped UML classes, attributes and associations forming the configuration information model of the platform.

Constraints have been manually derived from requirements 1, 2 and 3 (cf. Section IV) and expressed in OCL. The following are examples of constraints that have been implemented:

- Each server should have an unique server id.
- Each server should provide a message destination and a connection factory (administered objects).
- Each administered object should have a JNDI name.
- A directory service (JNDI) should be available.
- The JNDI service should be activated and running.
- A queue should not be loaded at more than 80% of its maximum capacity.

The last two constraints are online constraints, they can only be evaluated at runtime, against operational conditions, thus requiring access to monitored data. This operational data has been identified (e.g., servers' operational status, queues' pending messages size, current number of client connections) and modeled as classes and attributes thanks to MeCSV *ManagedElement* and *StateParameter* stereotypes.

Fig. 10 shows an excerpt of the defined MeCSV reference model. This reference model subset contains the high-level configuration structure of a message server including a message queue, the offline and online constraints that should be respected and depending state parameters necessary to enable the operational applicability verification.


«Configuration»
ServerConfig
«ConfigurationParameter» serverId
«ConfigurationParameter» serverName
«ConfigurationMetadata» configType
«ElementConfiguration»
«ManagedElement»
Server
«StateParameter» status
«StateParameter» engineAverageLoad1
«ConfigurationReference»
«ConfigurationComposition»
«Configuration»
QueueConfig
«ConfigurationParameter» jndiName
«ConfigurationParameter» nbMaxMsg
«ElementConfiguration»
«ManagedElement»
Queue
«StateParameter» pendingMessageCount
«StateParameter» nbMsgsSentToDMQSinceCreation
«OfflineConstraint»
QueueIntegrity
«OnlineConstraint»
OperationalApplicableNbMaxMsg

```
{
   {active = true}
   {level = ConstraintLevel.FATAL}
   {language = OCL}
   {body = self.jndiName <> null
           AND self.nbMaxMsg > 0}
}
```

```
{
   {active = true}
   {level = ConstraintLevel.ERROR}
   {language = OCL}
   {body = self.nbMaxMsg * 0.8 >
           self.managedElement.pendingMessageCount}
}
```

Fig. 10. Excerpt of the MeCSV reference model for a MOM application domain

B. Runtime: Online Verification Process

1) Dedicated Management System: For the verification process, we have set up a dedicated management system built upon the JMX management interfaces provided by the JORAM platform. The JMX interfaces comprise a monitoring interface allowing to collect metrics of interest about the running platform and a configuration interface capable of tuning the platform's configuration at runtime. Messaging servers, as well as messaging destinations, can be dynamically added or removed.

This management system is composed of a monitoring module and a decision module. The decision module is capable of choosing a configuration at runtime, requesting running operational data from the monitoring module and transforming these data into MeCSV-compliant instance models to be sent to verification.

In the same time, we implemented several client applications exchanging a high load of fictive messages to act on the monitored metrics (e.g., servers' average message flows, destinations' number of pending messages).

2) Verification Process: The verification process starts with the initialization of the verification service with the defined reference model. As for the management system, the decision module embeds different pre-defined configurations. At runtime, the decision module arbitrarily switches from one configuration to the other and requests the verification of its choice before deploying them.

Once the target configuration is selected, the management system follows the process previously illustrated in Fig. 9. It first retrieves running state values from the monitoring module, then translates them in instances conforming to the defined reference model and finally requests their verification.

C. Experimental Setup

The goal of the experiments was both to test the ability of the MeCSV metamodel to serve as a formal specification notation and to evaluate the effectiveness of the verification service for processing online configuration checking against the defined reference model. We also measured the execution time of the verification process.

We performed our experiments on three different platform configurations varying in size and complexity, namely the number of system's elements (messaging servers, available services, message queues) and dependencies between them.

- The first configuration (Test Case 1) is a centralized messaging server offering basic message features for a total of nine configurable elements.
- The second (Test Case 2) consists of two messaging servers (about eighteen configurable elements).
- The third (Test Case 3) has three messaging servers and holds thirty configurable elements.
- To test the scalability of the validator, we defined a fourth configuration (Test Case 4), made of 300 managed elements, that has been programmatically tested with random state values variations.

Witness verification tests on correct configuration instances have also been conducted for each case.

The summary of test cases data is shown in Table II.

