Formal rules for reliable component-based architecture evolution

Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, Zhang Huaxi Yulin

▶ **To cite this version:**

Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, Zhang Huaxi Yulin. Formal rules for reliable component-based architecture evolution. FACS: Formal Aspects of Component Software, Sep 2014, Bertinoro, Italy. pp.127-142, 10.1007/978-3-319-15317-9_8 . hal-01245857

HAL Id: hal-01245857
https://hal.science/hal-01245857

Submitted on 20 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Formal rules for reliable component-based architecture evolution

Abderrahman Mokni[+], Marianne Huchard*, Christelle Urtado[+], Sylvain Vauttier[+], and Huaxi (Yulin) Zhang[‡]

[+]LGI2P, Ecole Nationale Supérieure des Mines Alès, Nîmes, France
*LIRMM, CNRS and Université de Montpellier 2, Montpellier, France
[‡] Laboratoire MIS, Université de Picardie Jules Verne, Amiens, France
{Abderrahman.Mokni, Christelle.Urtado, Sylvain.Vauttier}@mines-ales.fr,
huchard@lirmm.fr, yulin.zhang@u-picardie.fr

Abstract. Software architectures are the blueprint of software systems construction and evolution. During the overall software lifecycle, several changes of its architecture may be considered (*e.g.* including new software requirements, correcting bugs, enhancing software performance). To ensure a valid and reliable evolution, software architecture changes must be captured, verified and validated at an early stage of the software evolution process. In this paper, we address this issue by proposing a set of evolution rules for software architectures in a manner that preserves consistency and coherence between abstraction levels. The rules are specified in the B formal language and applied to a three-level ADL that covers the three steps of software development: specification, implementation and deployment. To validate our rules, the approach is tested on a running example of Home Automation Software.

Keywords: software architecture evolution, component reuse, consistency checking, coherence checking, evolution rules, formal models, abstraction level, B formal language

1 Introduction

The great importance of evolution and maintenance in software systems engineering has been noticed over more than two decades ago. According to a highly cited survey conducted by Lientz and Swanson in the late 1970s [1], it has been proven that software maintenance represents the main part of a software lifecycle in terms of cost and time. In particular, this high fraction relates to component-based software engineering that tackles the development of complex software architectures (thanks to modularity, abstraction and reuse). Indeed, an ill mastered software system maintenance or a misconception during its evolving process may lead to serious architectural mismatches and inconsistencies. A famous problem that software architecture evolution is subject to is erosion. Introduced by Perry and Wolf [2] in 1992 and studied over many years [3], erosion can be defined as the deterioration or violation of architectural design decisions by the

software implementation. It is usually due to software aging and an undisciplined evolution of its architecture. While a lot of work was dedicated to architectural modeling and evolution, there is still a lack of means and techniques to tackle architectural inconsistencies, and erosion in particular. Indeed, almost existing ADLs hardly support the whole life-cycle of a component-based software and it often creates a gap between design and implementation, requirements and design or even both. These gaps make evolution harder and increase the risk of non-conformance between requirements, design and implementation hence leading to erosion.

In previous work [4,5], we proposed Dedal, an ADL that supports the full life-cycle process of component-based software systems. Dedal proposes to model architectures at three abstraction levels that correspond to the three steps of software development: specification, implementation and deployment. However, at this stage the ADL handles evolution in an *adhoc* manner and lacks rigorous support for reliable and automatic software evolution. In this paper, we propose a set of evolution rules specified using the B formal language [6] to automatically handle forward and reverse evolution among Dedal levels in a reliable way. We also show how evolution can be simulated at an early stage using the proposed rules, anticipating and preventing inconsistencies.

The remainder of this paper is outlined as follows: Section 2 discusses related work. Section 3 gives a brief overview of Dedal architecture levels and their formalization. Section 4 presents the three-level evolution approach, illustrated by some evolution rules. Section 5 gives the simulation of an evolution scenario example using the proposed rules. Finally, Section 6 concludes the paper and discusses future work.

