An Autonomous Engine for Services Configuration and Deployment

Félix Cuadrado, *Member*, *IEEE*, Juan C. Dueñas, *Member*, *IEEE Computer Society*, and Rodrigo García-Carmona, *Member*, *IEEE Computer Society*

Abstract—The runtime management of the infrastructure providing service-based systems is a complex task, up to the point where manual operation struggles to be cost effective. As the functionality is provided by a set of dynamically composed distributed services, in order to achieve a management objective multiple operations have to be applied over the distributed elements of the managed infrastructure. Moreover, the manager must cope with the highly heterogeneous characteristics and management interfaces of the runtime resources. With this in mind, this paper proposes to support the configuration and deployment of services with an automated closed control loop. The automation is enabled by the definition of a generic information model, which captures all the information relevant to the management of the services with the same abstractions, describing the runtime elements, service dependencies, and business objectives. On top of that, a technique based on satisfiability is described which automatically diagnoses the state of the managed environment and obtains the required changes for correcting it (e.g., installation, service binding, update, or configuration). The results from a set of case studies extracted from the banking domain are provided to validate the feasibility of this proposal.

Index Terms—Autonomic systems, model-based management, satisfiability, service configuration.

1 Introduction

The Service-Oriented paradigm aims at revolutionizing how software is developed, assembled, and put into production. Vendor independence, dynamic composition, runtime adaptation, and Internet-wide communications are key elements for the success of this paradigm. Services are, in essence, distributed applications whose functionality is described by well-defined interfaces, running on logical containers over networked computers.

Nowadays, services seem to be the preferred way to implement domain specific processes in industry as services can be composed and connected across organizational boundaries. Thus, business level requirements on company processes, such as those related to reliability, security, or performance, must be supported by the services taking part in the processes. However, this does not change the fact that services are provided by software components that run over service execution platforms (including containers, computers, and network elements). Therefore, those requirements imposed on services are translated to their execution infrastructure.

This context presents multiple challenges to ensuring services and service environments fit to the business requirements they have been designed for. Management operations are especially affected by those factors as services implementations, as well as their infrastructure, frequently change. The challenges for automating the change operations to enterprise services include managing the complexity and heterogeneity of the execution environments, reasoning about all elements running over the distributed platform, taking care of the existing dependencies, relationships, and constraints between all of them, and ensuring that the managed environment complies with the defined business objectives.

Heterogeneity, distribution, and awareness of outer elements greatly complicate the tasks of deployment and runtime operation. These factors are taken into account by operation tools following the traditional management paradigm—a human operator working with a management console. However, reasoning activities such as adaptation and decision making on the information provided by the services platform are seldom automated, resulting in a true bottleneck for the operation of services.

In this paper, we present our results on improving the automation capabilities of the management infrastructure for heterogeneous, distributed services infrastructure. The main contribution of this work is the characterization of the correctness of a distributed system as a quantifiable formula (including functional aspects, logical dependencies, restrictions over the hardware platform, and business objectives of the whole environment). Building on top of that, we have designed a satisfiability (SAT)-based resolution engine that is able to automatically reason about that information in order to determine the required management operations. As an additional contribution, we present a set of models that can be obtained by instrumentation agents and contain all the required information about the runtime platform and the available services, serving as input of the previous elements.

- *F. Cuadrado is with the School of Electronic Engineering and Computer Science, Queen Mary University of London, Mile End Road, E1 4NS, London, United Kingdom. E-mail: felix.cuadrado@eecs.qmul.ac.uk.*
- *J.C. Duenas and R. Garcia-Carmona are with the Departamento de Ingeniería de Sistemas Telemáticos, Escuela Técnica Superior de Ingenieros de Telecomunicación, Universidad Politécnica de Madrid, Avenida Complutense 30, Ciudad Universitaria, Madrid E28040, Spain. E-mail: {jcduenas, rodrigo}@dit.upm.es.*

Manuscript received 24 Mar. 2010; revised 30 June 2010; accepted 26 Jan. 2011; published online 2 Mar. 2011.
Recommended for acceptance by C. Ghezzi.
For information on obtaining reprints of this article, please send E-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2010-03-0085.
Digital Object Identifier no. 10.1109/TSE.2011.24.

The paper is structured as follows: Section 2 provides some background information about the problem of automating services management and the selected procedure for this. Section 3 describes the information models we use in our solution and outlines some key management definitions built over them, such as environment changes, as well as desired, stable, and correct configurations. The next one details the algorithms for reasoning upon them autonomously, obtaining, if possible, a solution for reaching a desirable and stable configuration; this solution must also fit other restrictions and constraints imposed by the services platform infrastructure. In Section 5, the proposal is validated through a set of case studies derived from the banking domain. Finally, the last section reflects on the explained aspects, focusing on the impact and potential extensions to this work.

2 Background Information

In this section, we will analyze the underlying challenges for automating the service management tasks and evaluate the existing approaches and alternatives for addressing them.

2.1 Automated Services Management

The Service-Oriented Architecture (SOA) paradigm [1], [2] is fundamentally changing the way systems are created and operated as it paves the way for reuse and collaboration in a dynamic, distributed context [3]. However, in order to truly seize its potential, all the software engineering disciplines must provide solutions to the new challenges which have emerged, ranging from service design to their dynamic runtime adaptation [4].

In this paper, we will focus on the intermediate and latter stages of the service life cycle: the deployment (or provisioning) of the services and their runtime management, including reconfiguration for corrective maintenance. The complexity of these activities lies in the need to support the runtime and generic characteristics of SOA while at the same time coping with the complexity and heterogeneity of the supporting infrastructure. In this context, it is fundamental to provide methods and techniques that improve the automation capabilities of the management systems as manual operations are no longer cost-effective and greatly impact the agility of service-based applications.

Automated management has been associated recently with the autonomic computing paradigm. This approach aims to lessen the burden on system administrators by enabling a completely automated management of the infrastructure [5]. The potential advantages that can be obtained by this paradigm are usually expressed in the form of four distinct self-management capabilities. *Self-Configuration* consists of the automatic installation and configuration of the components of an autonomic system, without needing human intervention. A *Self-Healing* system diagnoses itself continuously, detecting functional failures and reconfiguring itself in order to correct them. *Self-Optimization* is achieved whenever the system monitors its resources with respect to defined requirements and policies, continuously configuring them in order to improve its service-level quality. Finally, *Self-Protection* capabilities enable the system to proactively identify intruders and defend from their attacks. Any system supporting one of those self-capabilities is said to present autonomous behavior. The importance of those aspects for service management is highlighted in [6].

The behavior of a completely automated (autonomous) management system is often represented with the concept of a closed control loop. The manager monitors the managed system through a sensor channel, analyzes its state (including all the relevant information), and automatically obtains and executes the required changes through an actuator channel [7].

As regards the autonomic support of both deployment and reconfiguration of services, there is no single initiative that directly addresses these topics. Several contributions have been proposed to automate network management through the use of policies and ontologies, with PMAC [8] and FOCALE [9] being the prime representatives. However, these initiatives do not address the service management aspects. Nonetheless, at the application and service level some contributions can be found which bring self-management capabilities to specific elements of the services domain, such as a single application server [10] or a home service gateway [11]. However, in those cases the distribution aspects of a service-based system are not supported. Finally, there are several interesting works which propose autonomic architectures to support runtime services management [12], [13]. However, these alternatives are tied to a specific middleware technology, the CORBA distributed object model, which complicates their adoption for a generic service infrastructure management system.

An autonomous service management system must be generic (technology-independent) and at the same time support automated reasoning over the heterogeneous and distributed infrastructure. These factors can be addressed by adopting a standard information model to capture all the relevant information about the managed environment in a generic way. Since this necessity has been addressed for a long time, we will first provide a brief overview of the most extended information modeling standards from the network and systems management domain. The DMTF Common Information Model (CIM) is the best-known standard [14], providing a comprehensive characterization of every manageable element, from the network to the specific applications. The OMG Deployment and Configuration of Distributed Systems (D&C) defines a clean model of a distributed system and their managed elements [15]. Recently, a lot of interest has been originated by the OASIS web Services Distributed Management (WSDM) initiative [16], [17], promoting the use of web services as the main driver for distributed management through the Management Using web Services (MUWS) and Management of web services (MOWS) [18], [19] specifications. Finally, the OASIS Solution Deployment Descriptor (SDD) [20] presents a deployment-centric model for service operations. All of them share the same fundamental abstractions, built over the concept of resource, which encapsulates an atomic management unit. Their abstractions range from the extensive subclassing proposed by CIM to more generic approaches such as the extensible D&C typing mechanism, which allows grouping similar elements. Those abstractions greatly simplify the implementation of automated solutions, but they must be complemented in order to fully capture the required information and restrictions for an effective service management.

2.2 Applications of Satisfiability

In order to close the control loop the manager must be able to automatically process all the management information and find the required correction operations which will be applied. Recently, Satisfiability has gained traction as a technique for efficiently finding solutions to problems with very large search spaces. Some problems expressed in SAT include automatic test pattern generation, redundancy identification, and elimination, Field Programmable Gate Array (FPGA) routing, or model correctness checking [21]. Over the following paragraphs we will provide some basic information about this technique and highlight its use on some specific problems closely related to the context of this paper.

The SAT problem consists of finding an assignment to the variables of a Boolean function that evaluates it to true [22]. In addition to the base SAT problems, there are several variants, such as MaxSAT, consisting of finding the maximum number of positive literals of the formula, or Pseudo Boolean SAT (PB SAT), which supports inequality clauses with linear expressions over Boolean variables, as well as the definition of an optimization function which will be maximized. In spite of the NP-complete complexity of the problem family, current generation solvers are able to find a suitable solution for a large subset of them in a very short period of time.

Modern solvers are generally based on the Davis-Putman-Logemann-Loveland (DPLL) algorithm [23], which provided a sound and complete way of obtaining the solution to the problem of satisfiability. This base algorithm has been heavily optimized in recent years by adopting strategies such as conflict analysis techniques (which try to obtain the reason for an assignment conflict), conflict-driven learning (adding clauses to avoid the same conflict in the future), or the incorporation of heuristics to detect conflicts before they occur, and optimizing the way the search space is explored [24], which has greatly extended the domains where this technique can be applied.

There is one specific field of application of SAT which has led to its consideration for the proposed algorithm: the support for dependency resolution in installation processes, first adopted by the OPIUM prototype [25]. The SAT applicability to this field has been demonstrated since 2008, with a refinement of this technique being adopted for powering Eclipse's p2 update manager operations (install, update and uninstall) [26]. The main innovation of this example is the use of a Pseudo Boolean SAT solver, which allows the expression of linear inequalities in addition to SAT first order constraints, and also supports the definition of an optimization function to obtain more satisfactory results. As it is described in [27], this is an increasing trend, with both the dependency management of the Maven 3.0 release and the openSuse 11.0 Linux distribution being addressed by this technique.