TABLE II. SUMMARY OF TEST CASES DATA

	Nb. servers	Nb. managed elements	Nb. configuration parameters	Nb. state parameters
Test case 1	1	9	57	25
Test case 2	2	18	110	64
Test case 3	3	30	197	103
Test case 4	30	300	1970	103

For each proposed configuration instance, we gradually ran complete verifications with ten, fifty and one hundred OCL constraints with a ratio of 80% offline constraints for 20% online constraints. For each verification request, we took 100 measurements of the execution time in milliseconds and computed the arithmetic mean.

Furthermore, we test selective verifications requesting the evaluation of specific subsets of constraints filtered according to their type and severity. We also test the online edition of constraints, verifying that the verification engine considers their modification.

The tests were run on a Intel® Core™ 2 Duo with 2.66 GHz and 4 Gigabytes of main memory.

D. Results and Discussions

1) Feasability: The verification service has been successfully tested: the received instances were checked against the stored reference model with both offline and online constraints violations detected and notified, both in the case of complete as well as selective verification requests. This permitted the decision module not to apply non-valid configurations.

The detection of online constraints violations, especially in the case of witness verification tests, confirms our thesis about *operational applicability verification*.

A first conclusion that can be drawn from these tests is the effectiveness of the verification service, thus the ability for MeCSV to be used to specify a real-life system's configuration schema and subsequent constraints for online configuration verification.

2) Verification Time: Concerning the verification time, the verification service has a noticeable but reasonable initialization overhead where the MeCSV reference classes and constraints are registered, but after this time, it processes constraint evaluations quickly.

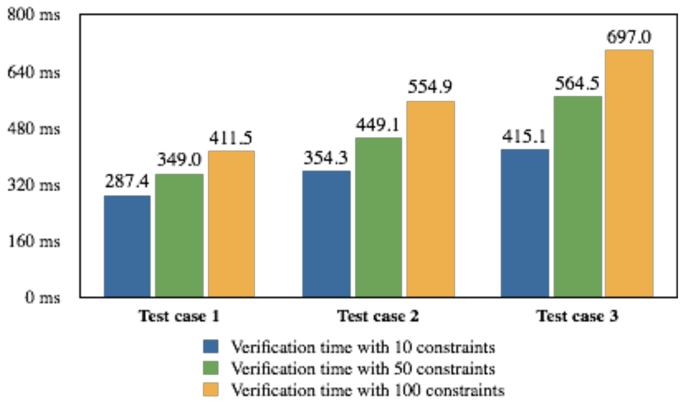


Fig. 11. Verification overhead for the first three test cases

The overall checking time for the three deployed scenarios is under 700 ms, which is very encouraging. It comprises the time taking to check the received instance conformance to the reference model, the constraint evaluation time and the reporting time (negligible).

A very important result lies in the effect of the number of system elements and the number of OCL constraints on the verification time. As shown in Fig. 11, the execution time is not proportional neither to the number of system's elements nor to the number of constraints. For example, while the size of elements quintuples from *test case 1* (6 managed elements) to *test case 3* (30 managed elements), their average verification time ratio hardly doubles (ratio is 1.73). Similarly, although the number of constraints increased by ten, the average verification cost is barely multiplied by 1.5. Further analysis of collected measures showed that constraints were checked in linear time.

We can conclude that in small configurations, the number of system's elements or the number of constraints scarcely affects the verification performance.

Furthermore, we observed that the error rate is not a factor impacting the verification time. An error-free configuration takes the same time as a highly erroneous configuration.

3) Scalability of the approach: The fourth case offers particular insights on the performance of the approach on a very large configuration (Fig. 12). The worst case verification time of 30 message servers (2000 managements parameters and 100 constraints) is far below 2 seconds, which is still an acceptable time for a runtime verification program. This progression confirmed our first conclusion that the verification performance is not proportional to the size of configuration elements. While system's elements increased by 50 (from 6 managed elements to 300 managed elements), verification time increased by less than 5.

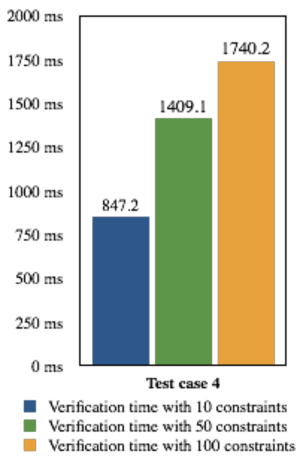


Fig. 12. Verification overhead for the scalability test case (Test Case 4)

The complete verification process overhead is very encouraging regarding the added capability to detect configuration errors at runtime. Indeed, even though configuration instances can be verified beforehand at design time, the difficulty to predict the varying operating conditions can compromise the success of runtime configuration changes.