2 Related work

Over the two past decades, a wide area of related work has addressed the problem of software evolution. Indeed, many ADLs have been proposed [7]. Examples include C2SADL [8], Wright [9], Rapide [10], ACME [11], Darwin [12] and π-ADL [13]. While "box-and-line" seems to be the easiest way to represent architectures for practitioners [14], this notation is informal and leads to ambiguity and imprecision. For this reason, the use of a formalism and its integration into an ADL is crucial. To cope with software evolution and particularly dynamic change, existing ADLs use several kinds of formal ground depending on their application domain. For instance, C2SADL uses event-based processes to model concurrent systems while Dynamic-Wright lies on CSP [15], a process algebra formalism to support the specification and analysis of interactions between components. ACME, which was basically designed to define a common interchange language for architecture design tools, is based on first-order logic. The ADL was extended with Plastik [16] to support dynamic reconfiguration of architectures. π-ADL was designed for concurrent and mobile systems. It lies on π-calculus [17], a higher-order logic formalism to model and evolve the behavior of the architectures. C2SADL, Pi-ADL, ACME and Dynamic-Wright support dynamic recon-

figuration of architectures. However, they lack analysis support for the evolution activity and hardly cover the whole lifecycle of component-based software.
In our work, we propose a solution for the simulation and verification of software architecture evolution using B [6] formal models. The choice of B is motivated by its rigorism (first-order logic) and its expressiveness that enables modeling concepts in a reasonable easy way. The B formal models correspond to the definitions of our three-level Dedal ADL that covers the whole lifecycle of a software system (*i.e.* specification, implementation and deployment). Hence, we address both static and dynamic evolution by proposing change rules at each of the three abstraction levels of our ADL.

3 Overview of Dedal

3.1 Dedal abstraction levels

Dedal is a novel ADL that covers the whole life-cycle of a component-based software. It proposes a three-step approach for specifying, implementing and deploying software architectures as a reuse-based process(*cf.* Figure 1).

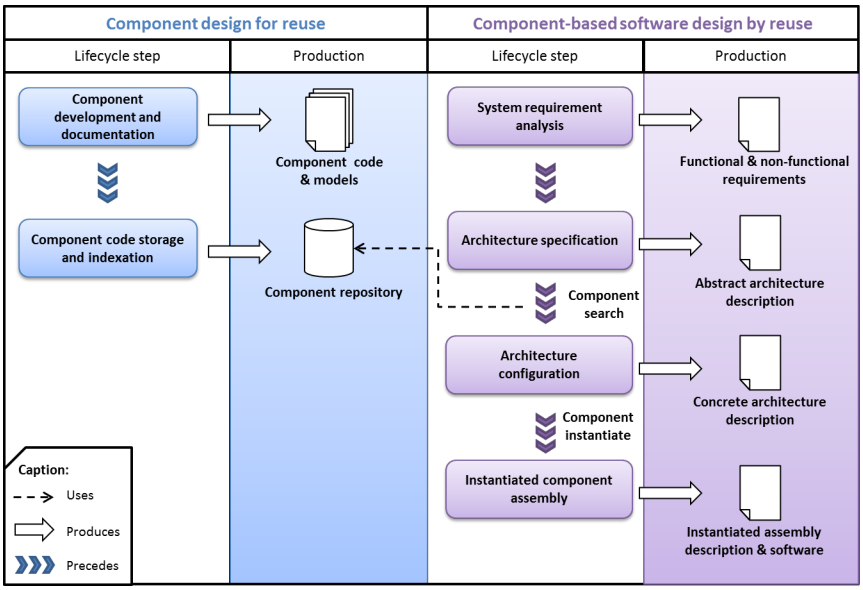

Fig. 1. Dedal overall process [5]

To illustrate the concepts of Dedal, we propose to model a home automation software (HAS) that manages comfort scenarios. Here, it automatically controls the building's lighting and heating in function of the time and ambient temperature.

For this purpose, we propose an architecture with an orchestrator component that interacts with the appropriate devices to implement the desired scenario.