SAT solvers are also applied for automatically obtaining valid configurations of network equipment such as routers and bridges [28]. By modeling the relevant characteristics of every manageable element and defining their restrictions using propositional logic, these engines can automatically find correct configurations for each element or diagnose the correctness of a preset configuration. An alternative approach for the same problem is addressed in [29], where instead of modeling the problem as a SAT, the solution is based on the tool Alloy. This tool provides a solver that, given a model definition with first order logic constraints, obtains an instance of the model matching the restrictions. While Alloy internally uses a SAT solver, the tool abstracts users from the conversion to a SAT problem and SAT results interpretation steps. The SelfSoC [29] approach provides another example of the use of Alloy, addressing the problem of dynamic component assembly similar to the previously presented deployment solutions.

Before ending this overview we will mention another interesting application of SAT, in this case for generating test cases of a complex system configuration [31]. This work highlights how this technique can be applied to find efficiently solutions to a search space where multiple constraints over the correct solution are defined.

As a conclusion, none of the analyzed initiatives addresses the problem of automating the management of a service infrastructure, although SAT has been successfully applied to affine problems (dependency management and constraints-based configuration search).

3 A Model for Services Deployment and Configuration

As it has been previously described, an autonomic service manager must be based on a metamodel that captures all the relevant management information of a heterogeneous infrastructure in a generic way. These abstractions are vital, as management decisions and actions will be based exclusively on the modeled concepts and relationships. This section describes the main model definitions, the relevant information, and the inferred environment stability and correctness definitions, which will completely characterize the task to be solved by the automated system.

3.1 Model Foundations

The proposed information model is generic and object-oriented, building over the base concept of *Resource*. This base definition characterizes a manageable element (described with a set of *Properties*) with a *name,* an optional *version*, and a *type*. The *type* concept is fundamental as it classifies an initially unknown set of resources. This allows us to define expressive constraints in a generic way which can be automatically processed. These concepts constitute the common ground for most of the analyzed standards, including D&C, SDD, and MUWS (as it is described in [32]), making the proposed solution compatible with them.

On top of those basic concepts, we present a complete model of the services configuration and deployment information, detailing its inferred stability conditions for an automated diagnosis. This is an evolution of our previous work on modeling to characterize virtualized environments [33] and support installation processes [34], enabling an autonomic management of the services infrastructure.

Resource type attributes belong to the *rType* class, which provides increased expressiveness over individual

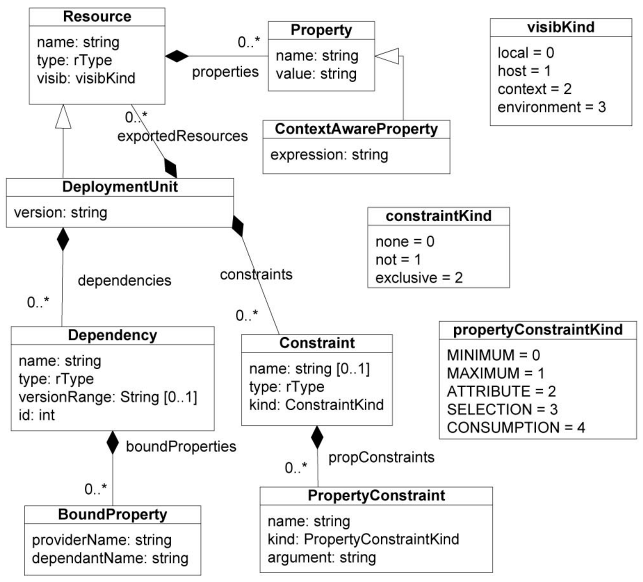

Fig. 1. The logical information model.

classifications. An *rType* object contains a hierarchy of types, with progressively more specific categories. For serialization purposes, *rType* instances are represented similarly to Java packages. This way, a network card of type "hw.net.interface.ethernet" belongs simultaneously to the types {"hw," "hw.net," "hw.net.interface," "hw.net.interface.ethernet"}.

The presented abstractions can be divided into two complementary groups, depending on if they define logical or runtime information. Both are built over the same resource abstractions and are tightly integrated: The logical model describes the services and artifacts which must be considered over the deployment and configuration activities, while the runtime information characterizes all the managed elements from the runtime environment, including the instantiation of the logical elements.

3.2 Logical Information Model

The logical information model characterizes the services and deployable artifacts that provide them. This model provides a deployment and configuration perspective of the artifacts obtained from the service development process. The presented abstractions keep the requirements of the applications and services that have been taken into account over the analysis, design, implementation, and testing phases. Fig. 1 shows the main elements of this model. The root element of the model is the *Deployment Unit, a* subclass of *Resource* representing each indivisible artifact which provides functionality when provisioned to the environment (typical units are WAR and EAR JEE deployable files, OSGi JAR Bundles, and packaged SQL scripts). In addition to the *name* and *version*, the unit's *type* identifies the kind of packaging (i.e., the format of the file).

A *Deployment Unit* aggregates functionality, such as services, libraries, web interfaces, and business logic components, which will be available when the unit is deployed to the runtime environment. These elements are modeled as *Resources.* The typing mechanism allows a rich characterization of different elements with the same common concepts, and a *visibility* attribute controlling what elements from the environment can access them. Finally, the internal configuration of the *Deployment Unit* is reflected as a set of *Properties* (e.g., the connection parameters for accessing an external database).

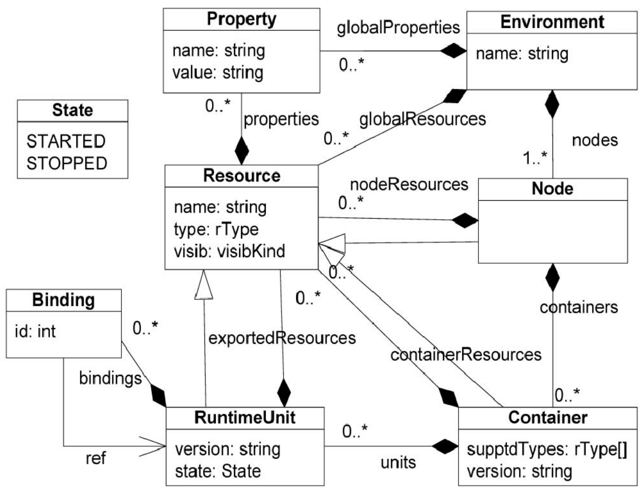

Fig. 2. The runtime information model.

3.3 Runtime Information Model

The runtime information model defines the resources, which constitute the managed environment and their relationships, ranging from hardware assets (such as hard disks, RAM memory, or network devices) to software elements (such as available ports from a machine, operating system, and application server services, or a directory service). Fig. 2 depicts its main elements, which describe a hierarchical *Resource* structure. This presents a clear management view of the infrastructure and running services, with specific elements representing the key concepts of the system (nodes, containers, units).

The root element is the *Environment*, which represents the runtime environment itself. The *Environment* is composed of *Nodes*, global *Resources* (such as a DNS naming service, or third-party remotely available web services), and global configuration *Properties*.

Nodes model *Resources* with computing capabilities (like PCs or servers). A *Node* comprises all the hardware and low-level software layers of the device (such as the operating system). The specific components, libraries, communication channels, and devices are abstracted as *nodeResources*. On top of that substrate, a *Node* hosts any number of *Containers*.

A *Container* is the base execution platform for the applications and services (modeled as *Deployment Units*). *Containers* have a *name* (unique over the environment), a *type* classifying it (examples include "*container.database.jdbc.oracle*" or "*container.jee.websphere*"), and a *version*. These elements provide platform services to the hosted units, which are expressed as a set of *Resources* (typical application server resources would be Datasource connections, JMS queues, or remote system connectors).

Because of its key role in service configuration management, the hosted runtime instances of the *Deployment Units* are explicitly represented in the model. The instantiation of a *Deployment Unit* in a *Container* from the environment creates

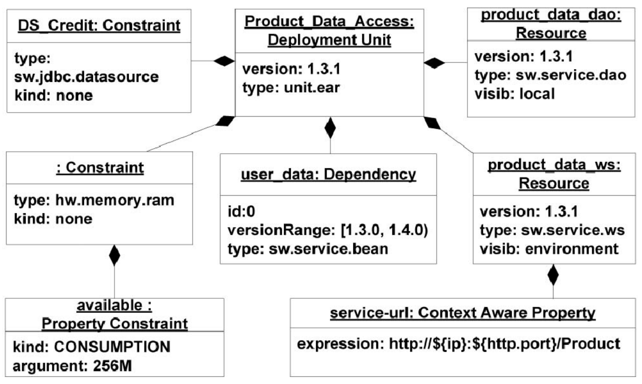

Fig. 3. Sample deployment unit instance.

a *Runtime Unit*. The *supportedTypes* attribute of the *Container* restricts which types of units can be deployed over it.

A *Deployment Unit* shares with all its *Runtime Unit* instances the identity information (*name*, *version*, and *type*), as well as the initial configuration *Properties* and exported *Resources*. This allows keeping the traceability between the two models. *Runtime Units* also have a *state*, indicating their life cycle stage.

3.4 Unit Configuration Restrictions

Up to this point, we have explained the main entities of the model, the contained information, and its relationships. These constructs provide us with all the observed information for managing the system. However, these concepts are not sufficient for enabling an autonomic management. It is also necessary to define the configuration restrictions of each managed element. Following the logical and runtime partition of model concepts, the restrictions will be defined at the *Deployment Units* and will be validated at each one of the runtime instances. For illustrating these concepts, an exemplary *Deployment Unit* instance, shown in Fig. 3, will be used over this section to provide specific examples of the restrictions imposed by model instances.

Deployment Units are not independent entities; whenever they are instantiated at the environment they collaborate with other resources to provide the final services. Therefore, those interactions must be explicitly reflected as part of the model, allowing its automatic check. We have identified two types of relationships depending on the type of elements. The relationships between units (such as service provider/consumer) are defined as *Dependencies*. On the other hand, some artifacts require additional capabilities from the execution environment *Resources* for a correct operation (such as a TCP port). Those conditions are expressed with *Constraints*. Fig. 1 shows how the model allows *Deployment Units* to express these restrictions.

A unit can define any number of *Dependencies*, with each one demanding the access to a runtime *Resource* provided by another unit. The *Resources* that can satisfy the dependency are identified by defining a filter consisting of *name*, *type*, and, optionally, *versionRange*. For a resource to match the filter, both names must be equal, the resource version must be contained in the presented range, while the filter type must be a subset of the resource type. For the example depicted in Fig. 3, the satisfying resource must be named *user_data*, contain the type *sw.service.bean*, and have a version value between 1.3.0 (inclusive) and 1.4.0. In addition to this, the *Runtime Unit* definition contains in the *Bindings* attribute the information about how *Dependencies* are satisfied at the runtime environment. This way, each logical *Dependency* must result in a runtime *Binding,* containing a reference to the satisfying unit. In addition to the filter restriction, the bound *Resource* must be accessible (as restricted by the *Resource* visibility attribute) to the dependent *Resource* in order for the *Binding* to be valid. The addition of *Bindings* to the runtime information model builds a dependency overlay structure over the runtime containment hierarchy. This environment-wide dependency layer reflects real restrictions which must be considered in order to safely apply any changes over already existing *Runtime Units*, as well as tracing the impact derived from any change to the runtime elements. Therefore, the expressivity of the runtime model is considerably improved by this addition.