Altogether, these experimental results confirm the importance of online configuration verification and show the feasibility of our verification framework to enrich a dynamically reconfigurable platform with runtime configuration verification.

VIII. Conclusion and Future Work

Designing lightweight online verification approaches is a critical requirement if we are to build reliable self-adaptive management systems and ease their adoption. This is fundamental as misconfigurations can be prejudicial to the proper operation of the system.

This paper presented a verification framework including an online configuration verification service relying on a high-level specification language named MeCSV. The framework aims to enrich existing management systems with platform-neutral and flexible configuration verification capabilities based not only on structural checks but also on running operational conditions.

We then described a methodology for using the framework from design time to runtime. We applied this methodology on a real-life message-oriented middleware case study where we successfully modeled the configuration schema, validity constraints and operational state data in a platform-independent fashion. This reference model was used by the verification service to process verification requests of configuration instances in viable time.

A series of verification experiments during reconfigurations allowed us to discuss results and observations demonstrating the feasibility of the approach. In future work, we intend to further experience the methodology and integrate more legacy systems so that we can ease the integration process and lower subsequent costs.

References

[1] L. Akue, E. Lavinal, and M. Sibilla, "A model-based approach to validate configurations at runtime," in *4th International Conference on Advances in System Testing and Validation Lifecycle (VALID)*, 2012, pp. 133–138.

[2] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[3] J. Strassner, N. Agoulmine, and E. Lehtihet, "Focale–a novel autonomic computing architecture," in *Latin–American Autonomic Computing Symposium*, 2006.

[4] L. Akue, E. Lavinal, and M. Sibilla, "Towards a Validation Framework for Dynamic Reconfiguration," in *IEEE/IFIP International Conference on Network and Service Management (CNSM)*, 2010, pp. 314–317.

[5] M. MacFaden, D. Partain *et al.*, "Configuring networks and devices with Simple Network Management Protocol (SNMP), RFC 3512," *IETF Request for Comment,[Online]*, pp. 1–69, 2003.

[6] B. H. Cheng, R. De Lemos *et al.*, "Software engineering for self-adaptive systems: A research roadmap," in *Software engineering for self-adaptive systems*, 2009, pp. 1–26.

[7] P. Anderson and E. Smith, "Configuration tools: working together," in *19th conference on Large Installation System Administration (LISA) Conference*, 2005.

[8] N. Samaan and A. Karmouch, "Towards autonomic network management: an analysis of current and future research directions," *Communications Surveys & Tutorials, IEEE*, vol. 11, no. 3, pp. 22–36, 2009.

[9] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, ser. USITS'03, 2003, pp. 1–1.

[10] "CIM Schema version 2.29.1 - CIM Core," Distributed Management Task Force (DMTF), June 2011.

[11] M. Bjorklund, "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)," Internet Engineering Task Force (IETF), RFC 6020, October 2010.

[12] A. V. Konstantinou, D. Florissi, and Y. Yemini, "Towards Self-Configuring Networks," in *DARPA Active Networks Conference and Exposition (DANCE)*, 2002.

[13] D. Agrawal, J. Giles *et al.*, "Policy-based validation of san configuration," in *5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY) 2004*, 2004, pp. 77–86.

[14] E. Gençay, C. Sinz *et al.*, "SANchk: SQL-based SAN configuration checking," *IEEE Transactions on Network and Service Management*, vol. 5, no. 2, pp. 91–104, 2008.

[15] P. Goldsack, J. Guijarro *et al.*, "The SmartFrog Configuration Management Framework," *ACM SIGOPS Operating Systems Review*, vol. 43, pp. 16–25, 2009.

[16] T. Hinrichs, N. Love *et al.*, "Using object-oriented constraint satisfaction for automated configuration generation," in *DSOM*, 2004, pp. 159–170.

[17] L. Ramshaw, A. Sahai *et al.*, "Cauldron: a policy-based design tool," in *7th IEEE International Workshop on Policies for Distributed Systems and Networks*, 2006, pp. 113–122.

[18] T. Delaet and W. Joosen, "PoDIM: A Language for High-Level Configuration Management," in *LISA*, 2007, pp. 261–273.

[19] "Java^{TM} Open Reliable Asynchronous Messaging (JORAM)," OW2 Consortium, June 2013. [Online]. Available: http://joram.ow2.org/

[20] "OMG Unified Modeling Language (OMG UML), Superstructure V2.1.2," november 2007.

[21] "Dresden OCL," TU Dresden, Software Technology Group, June 2013. [Online]. Available: http://www.dresden-ocl.org/