The abstract architecture specification is the first level of software architecture descriptions. It represents the architecture as imagined by the architect to meet the requirements of the future software. In Dedal, the architecture specification is composed of component roles, their connections and the expected global behavior. Component roles are abstract and partial component type specifications. They are identified by the architect in order to search for and select corresponding concrete components in the next step. Figure 2-a shows a possible architecture specification for the HAS. In this specification, five component roles are identified. A component playing the *Orchestrator* role controls four components playing the *Light*, *Time*, *Thermometer* and *CoolerHeater* roles.

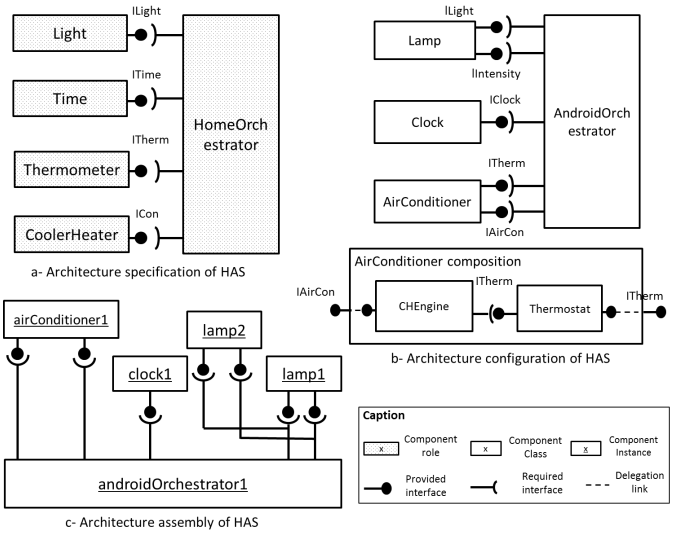

Fig. 2. Architecture specification, configuration and assembly of HAS

The concrete architecture configuration is an implementation view of software architectures. It results from the selection of existing component classes in component repositories. Thus, an architecture configuration lists the concrete component classes that compose a specific version of the software system. In Dedal, component classes can be either primitive or composite. *A primitive component class* encapsulates executable code. *A composite component class* encapsulates an inner architecture configuration (*i.e.* a set of connected component classes which may, in turn, be primitive or composite). A composite component class exposes a set of interfaces corresponding to unconnected interfaces of its inner components.

Figure 2-b shows a possible architecture configuration for the HAS example as well as an example of an *AirConditioner* composite component and its inner configuration. As illustrated in this example, a single component class may realize several roles in the architecture specification as with the *AirConditioner* component class, which realizes both *Thermometer* and *CoolerHeater* roles. Conversely, a component class may provide more services than those listed in the architecture specification as with the *Lamp* component class which, provides an extra service to control the intensity of light.
The instantiated architecture assembly describes software at runtime and gathers information about its internal state. The architecture assembly results from the instantiation of an architecture configuration. It lists the instances of the component and connector classes that compose the deployed architecture at runtime and their assembly constraints (such as maximum numbers of allowed instances). *Component instances* document how component classes in an architecture configuration are instantiated in the deployed software. Each component instance has an initial and current state defined by a list of valued attributes. Figure 2-c shows an instantiated architecture assembly for the HAS example.

3.2 Dedal formal model

Dedal is enhanced by a formal model using the B specification language. The proposed model covers all Dedal concepts and includes rules for substitutability and compatibility among each level as well as the rules that govern interrelations between the different levels (*cf.* Figure 3). These rules, which were discussed in previous work [18], are the basis for controlling the evolution process. Indeed, evolution needs a subtyping mechanism to manage change locally (at the same abstraction level) and then, inter-level rules to propagate change to the other levels.

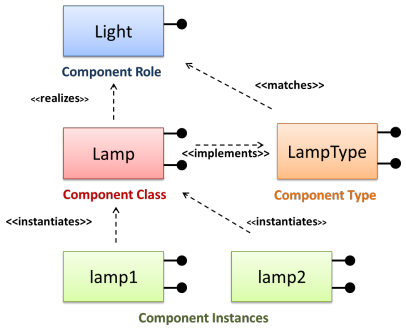

Fig. 3. Component interrelations in Dedal

For the sake of simplicity, we present in Table 1 a generic formal model covering the underlying concepts of Dedal.