Constraints are declared in the *Deployment Unit* definition with a similar mechanism. Each *Constraint* element will demand the presence of a specific *Resource* at the unit's execution environment (composed of its runtime hosts: the *Container*, *Node*, and *Environment*). Similarly to the previous restriction, valid *Resources* are identified by a filter composed by a *type* and optionally a *name*. The example shows two *Constraints* representing different restrictions. The *Constraint* depicted at the top-left corner represents a basic definition; it demands the existence of a *Resource* named *DS_Credit*, of type *sw.jdbc.datasource*. This *Constraint* models a JDBC Datasource Connection, which is provided by standard Java Enterprise Edition Application servers (modeled as *Containers*) and can be configured through a management operation.

However, the model for *Constraints* provides additional expressivity. First, the *kind* attribute supports two variants to the default (of kind *none*) restrictions. *Exclusive Constraints* demand that no other *Runtime Unit* simultaneously access the same *Resource* (representing scenarios such as the reservation of a TCP machine port). *Not Constraints* declare an incompatibility with the identified *Resource*, which cannot be part of the unit execution environment.

Second, the resource identification filter can be extended with additional requirements over the values of the *Resource Properties*, expressed in *Property Constraint* elements. Each *Property Constraint* imposes a restriction to the value of a *Property* with the specified *name*. *Property Constraints* are based in the restrictions defined in the D&C model. The *kind* attribute defines what type of check must be applied (minimum amount—at least the specified value, maximum amount, selection from a list of valid values, and consumption of numeric resource), and the *argument* attribute provides the values to be compared with. They represent restrictions such as hard disk capacity usage, or minimum processor speed. The RAM *Constraint* from Fig. 3 shows another example of the numeric consumption kind. The definition contains a *Property Constraint* demanding the consumption of 256 M of the property named *available* of a *Resource* of type *hw.memory.ram.* As in the other example, the satisfying *Resource* must belong to the execution environment of the unit. This *Constraint* would typically be satisfied by a *Node Resource* representing the RAM memory state.

These restrictions allow declaring what must be available in the runtime environment in order for the *Runtime Unit* to work correctly. In addition to that, the defined model provides abstractions that further support automated service management. This is achieved by linking the internal configuration of a *Runtime Unit* to the environment state. Depending on the runtime elements which provide the configuration values, two types of restrictions can be established to *Property* values. *Bound Properties* link *Property* values between the *Runtime Units* connected with a runtime *Binding*. A *Bound Property* mandates that the value of the *Property* from the dependent unit named *dependentName* must be equal to the value of the *Property* named *providerName* from the *Resource* of the bound unit. This allows automatic support of scenarios such the automatic configuration of a connection URL to access a web service provided by another *Runtime Unit*.

The second type of restrictions is expressed through a *Property* subclass named *Context-Aware Property*. This specialized subclass specifies the *expression* attribute that defined how the *Property value* can be obtained by composing values from *Properties* of their execution environment (e.g., retrieving the temporal folder from an application server). The example unit shows how this mechanism can be used for internal configuration enforcement. In this case, the value of the *Context-Aware Property* service-url will be obtained by substituting in the provided expression the two variables (*ip* and *http-port*). Their values are obtained from the execution context *Resources*.

3.5 Environment Stability Definition

The definitions of *Constraint*, *Property Constraint*, *Dependency*, *Bound Property*, and *Context Aware Property* comprise every restriction that can be expressed at the logical definition over the configuration of a *Runtime Unit.* They specify where the unit can be deployed, what other units can be dynamically bound to it, and whether the values of its properties have correct values.

We have defined the concept of stability for reflecting these configuration requirements. An existing *Runtime Unit* is stable if it is correctly configured at the environment (assuming for management purposes that it works properly). *Runtime Unit* stability is defined the following way: A *Runtime Unit* is stable if it is instantiated at a compatible *Container* and every configuration restriction defined at its corresponding *Deployment Unit* model is met by the current runtime environment configuration.

The stability concept is extensible to the remaining resources of the *Environment (Containers* and *Nodes)*. However, as the model does not define any restriction about their configuration, they are always stable. Finally, we can define stability for the complete *Environment* configuration by combining the stability checks of each of its members. Consequently, an *Environment* configuration *Cf* will be stable if every *Runtime Unit* u_i, $u_i \in Cf$, is stable, as shown in

$$Stability(Cf) = \bigwedge_{i=1}^{n} Stability(u_i). \qquad (1)$$

The stability formula is fundamental for management automation as it dictates whether the system is adequately

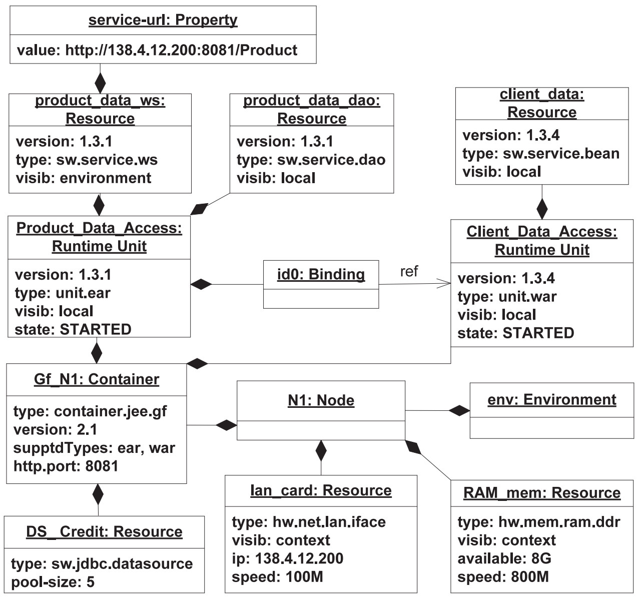

Fig. 4. Sample runtime environment configuration.

configured or not. Consequently, this function must be obtainable and evaluable just from a runtime environment model and a set of logical *Deployment Unit* definitions. The formula is composed of multiple restrictions, with each of them needing to be matched for a global stable verdict. Each individual restriction is originated by one of the existing *Runtime Units*, so the formula can be built by combining the restrictions from each existing unit.

The stability restrictions related to a *Runtime Unit* are obtained in the following way: All *Units* impose the host compatibility restriction (mandating that the *Container* where the unit is executing supports the unit's type). Every remaining restriction is specified in the *Deployment Unit* model from which the *Runtime Unit* has been instantiated. Specifically, each *Constraint*, *Property Constraint*, *Dependency*, *Bound Property*, and *Context Aware Property* can be directly translated to a function that will be evaluated to true or false when its variables are substituted with the runtime values.

In order to illustrate how the environment stability function can be obtained and evaluated, we will use the example *Environment* presented in Fig. 4. It is composed of a *Node*, a *Container*, and two *Runtime Units*. For space reasons, node resource *Properties* are represented as attributes. This *Environment* will be stable if both Client Data Access (CDA) and Product Data Access (PDA) *Runtime Units* are stable. First, each unit imposes a restriction about its compatibility with the host *Container* (identified with the function *HostOf)*:

$$unit.ear \in HostOf(PDA).supportedTypes, \qquad (2)$$

$$unit.war \in HostOf(CDA).supportedTypes. \qquad (3)$$

The remaining stability restrictions from each unit are obtained from its corresponding *Deployment Unit* model. We will focus on PDA, whose logical definition was presented in Fig. 3. Its model includes five elements that restrict the unit configuration, resulting in the following five blocks of stability constraints.

1. Stability restriction from *Constraint* C1 (DS Credit):

$$\begin{gathered}\exists Resource\ r_1, r_1 \in ExecutionContext(PDA)| \\ r_1.name = DS\_Credit \wedge \\ r_1.type \subseteq sw.jdbc.datasource.\end{gathered} \tag{4}$$

2. Stability restriction from *Constraint* C1 (DS Credit):

$$\begin{gathered}\exists Resource\ r_2, r_2 \in ExecutionContext(PDA)| \\ r_2.type \subseteq hw.memory.ram.\end{gathered} \tag{5}$$

3. Stability restrictions from *Property Constraint* PC of C2, of kind *consumption*. *Property Constraint* restrictions inherit the scope from the parent *Constraint* restrictions (in this case C2):

$$\begin{gathered}\exists Property\ p, p \in r_2.properties| \\ p.name = available.\end{gathered} \tag{6}$$

The second restriction is determined by the consumption kind. As these restrictions are competitive, the group of *Runtime Units* $\{u_i\}, i \in [1, n]$, consuming *Property* p of $Resource\ r_2$, must be obtained. The set $\{c_i\}$ contains the amount (*argument* attribute) consumed by each resource (256 for PDA):

$$\sum_{i=1}^{n} c_1 < p.value. \tag{7}$$

4. Stability restrictions from *Dependency* D0, with Runtime *Unit* BU being the one referenced by the *Binding* id0 of PDA:

$$\begin{gathered}\exists Resource\ r_3, r_3 \in \{BU\} \cup BU.exportedResources| \\ r_3.version \in [1.3.0, 1.4.0) \wedge \\ r_3.type \subseteq sw.service.bean.\end{gathered} \tag{8}$$

Dependencies also impose a visibility stability check. In this case, as the Resource r_3 of BU is visible locally, the following restriction is defined (mandating the same *Container* for both Units):

$$HostOF(BU) = HostOF(PDA). \tag{9}$$

5. Stability restriction for *Context Aware Property* service-url: The value of the service-url *Property* from PDA must be equal to the *expression* attribute, with the two defined variables substituted by the correct values. This way, ${ip} must be substituted by the value of a *Property* named ip of a *Resource* in the PDA execution context (in case there are multiple matching properties, the nearest in the containment hierarchy will be the one whose value is taken). The ${http.port} variable must be substituted the same way. Additionally, the *Context Aware Property* imposes an implicit restriction: The execution context of the unit must contain resources with properties named "ip" and "http.port" so that the value can be successfully obtained.

The obtained stability restrictions can be evaluated by assigning to the variables the actual runtime environment values. From what we can discern, the environment depicted at Fig. 4 is stable as it addresses every known stability restriction.

Although only a specific example has been described, the same method can be applied to obtain the stability formula of any combination of a runtime environment model and a set of definitions of the corresponding *Deployment Units*. The formula will be composed of a number of stability clauses derived from the logical definitions, and it will be evaluated by substituting the values of the existing runtime resources.

To sum up, the described information model defines a rich set of restrictions for software applications and services that can be clearly evaluated against the runtime state through the stability formula. This is achieved by defining a common set of abstractions (the *Resource* concept from WSDM and, especially, D&C, and the *Deployment/Runtime Unit* concept) for both logical and runtime modeling, which had not been simultaneously covered by the analyzed specifications.