MACHINE $Arch_concepts$
INCLUDES $Basic_concepts$
SETS
$ARCHS; COMPS; COMP_NAMES$
VARIABLES
$architecture, arch_components, arch_connections, component,$
$comp_name, connection, comp_interfaces, client, server$
$arch_clients, arch_servers$
INVARIANT
/* A component has a name and a set of interfaces */
$component \subseteq COMPS \wedge$
$comp_name \in component \rightarrow COMP_NAMES \wedge$
$comp_interfaces \in component \rightarrowtail \mathcal{P}(interface) \wedge$
/* A client (resp. server)is a couple of a component and an interface */
$client \in component \leftrightarrow interface \wedge$
$server \in component \leftrightarrow interface \wedge$
/* A connection is a relation between a client and a server */
$connection \in client \leftrightarrow server \wedge$
/* An architecture has a set of components and connections */
$architecture \subseteq ARCHS \wedge$
$arch_components \in architecture \rightarrow \mathcal{P}(component) \wedge$
$arch_connections \in architecture \rightarrow \mathcal{P}(connection)$
/* Arch_clients (resp. arch_servers) lists the connected clients(reps. servers) within an architecture */
$arch_clients \in architecture \rightarrow \mathcal{P}\ (client) \wedge$
$arch_servers \in architecture \rightarrow \mathcal{P}\ (server)$

Specific B notations:
$\leftrightarrow$: relation $\rightarrowtail$: injection $\mathcal{P}$(`<set>`): powerset of `<set>`

Table 1. Formal specification of underlying concepts

For instance, the concept of *component* is specialized into *compRole* at the specification level and the concept of *architecture* is specialized into *configuration* at the configuration level.
This model is used to set generic evolution rules which are specialized for each of the three abstraction levels of Dedal. An evolution scenario is presented in Section 5 as an illustration.

4 The formal evolution approach

In this section, we present our approach to handle multi-level software evolution as a reuse-based process. The objective of this approach is twofold: (1) capture software change and control its impact on architecture consistency and, (2) propagate change between multiple architecture levels to preserve global coherence. The approach is formal model-based since it relies on the formal models of our three-level ADL and uses consistency and coherence properties and a set of evolution rules (*cf.* Figure 4). The approach is also dynamic in the sense that it performs analysis and simulates change on executable models. The formal models may be generated through a MDE (Model Driven Engineering) process where the source models are textual or graphical (UML profile) descriptions of Dedal. Since the transformation is not tooled yet, this issue is out of the scope of the present paper.
The evolution management is composed of three main activities: consistency analysis, inter-level coherence analysis and evolution rules triggering. In the re-

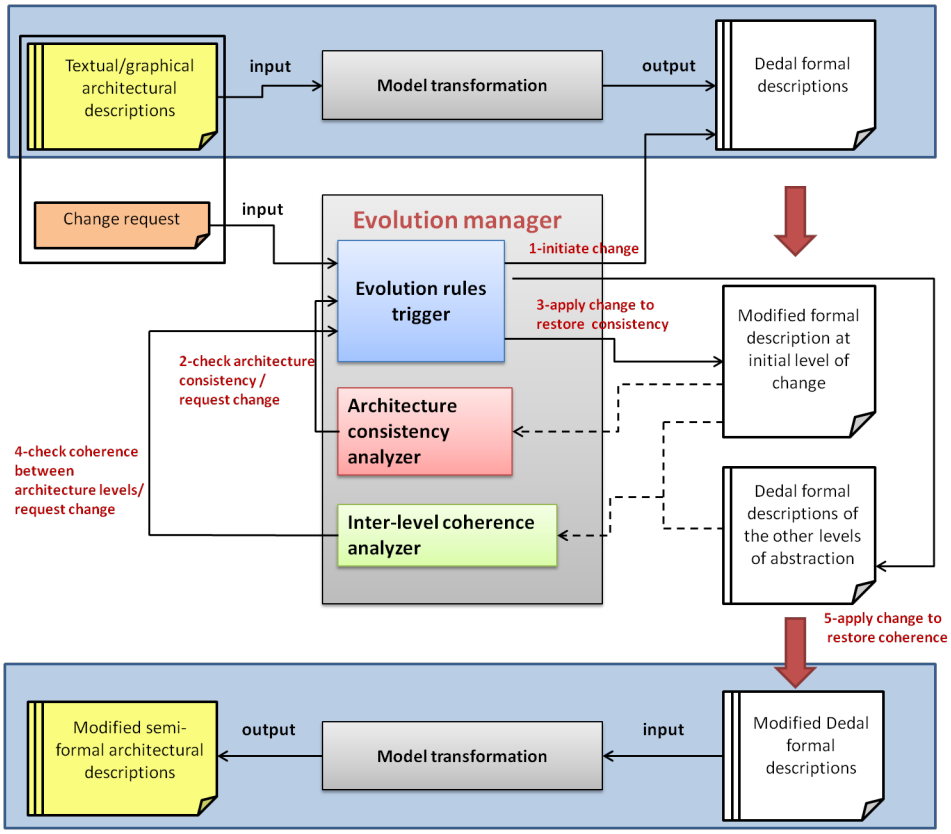

Fig. 4. The formal evolution process

mainder, we present the foundations and the mechanisms of each of these three activities.

4.1 Architecture consistency analysis

Taylor *et al.* [19] define consistency as an internal property intended to ensure that different elements of an architectural model do not contradict one another. Usually, this property includes five sorts of consistency: name, interface, behavior, interaction and refinement consistency. Some properties such as parameters, names and interfaces are taken into account by adding constraints in the definition of our architectural formal model [18]. In our approach, we focus on three main properties: name consistency, connection consistency, which includes interaction and compatibility between components, and architecture completeness.
Name consistency. This property ensures that each component belonging to the architecture holds a unique name and hence avoids conflicts when selecting components.

Connection consistency. This property ensures that all architecture connections are correct and satisfy compatibility between both sides (*i.e.* a required interface is always connected to a compatible provided one). In addition, connection consistency stipulates that the architecture graph is consistent (*i.e.* each component is connected to at least another one).
Architecture completeness. This property ensures that the architecture realizes all its functional objectives. From an internal point of view, completeness is satisfied when all the required services in the architecture are met. Structurally, it means that all the required interfaces are connected to a compatible provided one.
When a change occurs, the analyzer checks all the aforementioned properties and notify the evolution manager in case a violation is detected. Then, the adequate evolution rules are triggered to reestablish architecture consistency. The properties are defined using B, a first order set-theoretic formalism and hence analysis is performed using a B model checker.

4.2 Inter-level coherence properties

Coherence analysis is managed using inter-level rules (*cf.* Figure 5). These rules are defined to check whether a configuration conforms to its specification or a software instantiation is coherent with its configuration.

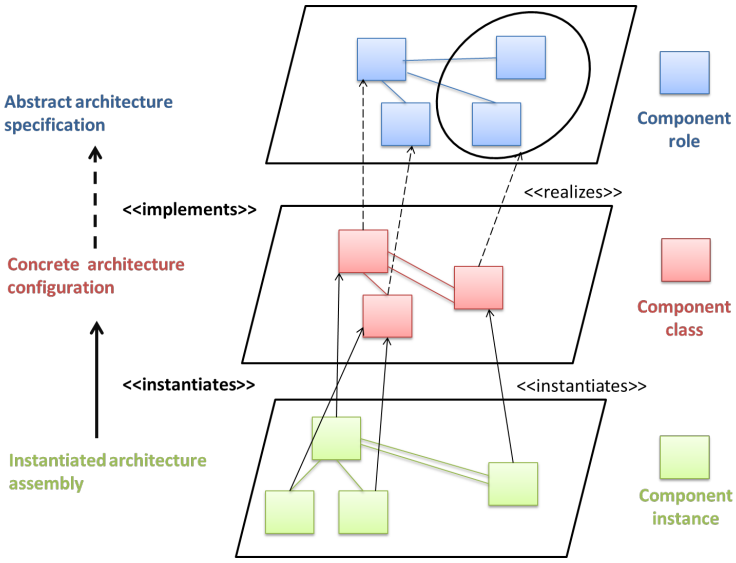

Fig. 5. Coherence between architecture levels

Coherence between specification and configuration. A specification is a formal description of software requirements that is used to guide the search for