The logical model characterizes logical units and services, extending D&C and SDD restrictions in order to provide a complete set of abstractions. The main additions are *Resource*-based *Dependencies* and configurable *Properties*, whose correct values are extracted from either the *Runtime Unit Bindings* or its execution context.

The runtime model extends D&C in order to improve its representation of running applications and services. In addition to *Runtime Units*, the *Container* concept explicitly characterizes the direct execution platform of runtime software and services, while *Bindings* enable runtime traceability of logical *Dependencies*. Logical definitions are instantiated into runtime elements, and their restrictions can be validated through the existing resources.

3.6 Environment Correctness Definition

In the previous sections, we have presented a model which provides a complete view of the managed environment, as well as a set of abstractions for diagnosing its *stability*. This concept is very important for automating management activities as it allows diagnosing the runtime environment with a check against the modeled information.

However, this is not sufficient to enable an autonomic service management. In addition to being able to determine whether the environment elements have a stable configuration, we must also be able to know if the system is fulfilling the desired business objectives. In other words, we must know if *the environment is doing what it must and not doing what it must not do*.

We have made this requirement possible by incorporating a third branch of concepts to the presented model: the *Objectives*. These elements represent conditions which must be met by the existing environment in order to provide the desired functionality. *Objectives* refer to *Resources* in order to preserve consistency among the information models. Two types of *Objectives* have been introduced: EXISTS(r) demands the presence of a Resource in the environment, while its opposite condition, NOTEXISTS(r), forbids its existence. This way, the functional requirements of the runtime infrastructure are expressed through Resources (e.g., what services must be running).

TABLE 1
Internal Changes Definition

Name	Arguments	PostCondition
INSTALLUNIT	*Container cont, DepUnit du*	$\exists RuntimeUnit\ ru, ru \in cont.units \land ru\ instanceof\ du$
UNINSTALLUNIT	*Container cont, RuntimeUnit ru*	$ru \notin cont.units$
UPDATEUNIT	*Container cont, RuntimeUnit ro, DepUnit dun*	$\exists RuntimeUnit\ rn, rn \in cont.units \land rn\ instanceof\ dun \land ro \notin cont.units$
ADDCONTRES	*Container cont, ConfCRes crc, String name*	$\exists Resource\ cr, cr \in cont.resources\ \land\ cr\ instanceof\ crc \land\ cr.name = name$
RMVCONTRES	*Container cont, Resource rc*	$rc \notin cont.resources$
CONFIGRES	*Container cont, Resource rc, Props props*	$props \subseteq rc.properties$
STARTUNIT	*RuntimeUnit ru*	$ru.state = STARTED$
STOPUNIT	*RuntimeUnit ru*	$ru.state = STOPPED$
CONFUNITPROP	*RuntimeUnit ru, Property[] props*	$props \subseteq ru.properties$
CONFBINDING	*RuntimeUnit ru,rb, int bindId*	$\exists Binding\ b, b \in ru.bindings \land\ b.id = bindId \land b.ref = rb$

Once these elements have been introduced, the *Desirability* of an environment configuration *Cf* with an associated set of defined *Objectives O* can be defined analogously to the previous concept of environment *Stability*:

$$Desirability(Cf) = \bigwedge_{i=1}^{|O|} o_i, o_i \in O. \qquad (10)$$

Finally, it is possible to formalize the requirements for any environment configuration to be correct as a combination of these two factors. We define the *Correctness* of a configuration as the state in which that configuration is at the same time *desirable* and *stable*:

$$Correctness(Cf) = Stability(Cf) \land Desirability(Cf). \qquad (11)$$

As the combination of logical resources, runtime resources and objectives encapsulates all the information relevant to the management of any specific environment; we will use the term *Management Domain* to refer to the information collectively. A *Domain D* is a triplet composed of the current configuration of the environment Cf_0, a *Logical Resource Base* (*LRB*) containing the defined *Deployment Units*, and the defined *Objectives O*:

$$D = (Cf_0, LRB, O). \qquad (12)$$

As we have explained over these sections, the correctness formula of any *Domain* is obtained just from these three models. It is composed of a set of restrictions that must be fulfilled, which can be individually evaluated.

3.7 Runtime Changes

The previous section presented the concept of *Domain*. From a management view, the *Domain* is a mutable entity, as all its elements (logical definitions, objectives, and runtime resources) can suffer modifications with the passing of time, potentially affecting the *Correctness*.

Depending on the nature of the initiating agent, domain changes can be classified as *external changes* (initiated by external entities) or *internal changes* (applied by the management system). The scope of both types of changes is also different: Internal changes can only modify the current configuration, whereas external changes can affect every element of the *Domain*. Examples include the definition of a new business objective or the release of an updated version of a component (that will appear at the *LRB*).

In this context, it becomes clear that for management purposes it is vital to provide a comprehensive identification of the internal changes as they define the scope of potential actions that can be applied by the management system. Table 1 provides a short overview of the 10 deployment and configuration primitives (internal changes) which we have deemed necessary for a service management system. For each operation, we detail the code name, the required arguments, and the changes it causes to the configuration when applied (including value changes and the existence, or not of the elements). The management system can only affect the topology of the Environment at the Runtime Unit level. It is possible to remove existing units (UNINSTALLUNIT) and instantiate new Runtime Units defined at the LRB (INSTALLUNIT, UPDATEUNIT). The configuration of the existing units can also be managed, with control over their lifecycle (STARTUNIT, STOPUNIT), as well as Property configuration (CONFUNITPROP) and Bindings (CONFBINDING). Containers can also be partially managed, in the form of Container Resource configuration changes (ADDCONTRES, RMVCONTRES, CONFCONTRES), allowing us to address Unit Constraints. Node management is out of scope.

The autonomic management control loop we intend to implement can be clearly defined using the concepts presented: After an external change happens to the *Domain* and is detected by the management system, it is diagnosed in order to evaluate if the *Correctness* has been broken. In that case, all the information will be analyzed in order to obtain a set of internal changes to the runtime configuration which will restore it to a correct state.

4 A Technique for Automated Service Change Identification

The previous section provided a complete definition of the service management problem. All the relevant management information has been defined based on a common model, and a function to evaluate the correctness of the domain has been derived from the model abstractions. Finally, the operations available to the management system to change the environment have been identified. After those concepts have been defined, we will propose in this section a technique

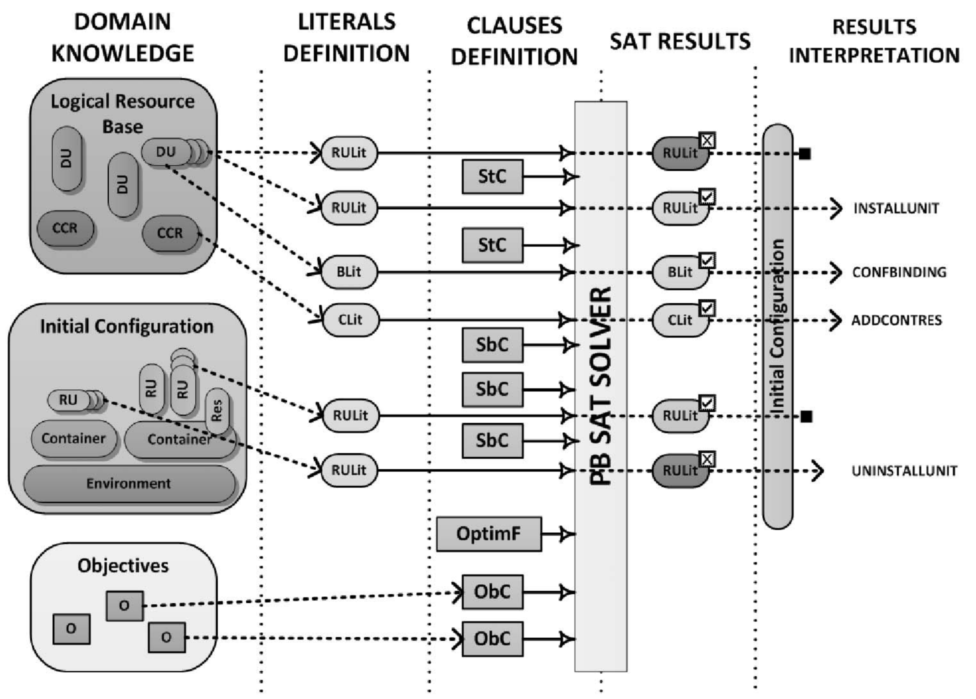

Fig. 5. The change identification process.

for automatically finding a correct solution by exploring the changes that can be applied to the environment.

Clearly brute-force approaches are not feasible in this context, as previous theoretical analysis have proven the NP-complete algorithmic complexity of the problem of finding a correct configuration for a distributed system [35]. On the other hand, we have already presented how SAT solvers have been successfully applied to problems of a similar nature, such as software packages dependency management, as well as exploring a configuration space with multiple defined constraints. Because of those factors, we have opted to express the problem of finding a reachable correct configuration as a satisfiability formula. Among the available variants, we have selected a Pseudo Boolean SAT solver as the base of our solution.

The main alternative to the selected approach was the previously described Alloy tool. Alloy offers a higher level abstraction over base SAT solvers, directly supporting the generation of model instances satisfying a set of first order constraints (equivalent to base SAT clauses) expressed over the model abstractions. However, for the purposes of our work, the use of a Pseudo Boolean SAT provides two important advantages: Linear clauses allow enforcing constraints such as value consumption. Second, and more importantly, the availability of an optimization function provides much greater influence over the proposed solution. For our context, minimizing the number of required changes was a fundamental factor for the applicability of the solution. As there were reports of industrial adoption of PB SAT Solvers for considerable large problem sizes with good results, we opted to explore that approach for our specific problem.

This way, the management problem is converted to the following SAT terms. Defined SAT variables represent every potential decision about the final environment configuration, with all of them being reachable by applying internal changes. Therefore, the true or false values of the proposed solution determine the final runtime configuration. The proposed solution must represent a coherent environment configuration, both stable and desirable. This will be enforced by defining SAT clauses over the variables. Finally, the Pseudo-Boolean optimization function will allow further control over the final result.

Fig. 5 shows a high-level representation of the complete change identification process, which receives the information models as input and returns the required set of changes for restoring the correctness of the environment. The technique can be divided into four steps: literals definition, clauses definition, SAT invocation, and results interpretation. Stability and desirability restrictions will be evaluated over different steps of the process (depending on its kind) in order to ensure that the proposed solution is correct.

The *literals definition* step takes as input the information from the logical deployment units and the current state of the environment, obtaining from them a set of SAT literals that represent all the potentially reachable configurations. After all the literals have been defined, the *clauses definition* step defines all the required clauses to ensure that the potential results from SAT execution will yield a stable and desirable solution. In order to do so, the literals and objectives are analyzed. Once the SAT solver has been completely configured, it will be invoked, obtaining a proposed final state of the environment. Finally, the *results interpretation* step will evaluate each clause value against the current environment state in order to obtain the required set of changes to be invoked by the management system.

Over the following sections, we will provide additional details over each one of the steps, presenting the techniques that, starting from the Domain information, generate the required input for the SAT solver (literals, clauses, and optimization Function), invoke the engine, and interpret the results as the set of required changes.

4.1 Literals Definition

The first step of the presented algorithm consists of defining SAT literals that capture every potential decision about the final environment state (e.g., what resources will be part of it and how will they be configured). The scope of these decisions is limited by the available management operations (the 10 primitives presented in the previous table). In order to identify these decisions, the available logical units and the current environment model must be analyzed. The defined deployment units determine what services can be instantiated over the environment, whereas several elements from the runtime also influence the potential actions. The runtime containers determine where units can be deployed over the environment, and provide a set of resources which can satisfy the defined constraints. Additionally, the configuration contains existing units and services which can be potentially removed. On the other hand, defined objectives are not relevant at this stage as they don't influence what configurations can be reached, they only evaluate their desirability.

After evaluating these concerns, the following types of SAT literals have been defined to capture the problem: *Runtime Unit Literals*, *RULit*, represent the decision of a runtime unit belonging or not to the final configuration (it might originate from the instantiation of a logical unit over a resource or it might exist initially). *Binding Literals*, *BLit*, represent the decision of configuring a specific binding of a Runtime Unit to another in order to satisfy the logical dependency. We use the term group of literals to identify all the *BLits* which belong to the same decision about a runtime *Binding*. Finally, the *Container Resource Literals*, *CLit*, represent the decision of creating or not a Resource at a runtime container in order to satisfy a runtime resource constraint.

Literals are obtained through a two-step algorithm. First, the *RULits* are obtained. In order to do so, the Deployment Units from the LRB are iterated over the Containers existing at the runtime configuration. This Cartesian product contains every Runtime Unit that can potentially appear at the environment by the actions of the management system. However, many of these Units would actually never be stable, as they would be deployed over incompatible Containers, or have Constraints that cannot be satisfied. In order to avoid that, several stability restrictions are applied at this stage. The algorithm checks the basic part of the Constraints (the name and type filter) of the Deployment Units against the execution context that would correspond to the Runtime Unit (the Container and its Resources, the Node and its Resources). Additionally, the algorithm checks the host compatibility restriction (ensuring that Container's supportedTypes include the unit's type), as well as the implicit restrictions derived from Context Aware Properties. This way, *RULits* that would represent unstable decisions are not defined, and *CLit* decisions are defined for those situations when the creation of a specific Resource at the Container can address unit stability.

The second step of the algorithm generates the *BLit* literals that contain the potential decisions about the Bindings established between the Runtime Units configurations. In order to do so, the *RULits* representing units with Dependencies are iterated against the complete set of *RULits*, obtaining every possibility. Similarly to the previous stage, stability restrictions are applied over the literals generation loop in order to discard unfeasible options. In particular, Dependency's bound resource compatibility (expressed by the name, version, and type filter, as well as the existence of the Properties requested by children Bound Property elements), and visibility of the potential bound resource are verified before defining the corresponding *BLit*. Additionally, *RULits* representing units with a dependency which cannot be solved will be removed from the SAT problem (as a true assignment for them would always result in an unstable solution).

The presented steps capture every potential decision about the final configuration topology (Units, Bindings, and Container Resources) which can result in a correct solution. Several stability restrictions are checked as part of the loops, imposing a slight overhead in the specific step. Nonetheless, applying them at this point not only reduces SAT problem size but also simplifies later stages of the process that iterate over previous results.

4.2 Clauses Definition

After identifying the types of SAT variables and the mechanism for its definition, we will describe the *clauses identification* step. From this point onward, we will focus on what restrictions need to be expressed to the values of the SAT literals in order to ensure that the proposed solution is correct. Restrictions are expressed through standard SAT clauses and pseudo-Boolean linear clauses (linear inequality with Boolean literals) [22]. Three categories of clauses will be introduced. *Structural clauses* enforce that the solution is coherent with the unit-resource structure (that is, the consistence of the values of related *RULit* and *BLit* variables). *Stability clauses* mandate this condition be met by the proposed solution. Finally, *Objective clauses* translate the defined objectives ensuring that the proposed solution is desirable.

Structural clauses link the values of a *RULit* and its associated *BLit* elements, as only one *BLit* from each group can be simultaneously true, and their value is also dependent on the existence or not of the Runtime Unit. These requirements will be expressed the following way. Let $\{A_i\}$, $A_i \in RULit$, be the set of defined *RULits* whose originating Deployment Unit declares n_i Dependencies, with $n_i. > 0$. Each Binding A_ib_j, $j \in [1, n_i]$, of the Runtime Unit represented by A_i can be potentially bound to m different units identified by the set of literals $\{B_k\}$, $B_k \in RULit$, $k \in [1, m]$. This way, the decision about the configuration of Binding A_ib_j is represented by the following set of literals: $\{A_ib_jB_k\}$, $A_ib_jB_k \in BLit$, $k \in [1, m]$. For each Binding A_ib_j of each *RULit* A_i the structure will be preserved by adding clauses (13) and (14) to the SAT:

$$A_i \leftrightarrow \bigvee_{k=1}^{m} A_ib_jB_k, \tag{13}$$

$$\sum_{k=1}^{m} A_i b_j B_k \leq 1. \tag{14}$$

Stability restrictions ensure that the decisions about Runtime Unit placement (*RULits*) and Binding configuration (*BLits*) at the final configuration respect the Dependency and Constraint restrictions. Property value restrictions from Context-Aware and Bound Properties need not to be expressed in the SAT problem as, instead of restricting valid decisions, they are directly derived from them. Therefore, they will be enforced at a later stage.

Dependency restrictions are expressed by adding clause (15) for each defined *BLit* element. This translates the SAT-ification of the simple dependency concept described in [25] to the current distributed problem through the use of bindings. As regards constraints addressable by the creation of new resources, a similar approach is followed for the basic restriction. Let $\{CR_i\}$, $CR_i \in CLit$, $i \in [1, n]$, the set of defined Container Resource Literals and $CU_i \in RULit$ the literal representing a unit whose Constraint will be satisfied by the creation of the resource represented by CR_i. Defining clause (16) for each *CLit* ensures the final configuration will create all the necessary Container Resources:

$$A_i b_j B_k \rightarrow B_k, \tag{15}$$

$$CU_i \rightarrow CR_i. \tag{16}$$

The presented definitions cover only the simple cases of Dependency and Constraint definitions. Additional clauses are defined to capture additional stability restrictions such as exclusive access, incompatibilities, or resource consumption that involve multiple units with conflicting requirements. As an example, we describe how to capture the consumption requirement whose function was presented at Section 3. Let a group of defined $RULits\{C_i\}$, $C_i \in RULit$, $i \in [1, n]$, representing units that consume the same resource (e.g., RAM memory from a node, as the unit depicted in Fig. 3). Each element consumes an amount $c_i \in \mathbb{N}$, as their Deployment Unit model contains a Constraint to a Resource of type *"hardware.memory.ram,"* with a Property Constraint of kind *CONSUMPTION* and value c_i (e.g., 256 for the unit presented in Fig. 3). If the value of the Resource property, representing total capacity is $cap \in \mathbb{N}$, the consumption restriction over the resource can be captured by adding a linear clause:

$$\sum_{i=1}^{n} c_i * \ C_i \leq cap. \tag{17}$$

Objective restrictions. Finally, desirability will be ensured by codifying the defined management objectives into clauses. Both types of objectives refer to one resource which must either exist or not exist at the final state of the runtime environment. The set of *RULits* providing the identified resource is called $\{OR_i\}$, $OR_i \in RULit$, $i \in [1, n]$. In the case of the EXISTS objective, at least one of the providers must appear at the solution, which is expressed by clause (18). On the other hand, NOTEXISTS objectives forbid the presence of that element. Therefore, no provider will appear at the solution, as is reflected in (19):

$$\bigvee_{i=1}^{n} OR_i, \tag{18}$$

$$OR_i = false, \ i \in [1, n]. \tag{19}$$

The definitions presented up to this point ensure that every proposed solution will be correct. However, as has been mentioned, this is insufficient for most scenarios. The solution should not only be correct but also enforce established management policies. These aspects can be expressed using additional Pseudo Boolean SAT restrictions and/or optimization functions. This allows the customization of aspects such as the number of desired instances of each resource, the preference in physically concentrating or distributing the services or minimizing the changes to the initial configuration as much as possible.

As an example of those customizations, we describe an optimization function that captures the objective of removing unneeded units and minimizing the number of changes to the runtime resources composition. This way, "purposeless" resources will be removed from the runtime configuration and, whenever possible, the proposed solution will respect the current state of the configuration while at the same time addressing the desirability and stability. In order to express this, the set of defined *RULits* must be partitioned into m groups, each of them composed of the literals instantiating the same Deployment Unit. These groups are: $\{A_{ij}\}$, $A_{ij} \in \ RULit$, $A_{ij}\ instance\ of\ Deployment\ Unit\ DU_j$, $i \in [1, n]$, $j \in [1, m]$.

For each group, if $n \geq 2$ and one of the literals A_{ij} represents an existing Runtime Unit, there is a potential SAT solution which does not include the already existing element. For those cases, a term GRP_j that expresses the preference for respecting the existing unit will be added to the optimization function, as is presented in

$$MINIMIZE\left(\sum_{i=1}^{m} GRP_j\right). \tag{20}$$

The GRP_j terms are defined as follows: For each group $\{A_{ij}\}$ of cardinality n, a linear expression will be defined from those literals with two different weights for the variables, as is shown in (21). The existing unit A_{xi} will have a weight $W_1 \in \mathbb{N}$, the rest having $W_2 \in \mathbb{N}$, with $W_2 > W_1$. The exact weights can be adjusted in order to prioritize among additional optimization criteria. This way, when the SAT minimizes the expression, the preferred option (in case it is correct) will be the nonexistence of every runtime unit from the group. If the unit is necessary for correctness, keeping the existing unit will be a better solution than any other alternative as the weight contribution will be lower:

$$GRP_j = W_1 * A_{xj} + \sum_{i=1}^{n-1} W_2 * A_{ij}, A_{xj} \in C_0, n \geq 2. \tag{21}$$

4.3 Results Interpretation

After all the literals and clauses have been identified, the SAT solver will be invoked, finding a solution (if possible) which consists of a proposed true or false assignment for each variable. The solution defines a correct environment state which can be obtained by applying internal changes (from the primitives presented in Table 1). The final step of the service change identification process will interpret the SAT results and obtain the changes which will be applied. As the value of each SAT literal represents a decision about one element of the final environment state (either a Container Resource, a Runtime Unit, or the configuration of a unit Binding), the required changes to apply each decision can be evaluated individually. Over this process, the remaining stability restrictions (related to Bound and Context-Aware Properties) are verified, ensuring the correctness of the final state. We will show how each type of literal is interpreted into changes to the runtime configuration by comparing them with the initial domain state.