suitable concrete component classes to implement the software. An architecture definition is coherent when all component roles are realized by component classes in the configuration. This results in a many-to many relation. Indeed, several component roles may be realized by a single component class while, conversely, a composition of component classes may be needed to realize a single role. Formally:

$$
\begin{array}{l}
implements \in configuration \leftrightarrow specification \wedge \\
\forall\ (Conf,\ Spec).(Conf \in configuration \wedge Spec \in specification \\
\quad \Rightarrow \\
\qquad (Conf,\ Spec) \in implements \\
\qquad\quad \Leftrightarrow \\
\quad \forall\ CR.(CR \in compRole \wedge CR \in spec_components(Spec) \Rightarrow \\
\qquad \exists\ CL.(CL \in compClass \wedge CL \in config_components(Conf) \wedge \\
\qquad\quad (CL,CR) \in realizes)))
\end{array}
$$

Coherence between configuration and assembly. Coherence between configuration and assembly levels is satisfied when all the classes of the configuration are instantiated at least once in the architecture assembly and, conversely, all instances of the assembly are instances of the component classes of the configuration. Formally:

$$
\begin{array}{l}
instantiates \in assembly \rightarrow configuration \wedge \\
\forall (Asm, Conf).(Asm \in assembly \wedge Conf \in configuration \\
\Rightarrow \\
((Asm, Conf) \in instantiates \\
\quad \Leftrightarrow \\
\forall CL.(CL \in compClass \wedge CL \in config_components(Conf) \\
\quad \Rightarrow \\
\quad \exists CI.(CI \in compInstance \wedge CI \in assm_components(Asm) \wedge \\
(CI, CL) \in comp_instantiates)) \wedge \\
\forall CI.(CI \in compInstance \wedge CI \in assm_components(Asm) \\
\quad \Rightarrow \\
\quad \exists CL.(CL \in compClass \wedge CL \in config_components(Conf) \wedge \\
(CI, CL) \in comp_instantiates))))
\end{array}
$$

Coherence analysis comes after consistency checking returns a positive result. Indeed, it is necessary that software system descriptions are consistent at all abstraction levels before checking coherence between them. When a change occurs at any level, this may result in erosion or drift (for instance, some higher level decisions are violated or not taken into account by the lower level). The evolution manager is then notified about the detected incoherence and propagates the change to the incoherent levels using the adequate evolution rules.

4.3 Specifying evolution rules

An evolution rule is an operation that changes a target software architecture by the deleting, adding or substituting of one of its constituent elements (components and connections). These rules are specified using the B notation and each rule is composed of three parts: the operation signature, preconditions and actions.

Architecture specification evolution. Evolving an architecture specification is usually a response to a new software requirement. For instance, the architect may need to add new functionalities to the system and hence add some new

roles to the specification. Moreover, a specification may also be modified during the change propagation process to preserve coherence and keep an up-to-date specification description of the system that may be implemented in several ways. The proposed evolution rules related to the specification level are the addition, deletion and substitution of a component role and the addition and deletion of connections. The following role addition rule is an example of evolution rules at specification level:

addRole(*spec*, *newRole*) =
PRE
$spec \in arch_spec \wedge newRole \in compRole \wedge newRole \notin spec_components(spec) \wedge$
/* *spec* does not contain a role with the same name*/
$\forall\ cr.(cr \in compRole \wedge cr \in spec_components(spec)$
$\Rightarrow comp_name(cr) \neg comp_name(newRole))$
THEN
spec_servers(*spec*) := *spec_servers*(*spec*) ∪ servers(newRole) ||
spec_clients(*spec*) := *spec_clients*(*spec*) ∪ clients(newRole) ||
spec_components(*spec*) := *spec_components*(*spec*) ∪ {*newRole*}
END;

Architecture configuration evolution. Change can be initiated at configuration level whenever new versions of software component classes are released. Otherwise, an implementation may also be impacted by change propagation either from the specification level, in response to new requirements, or from the assembly level, in response to a dynamic change of the system. Indeed, a configuration may be instantiated several times and deployed in multiple contexts. At configuration level, there is a need for two more evolution rules: the connection and the disconnection of the exposed services. Indeed, a component class used in a configuration may hold more provided interfaces than the component role that it implements. These extra interfaces may be left unconnected. On the contrary, a specification sets by definition the requirements, and hence the provided interfaces of all roles must be connected to keep the architecture consistent. As an example of evolution rule at configuration level, we list the following component class substitution rule:

replaceClass(*config*, *oldClass*, *newClass*) =
PRE
$oldClass \in compClass \wedge newClass \in compClass \wedge config \in configuration \wedge$
$oldClass \in config_components(config) \wedge$
/* The old component class can be substituted for the new one
(verified by the component substitution rule)*/
$newClass \notin config_components(config) \wedge (oldClass, newClass) \in class_substitution$
THEN
config_components(*config*) := (*config_components*(*config*) - {*oldClass*}) ∪ {*newClass*} ||
config_clients(*config*) := (*config_clients*(*config*) - *clients*(*oldClass*)) ∪ *clients*(*newClass*)
END

Architecture assembly evolution. Since the architecture assembly represents the software at runtime, evolving software at assembly level is a dynamic evolution issue. Several kinds of change may occur at runtime. For instance, dynamic software change may be needed due to a change in the execution context (*e.g.* lack of memory, CPU). Unanticipated changes are one of the most important issues in software evolution. Indeed, some software systems have to be self-adaptive to keep providing their functions despite environmental changes. This issues are

handled by the evolution manager which monitors the execution state of the software through its corresponding formal model. It then triggers the assembly evolution rules to restore consistency when it is violated. These rules include component instance deployment, component instance removal, component instance substitution, component instance connection / disconnection and service connection / disconnection. As an example of dynamic evolution rule, we state the following component instance addition rule:

deployInstance(asm, $inst$, $class$, $state$) =
  PRE
    $asm \in assembly \wedge class \in compClass \wedge$
/* The instance is a valid instantiation of an existing component class*/
    $inst \in compInstance \wedge class = comp_instantiates(inst) \wedge inst \notin assm_components(asm) \wedge$
/* The state given to the instance is a valid value assignment to the attributes of the instantiated component class*/
    $state \in \mathcal{P}\ (attribute_value) \wedge$ **card**$(state) =$ **card**$(class_attributes(class)) \wedge$
/* The maximum number of allowed instances of the given component class is not already reached*/
    $nb_instances(class) < max_instances(class)$
  THEN
/*initial and current state initialization*/
    initiation_state$(inst) := state$ ||
    current_state$(inst) := state$ ||
/*updating the number of instances and the assembly architecture*/
    nb_instances$(class) := nb_instances(class) + 1$ ||
    assm_components$(asm) := assm_components(asm) \cup \{inst\}$ ||
    assm_servers$(asm) := assm_servers(asm) \cup servers(inst)$ ||
    assm_clients$(asm) := assm_clients(asm) \cup clients(inst)$
  END;

5 Implementing an evolution scenario

To illustrate the use of evolution rules, we propose to evolve the Has architecture by adding of a new device that manages the building's shutters. The evolution simulation is performed using ProB [20], an animator and model checker of B models. Once the formal models corresponding to the three architecture descriptions are successfully checked, we use the ProB solver to trigger change as a goal to reach. In the remainder, we give some details about the example instances and the different steps of the evolution process.

5.1 Intra-level change

Figure 6 illustrates the old architecture specification and the evolved one.