CLit literals contain the decisions about the creation of additional resources on top of the existing Containers in order to satisfy unit constraints. Therefore, true values will be interpreted as *ADDCONTRES(cont, crc, name)* changes.

RULit literals determine what units and services will appear at the final configuration. In this case, both positive and negative assignments need to be checked, with its interpretation depending also on the initial environment state. True RULits demand the unit to appear, so if it was not initially deployed, it will be provisioned by the execution of *INSTALLUNIT(c,du)* and *STARTUNIT(c,ru)* changes. In case the literal is evaluated as false and the runtime unit was present at the initial configuration, it will have to be safely removed from the environment, by applying *STOPUNIT(c,ru)* followed by *UNINSTUNIT(c,ru)*. Finally, at this stage the stability restrictions from Context Aware Properties will be enforced. The required values for those Properties will be obtained from the final runtime state and will be reflected with a *CONFUNITPROP (cont,ru,props)* change.

The last set of literals (BLits) represents the decisions over the satisfaction of unit dependencies by establishing bindings among the runtime elements. For each positive value, a *CONFBINDING(ru,bindId,rub)* change will be applied. Finally, the internal unit configuration restrictions specified by Bound Properties will be computed, taking into account the decided Bindings and the source values. The required configuration for those properties will be specified through a *CONFUNITPROP(cont,ru,props)* change.

The list of changes contains every operation required for reaching the desired state, but it cannot be arbitrarily executed as there are partial ordering restrictions between the obtained changes. These dependencies are caused by two factors: Runtime Units have an execution life cycle, defined taking as reference the common ground of the main enterprise component specifications, such as Java Enterprise Edition's (JEE) Enterprise Java Beans (EJBs), or OSGi bundles. It includes provisioning, activation, stop, configuration, and uninstallation. The changes that can be applied to a unit are restricted by its life cycle state, demanding the definition of relative restrictions (i.e., the unit cannot be started before it has been installed, or the unit configuration must be completed before its activation). As the model aims at supporting heterogeneous components and services, *hot reconfiguration* is not allowed (applied while the component is active) [36]. Consequently, the life cycle mandates configuration operations to be applied only when a unit is stopped. Additionally, before starting a unit all its stability requirements (i.e., dependencies and constraints) must have been addressed. Therefore, before starting a runtime unit its internal configuration, bound units, and accessed container resources must have been left at a stable state.

These restrictions can be expressed in the following rules that, when applied, provide a partial ordering of the change activities.

For activities affecting the same runtime unit:

- INSTALLUNIT must be applied first.
- UNINSTALLUNIT must be applied last.
- STOPUNIT must be applied before every CONFUNITPROP and CONFBINDING activity.
- CONFUNITPROP changes which modify Bound Properties must be applied after the CONFBINDING changes.
- CONFUNITPROP and CONFBINDING must be applied before STARTUNIT.

Regarding activation of unstable units:

- ADDCONTRES and CONFIGRES must be applied before applying the constrained unit STARTUNIT and any CONFUNITPROP changes derived from Context-Aware Properties.
- STARTUNIT activities cannot be applied before every bound unit has already been started.

The partial order restrictions are obtained over the change generation process by applying these rules. The complete list of changes combined with their execution dependencies constitutes the change plan for the environment, which can be interpreted to reach the desired state.

5 Validation

The previous sections have detailed the technical foundations of our automated management engine. In this section, we present a set of validation experiments applied to an industrial case study from the banking domain.

The reference scenario for the set of validation experiments which will be presented is taken from the ITECBAN project. ITECBAN is a Spanish Research project from the CENIT program with academia and industry partners. The objective of this project is to propose a complete core banking solution based on the SOA/BPM paradigm. This way, the complete portfolio of the organization will be provided as services, including client services (internet banking, cashiers), internal services (for company workers at the bank offices), and B2B services for interbank transactions.

For these validation cases, there are 22 different artifacts which can be provisioned to the environment. Each artifact contains a manually defined deployment descriptor which describes its capabilities, as well as its logical and runtime execution requirements. Fig. 6 presents the 22 Deployment Units, as well as their declared dependencies. It can be seen how most of the elements are linked to one base element: the *Client Portal* end user Internet banking service. This service achieves its functionality by the composition of multiple lower level services, which will be distributed over the different runtime servers. The participating

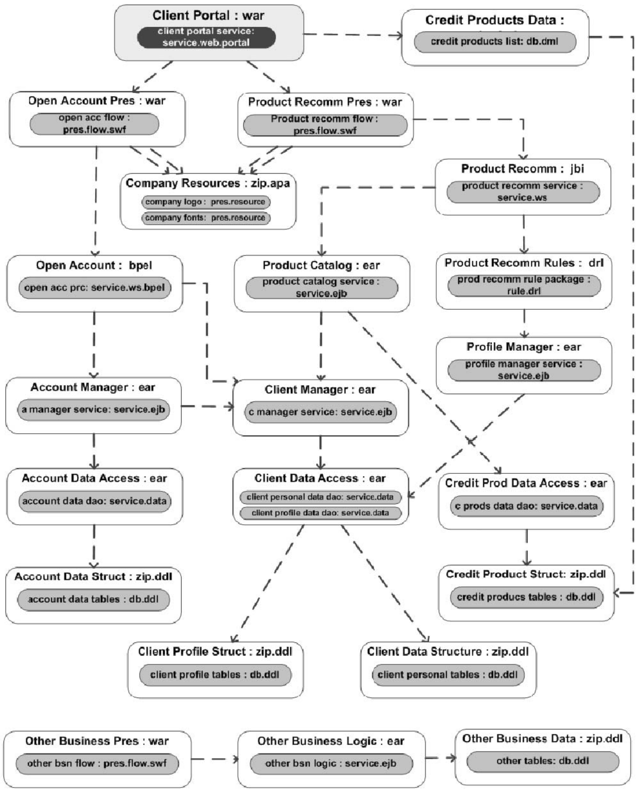
Client Portal : war
client portal service: service.web.portal
Credit Products Data :
credit products list: db.dml
Open Account Pres : war
open acc flow : pres.flow.swf
Product Recomm Pres : war
Product recomm flow : pres.flow.swf
Company Resources : zip.apa
company logo : pres.resource
company fonts: pres.resource
Product Recomm : jbi
product recomm service : service.ws
Open Account : bpel
open acc prc: service.ws.bpel
Product Catalog : ear
product catalog service : service.ejb
Product Recomm Rules : drl
prod recomm rule package : rule.drl
Profile Manager : ear
profile manager service : service.ejb
Account Manager : ear
a manager service: service.ejb
Client Manager : ear
c manager service: service.ejb
Account Data Access : ear
account data dao: service.data
Client Data Access : ear
client personal data dao: service.data
client profile data dao: service.data
Credit Prod Data Access : ear
c prods data dao: service.data
Account Data Struct : zip.ddl
account data tables : db.ddl
Credit Product Struct: zip.ddl
credit producs tables : db.ddl
Client Profile Struct : zip.ddl
client profile tables : db.ddl
Client Data Structure : zip.ddl
client personal tables : db.ddl
Other Business Pres : war
other bsn flow : pres.flow.swf
Other Business Logic : ear
other bsn logic : service.ejb
Other Business Data : zip.ddl
other tables: db.ddl

Fig. 6. Dependency graph of the client portal service.

resources represent business processes, business rules, service components, information from the database management systems, remote functionality exposed through web services, or corporate image material such as logos and fonts. The resource and typing mechanism allows capturing the characteristics and differences of those elements. Therefore, in order for the *Client Portal* service to work correctly, 19 different, distributed elements will have to be properly configured and deployed, making this scenario a good candidate for validating the proposed models and architecture.

These elements run at a complex service execution platform composed of multiple application servers, Business Rule Manager (BRM) servers, business process servers, mediation servers, databases, and additional infrastructure required to provide the aforementioned functionality with the adequate degrees of efficiency and robustness. Fig. 7 shows the main elements of the validation runtime environment. It is composed of 10 nodes, hosting 12 different containers. The picture also shows the type and supported artifacts for each container. It can be seen how some of them are replicated, whereas others have a single instance.

In this context, we have developed a prototype service management system. The main functionality is provided by a core set of services, which are domain agnostic as they operate at model level. The framework described in this paper is implemented by the Change Identification Service, which analyzes the environment state and proposes a set of changes using the previously described satisfiability-based approach. The current implementation uses the EMF Runtime v2.5.0 [38] for the de/serialization of model

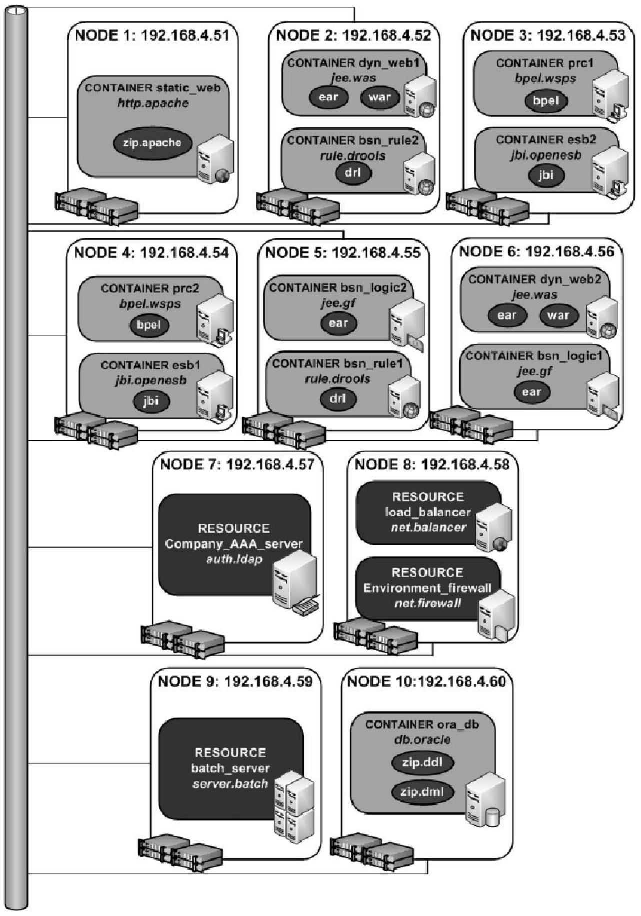

Fig. 7. Validation runtime environment.

instances, and the SAT4j 2.1 Pseudo Boolean SAT engine [26] for the implementation of the algorithm.

The interaction between the model-based management services and the physical elements that constitute the runtime infrastructure is handled by an agent infrastructure. Execution platform elements are instrumented by agents that interact with the management APIs of the runtime servers (e.g., JMX JSR 77 MBeans for Java Enterprise Edition servers). Agents translate the specific information into runtime model objects (Nodes, Containers, Runtime Units, and Resources) that reflect the current environment state. Agents also close the control loop by translating the obtained changes to container-specific management operations. Additional details about its internal architecture, as well as the mechanisms for automatically adapting to the heterogeneous environment and performing the adaptation are presented in [37].

5.1 Case Study Execution Results

Once the basic context for the validation cases has been introduced, we will describe several scenarios which illustrate the reasoning and reaction capabilities of our proposal for a wide range of situations.

5.1.1 Initial Environment Provisioning

The first scenario consists of a freshly installed environment, ready to start functioning after the hardware elements have been provisioned with the required servers

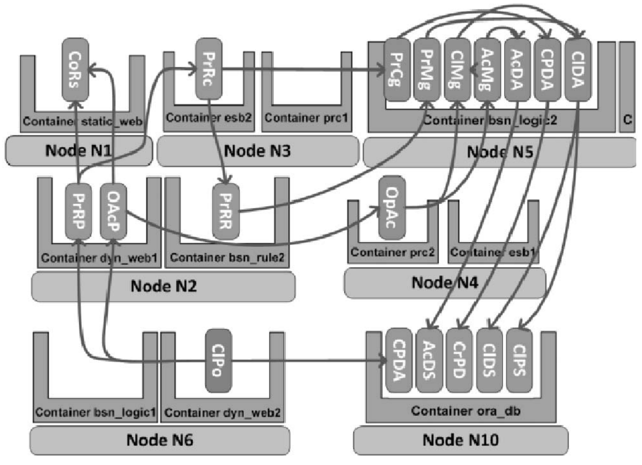

Fig. 8. Environment state after first scenario execution.

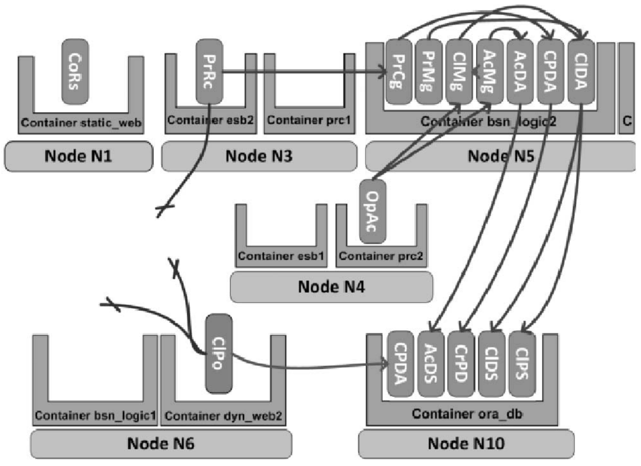

Fig. 9. Environment state after a malfunction in node N2.

and services. In order for it to start providing the desired functionality, a management objective is defined demanding that the *Client Portal* service be present at the environment. In order to correctly address this problem, the change identification process must install the unit providing that service as well as all its required units in compatible runtime containers. Each of those elements must also be correctly configured, both the bindings and properties' values. On top of that, each unit constraints must also be respected. Whereas none of these changes are explicitly expressed by the initial input, they must appear at the set of changes obtained by the execution algorithm.

When applying the previous algorithm, the scenario information is converted to a SAT problem composed by 145 literals and 280 clauses. The execution of the satisfiability engine found a solution with 45 variables assigned to true, which in its interpretation yielded a total of 70 required changes. Thirty-eight are derived from the positive RU literals (installation and activation), 23 from B literals, three from the CR literals, and the remaining six are related to configurable properties. The changes were handled by the agents infrastructure, which interpreted them into platform-specific commands. As an example, the INSTALLUNIT change to provision the Client Data Access unit (packaged as an ear) to the Glassfish container bsn_logic2 located at node 5 is translated by the agent into a JSR 88 "DeploymentManager.redeploy" instruction, invoked through the remote MBean offered by the Glassfish cluster manager. After applying all those changes, the 19 participating units have been deployed over eight containers and seven nodes from the total of 12 containers and 10 nodes belonging to the environment. The final runtime state is shown in Fig. 8.

In order to test the minimum change policy enforced by the optimization function, a false positive test was executed. The scenario included the same objective (Existence of the *Client Portal* service) and an environment configured with a correct configuration (the same 19 units deployed), but a different distribution over the runtime containers. The SAT arguments were similar to the previous case (with three less literals because of the already satisfied constraints), but the result did not identify any required change. This shows that the presented technique respects the existing state of the runtime, avoiding the application of unnecessary changes.

5.1.2 Service Decommission

The next scenario complements the previous ones, bringing a fundamental change: Starting with the environment state reached after the first scenario (with the 19 units correctly configured), the business objective mandating the presence of the client service is removed. This way, none of the available runtime units are contributing to the desirability of the current configuration, so they could be safely removed, as is enforced by the previously presented optimization function.

In this case, the execution resulted in no literal from the defined 142 being evaluated to true. This was interpreted into 38 change activities, consisting of stopping and uninstalling all the Runtime Units. As the *Client Portal* service is no longer desired, the change identification service correctly decides to remove it and all the related services from the environment.

5.1.3 Hardware Malfunction

The next scenario shows the self-healing capabilities of the presented engine showing how the Change Identification Service can react to unstable environment configurations. The environment is initially provisioned with the complete set of services, but a hardware malfunction brings down Node N2. The updated environment snapshot can be seen in Fig. 9. In total, two containers have disappeared (the JEE application server dyn_web1 and the BRM server bsn_rule2) and the three hosted runtime units are missing. The remaining elements are shown in the picture, with the broken bindings represented as loose connections on a different tone. With that initial input, the Change Identification Service is invoked to try to find an updated, correct state.

The first noticeable difference over the execution of this test is that the number of SAT literals has been reduced to 113. This decrement is caused by the removal of two containers, reducing the distribution options for two types of units. However, the same number of positive variables has been obtained, as all of them are required for a desirable and stable configuration. After the variables have been analyzed, a total of 28 changes have been identified. The set of changes consists of deploying the missing three components to the only possible options (the WAR units to the application server dyn_web2, and the business rule

TABLE 2
Case Study Execution Statistics

Input data statistics	**Case I**	**Case II**	**Case III**	**Case IV**	**Case V**	**Case VI**	**Case VII**	**Case VIII**
Number of units	22	22	22	41	121	1001	41	121
Number of dependencies	25	25	25	43	123	1003	43	123
Number of constraints	3	3	3	3	3	3	3	3
Number of nodes	10	30	90	10	10	10	30	90
Number of containers	12	36	108	12	12	12	36	108
SAT Problem Statistics	**Case I**	**Case II**	**Case III**	**Case IV**	**Case V**	**Case VI**	**Case VII**	**Case VIII**
Number of variables	145	963	7673	509	2109	19709	3783	138245
RuntimeUnit variables	43	129	387	119	439	3959	357	3951
Binding variables	96	816	7232	384	1664	15744	3408	134240
ConfRes variables	6	18	54	6	6	6	18	54
Number of functions	280	1952	15452	1000	4200	39400	7568	276524
Time and memory statistics	**Case I**	**Case II**	**Case III**	**Case IV**	**Case V**	**Case VI**	**Case VII**	**Case VIII**
Consumed memory	6.2MB	7,6MB	18,5MB	7.0MB	10.3MB	51.7MB	11.9MB	204.3MB
Total exec time	83ms	265ms	5892ms	160ms	657ms	40698ms	1598ms	2611720ms
Preparation time	59ms	206ms	5350ms	126ms	598ms	40426ms	1501ms	2609667ms
Execution time	18ms	42ms	246ms	27ms	41ms	155ms	63ms	1168ms
Interpretation time	6ms	17ms	296ms	7ms	18ms	117ms	34ms	885ms

artifact unit to the BRM container *bsn_rule*2), configuring both broken bindings to those units (initially stopping the broken units for proper configuration), as well as the bindings and bound properties from these three newly created units.

This case shows how the proposed algorithm reacts to unexpected changes to the runtime environment and restores the intended system functionality. The proposed solution also reuses the already available runtime units, in order to minimize the set of required changes to the environment.

The set of presented scenarios show the automated reasoning capabilities of the presented solution, with scenarios ranging from initial deployment to runtime adaptation being supported by the same abstractions and reasoning mechanisms.

5.2 Scalability Analysis

In addition to the previous experiments, we present the results of an additional set of cases which aim to test the scalability of the proposed algorithm. These tests build on top of the first scenario, starting with an empty environment, and the same defined objective (existence of the Client Portal Service). The variations among them will be located only in the knowledge base, increasing the number of deployment units at the logical repository and the size of the runtime environment.

A total of eight additional tests have been executed. Starting from the initial problem (Case I), there are three additional experiments (II, III, IV) increasing only the size of the environment (by replicating nodes and containers with a different name). Cases V and VI modify only the set of logical units, adding additional elements with service dependencies unrelated to the solution. Finally, Cases VII and VIII simultaneously increase both sets in order to see how their interference affects the problem complexity.

The execution of each of those cases did obtain a correct result (identifying 70 changes for deploying and configuring the service). Table 2 shows detailed results of each experiment execution. For each case, statistics are provided at three different stages of the process. The first set of elements informs about the size of the input models, which differentiates one case from the rest. The second set provides internal information about how the problem has been interpreted as SAT, informing about the number of literals and clauses. Finally, the last category shows the observed consumption in both time and memory for each execution, obtained from an average of five executions. Specific execution time is provided for the two stages composing model processing and SAT problem definition (literals and clauses). Tests were executed over a standard desktop PC (2 GB of RAM, 2 GHz Dual Core Processor).

The execution statistics show the interference of increasing both logical and runtime size simultaneously, with the biggest one providing a rough estimation of the limits of the algorithm implementation, with more than 200 Mbytes of problem size and a total execution time of 43 minutes. Nonetheless, it has been shown that the Change Identification Service is able to provide a solution for problems with a size of more than a hundred thousand variables and double that amount of functions, considerably more complex than the initial case, which already reflected an industrial case.

In case larger logical knowledge bases need to be supported, there is a simple way to greatly improve these cases by adding an initial filtering step. If the results of Case VIII are compared with case V—the one of equal environment size and reduced logical base—it can be seen that it takes almost 4,000 times more because of the unrelated units, which are very costly to process and could actually be discarded. Therefore, it is possible to apply an initial filter over the complete logical knowledge base by performing a dependency resolution process over the logical elements, such as the ones described at [25] or [26], for every resource mentioned at either the objectives or present current runtime state. The remaining logical elements can safely be discarded for the presented process, and the complete efficiency of the algorithm will experience a considerable improvement.

6 Conclusions and Future Work

In this paper, we have presented a method for automating management operations which provides self-configuration capabilities over the services infrastructure. Our work was motivated by the practical problem of deploying and configuring complex distributed heterogeneous services in industrial settings, particularly supporting change operations on the software components that provide the services.

These components show interdependencies among them and with the logical and physical environment they run on. Our contribution to the automation of these operations is based on three elements. First, we have defined a model covering all the information required for automating the management of the system, including the means to describe the system and diagnose its correctness (through the stability and desirability formulas). Second, we have described a satisfiability-based engine that can diagnose the health of any given configuration, and in case it is incorrect, explore the potential solutions and propose the required changes for reaching a new, correct state. Third, we have presented a mechanism for reconfiguring the runtime system through the application of the identified changes.

The presented approach is not tailored to specific use cases (initial deployment, service update, unit configuration), supporting instead a generic process consisting of evaluating the correctness of the environment starting state and obtaining the set of required changes for restoring the domain to a correct state. The flexibility of the proposed solution is supported by some of the original aspects of this work: the complete characterization of the managed system information (including the runtime state, the available logical definitions which can be instantiated, and the defined objectives for the environment) and the definition of stability and desirability conditions which can be evaluated from the modeled information.

The approach has been validated using several complex mid and large-size case studies in the banking domain. They represent compound enterprise services built with off-the-shelf and open source SOA-BPM components, which must be provisioned over a distributed infrastructure pertaining to a single organization. While building the proof of concept for the autonomous engine, we have used our models for describing the logical deployment units (services components), modeled the objectives (desired configurations) for the complete environment, and developed technology specific agents which extract the current services platform configuration and present it based on the runtime model. The algorithm has been implemented using an available Pseudo Boolean SAT engine (SAT4j) which offers good exploration time values and efficient memory consumption. The results from the experiments lead us to think this same platform could be used in production settings.

However, there are still some aspects that could be improved in our future work: We intend to extend the scope of the supported automated reconfiguration activities in order to support the self-optimization of quantifiable values, such as service level agreements. Also, by applying domain-specific strategies to the presented algorithm, we can reduce the search space in order to ensure the engine operates on certain time thresholds, improving its usefulness as a control loop for self-healing functions.

Furthermore, so far we have considered the physical and logical platforms as invariants in the calculation of stable configurations, but we believe that these base platforms could be controlled by the management engine, thus leading to a solution of the "initial provisioning" problem. A further evolution of this line would allow the application of this engine for the management of a Cloud Computing infrastructure. This application would require some modifications, as the problem is at the same time larger that the reference scenarios (in number of elements) but is simpler (as the infrastructure tends to be much more homogeneous).

There are also improvements to the practical application of this work that should be tackled to face the large variation in the problem space of services deployment and configuration, including the solution we propose into the operational processes in large organizations, supporting interorganization management processes, or increasing the range of instrumentation agents.

Acknowledgments

The authors wish to thank the anonymous reviewers for their insightful comments to the previous versions of this paper. This work was partially supported by the ITECBAN project from the CENIT program of the Ministerio de Industria, Turismo y Comercio (MITYC) and the Centro para el Desarrollo Tecnológico Industrial (CDTI).

References

[1] T. Erl, *Service-Oriented Architecture: Concepts, Technology and Design.* Prentice-Hall, 2005.
[2] N. Josuitis, *SOA in Practice: The Art of Distributed Design.* O' Reilly Media, 2007.
[3] Q. Gu and P. Lago, "Exploring Service-Oriented System Engineering Challenges: A Systematic Literature Review," *Service-Oriented Computing and Applications,* vol. 3, no. 3, pp. 171-188, Sept. 2009.
[4] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl, "A Journey to Highly Dynamic, Self-Adaptive, Service-Based Applications," *Automated Software Eng.,* vol. 15, pp. 313-341, 2008, DOI: 10.1007/s10515-008-0032-x.
[5] J.O. Kephart and D.M. Chess, "The Vision of Autonomic Computing," *Computer,* vol. 36, no. 1, pp. 41-50, Jan. 2003.
[6] M.P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B.J. Krämer, "Service-Oriented Computing: A Research Roadmap," *Int'l J. Cooperative Information Systems,* vol. 17, no. 2 pp. 223-255, 2008.
[7] *An Architectural Blueprint for Autonomic Computing,* fourth ed. Computing IBMA, June 2006.
[8] D. Agrawal, L. Kang-Won, and J. Lobo, "Policy-Based Management of Networked Computing Systems," *IEEE Comm. Magazine,* vol. 43, no. 10, pp. 69-75, Oct. 2005.
[9] B. Jennings, S. van der Meer, S. Balasubramaniam, D. Botvich, M.O. Foghlu, and W. Donnelly, "Towards Autonomic Management of Communications Networks," *IEEE Comm. Magazine,* vol. 45, no. 10, pp. 112-121, Oct. 2007.
[10] M. Desertot, C. Escoffier, P. Lalanda, and D. Donsez, "Autonomic Management of Edge Servers," *Proc. First Int'l Workshop Self-Organizing Systems,* pp. 216-229, 2006.
[11] J.E. López de Vergara, V.A. Villagrá, C. Fadón, J.M. González, J.A. Lozano, and M.A. Álvarez-Campana, "An Autonomic Approach to Offer Services in OSGi-Based Home Gateways," *Computer Comm.,* vol. 31, no. 13, pp. 3049-3058, Aug. 2008.

[12] J. Dubus and P. Merle, "Applying OMG D&C Specification and ECA Rules for Autonomous Distributed Component-Based Systems," *Models in Software Eng.*, pp. 242-251, 2007.

[13] F. Kon, J.R. Marques, T. Yamane, R.H. Campbell, and M.D. Mickunas, "Design, Implementation and Performance of an Automatic Configuration Service for Distributed Component Systems," *Software Practice and Experience,* vol. 35, pp. 667-703, 2005.

[14] DMTF (Distributed Management Task Force), *Common Information Model (CIM) Specification v2.27,* 2010.

[15] Object Management Group, Deployment and Configuration of Distributed Component-Based Applications Specification, Version 4.0, Apr. 2006.

[16] H. Kreger, W. Vambenepe, I. Sedukhin, S. Graham, and B. Murray, "Management Using Web Services: A Proposed Architecture and Roadmap," technical report, IBM, HP, and Computer Assoc., 2005.

[17] Y. Cheng, "Toward an Autonomic Service Management Framework: A Holistic Vision of SOA, AON, and Autonomic Computing," *IEEE Comm. Magazine,* vol. 46, no. 5, pp. 138-146, May 2008.

[18] W. Vambenepe and W. Bullard, "Web Services Distributed Management: Management Using Web Services," (MUWS 1.1) Part 1. OASIS Standard, Sept. 2006.

[19] K. Wilson and I. Sedukhin, *Web Services Distributed Management: Management of Web Services (MOWS 1.1),* OASIS Standard, Aug. 2006.

[20] B. Miller, J. McCarthy, R. Dickau, and M. Jensen, OASIS Solution Deployment Descriptor (SDD), 1.0, OASIS Standard, Sept. 2008.

[21] J. Marques-Silva, "Practical Applications of Boolean Satisfiability," *Proc. Int'l Workshop Discrete Event Systems,* May 2008.

[22] A. Biere, M. Heule, H. Van Maaren, and T. Walsh, *Handbook of Satisfiability,* Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, 2009.

[23] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem-Proving," *J. ACM Comm.,* vol. 5, no. 7, pp. 394-397, 1962.

[24] M. Prasad, A. Biere, and A. Gupta, "A Survey of Recent Advances in SAT-Based Formal Verification," *Int'l J. Software Tools for Technology Transfer,* vol. 7, no. 2, pp. 156-173, Apr. 2005.

[25] C. Tucker, D. Shuffleton, R. Jhala, and S. Lerner, "OPIUM: Optimal Package Install Uninstall Manager," *Proc. 29th Int'l Software Eng. Conf.,* 2007.

[26] D. Le Berre and P. Rapicault, "Dependency Management for the Eclipse Ecosystem. Eclipse p2, Metadata and Resolution," *Proc. Int'l Workshop Open Component Ecosystems,* Aug. 2009.

[27] D. Le Berre and A. Parrain, "On SAT Technologies for Dependency Management and Beyond," *Proc. First Workshop Analyses of Software Product Lines,* Sept. 2008.

[28] S. Hallé, E. Wenaas, R. Villemaire, and O. Cherkaoui, "Self-Configuration of Network Devices with Configuration Logic," *Autonomic Networking,* Springer, 2006.

[29] S. Narain, "Network Configuration Management via Model Finding," *Proc. the 19th Large Installation System Administration Conf.,* Dec. 2005.

[30] H. Foster, S. Uchitel, J. Kramer, and J. Magee, "Towards Self-Management in Service-Oriented Computing with Modes," *Proc. Workshop Eng. Service-Oriented Applications,* pp. 338-350, 2009.

[31] M.B. Cohen, M.B. Dwyer, and J. Shi, "Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach," *IEEE Trans. Software Eng.,* vol. 34, no. 5, pp. 633-650, Sept./Oct. 2008.

[32] H. Kreger and T. Studwell, "Autonomic Computing and Web Services Distributed Management," IBM DeveloperWorks, June 2005.

[33] J.C. Dueñas, J.L. Ruiz, F. Cuadrado, B. García, and H.A. Parada, "System Virtualization Tools to the Rescue of Software Developers," *IEEE Internet Computing,* vol. 13, no. 5, pp. 52-59, Sept. 2009.

[34] J.L. Ruiz, J.C. Dueñas, and F. Cuadrado, "Model-Based Context-Aware Deployment of Distributed Systems," *IEEE Comm. Magazine,* vol. 47, no. 6, pp. 164-171, June 2009.

[35] M. Burgess and L. Kristiansen, "On the Complexity of Change and Configuration Management," *Handbook of Network and System Administration,* pp. 567-622, Elsevier, 2007.

[36] J. Matevska, W. Hasselbring, and R.-H. Reussner, "Software Architecture Description Supporting Component Deployment and System Runtime Reconfiguration," *Proc. 33rd EUROMICRO Conf. Software Eng. and Advanced Applications,* 2007.

[37] F. Cuadrado, J.C. Dueñas, R. García, and J.L. Ruiz, "A Model for Enabling Context-Adapted Deployment and Configuration Operations for the Banking Environment," *Proc. Fifth Int'l Conf. Networking and Services,* Apr. 2009.

[38] D. Steinberg, F. Budinsky, F. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework,* second ed. Addison-Wesley, 2008.

Félix Cuadrado received the degree in telecommunications engineering from the Universidad Politécnica de Madrid (UPM), Spain, 2005, and the PhD degree from UPM in 2009. He is a lecturer (assistant professor) in the School of Electronic Engineering and Computer Science, Queen Mary University of London. His research interests include autonomic computing, services engineering, and cloud computing. He is a member of the IEEE.

Juan C. Dueñas received the degree in telecommunications engineering from Universidad Politécnica de Madrid (UPM), Spain, 1990, and the PhD degree from UPM in 1994. He is a professor in the Escuela Técnica Superior de Ingenieros de Telecomunicación at UPM. He is president of the Spanish Chapter of the IEEE Computer Society. His research interests include services engineering, Internet services, service oriented architectures, and software engineering. He is a member of the IEEE Computer Society.

Rodrigo García-Carmona received the degree in telecommunications engineering from the Universidad Politécnica de Madrid (UPM), Spain, in 2007 and is currently working toward the PhD degree in the telematics engineering program at UPM. He is a researcher in the Escuela Técnica Superior de Ingenieros de Telecomunicación at UPM. His research interests include services engineering and model-based engineering. He is a member of the IEEE Computer Society.