Initially, the instantiation of the formal Has specification is as follows:

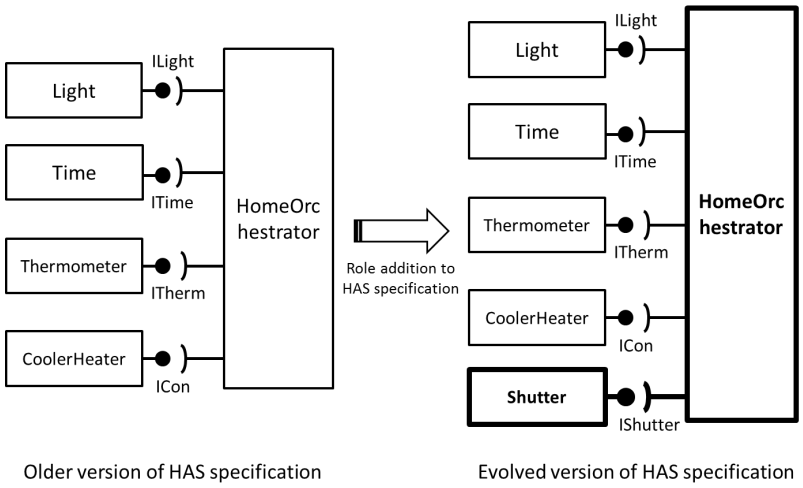

Fig. 6. Evolving the HAS specification by role addition

$$
\begin{array}{l}
\textbf{compRole} := \{cr1, cr1a, cr2, cr3, cr4, cr4a, cr5, cr6\}|| \\
\textbf{comp_name} := \{cr1 \mapsto Light, cr1a \mapsto ELight, cr2 \mapsto Time, \\
\quad cr3 \mapsto Thermometer, cr4 \mapsto HomeOrchestrator, \\
\quad cr4a \mapsto HomeOrchestrator, cr5 \mapsto CoolerHeater, \\
\quad cr6 \mapsto Shutter\}|| \\
\textbf{arch_spec} := \{HAS_spec\}|| \\
\textbf{spec_components} := \{HAS_spec \mapsto \{cr1, cr2, cr3, cr4, cr5\}\}|| \\
\textbf{spec_connections} := \{HAS_spec \mapsto \{ \\
\quad ((cr4, rintILight) \mapsto (cr1, pintILight)), \\
\quad ((cr4, rintITime) \mapsto (cr2, pintITime)), \\
\quad ((cr4, rintITherm2) \mapsto (cr3, pintITherm1)), \\
\quad ((cr4, rintICon) \mapsto (cr5, pintICon))\}\}|| \\
\textbf{spec_clients} := \{(HAS_spec \mapsto \{(cr4, rintILight), (cr4, rintITime), \\
\quad (cr4, rintITherm2), (cr4, rintICon))\}\}|| \\
\textbf{spec_servers} := \{(HAS_spec \mapsto \{(cr1, pintILight), (cr2, pintITime), \\
\quad (cr3, pintITherm1), (cr5, pintICon)\})\}
\end{array}
$$

The change is requested by the execution of the the role addition operation that takes as arguments the *HAS_spec* HAS architecture specification and the *Shutter* ($cr6$) component role.

$$\textbf{addRole}(HAS_spec, cr6)$$

The change process is initiated by setting a goal. When the goal cannot be reached, the change process rolls back to the initial state of the architecture. In this case, the goal is to add a *Shutter* to the HAS specification while maintaining architecture consistency (as defined in Section 4):

$$
\begin{array}{l}
GOAL == changeOperation = ADDITION \wedge selectedRole = cr6 \wedge \\
\quad selectedSpec = HAS_spec \wedge specification_consistency
\end{array}
$$

The change entails the disconnection of all servers, the deletion of the old orchestrator (cr4), the addition of the new orchestrator (cr4a) and finally the

connection of all servers. These operations are automatically generated by the ProB solver:

$$
\begin{array}{l}
disconnect(HAS_spec, (cr4, rintILight), (cr1, pintILight)) \\
disconnect(HAS_spec, (cr4, rintITime), (cr2, pintITime)) \\
disconnect(HAS_spec, (cr4, rintITherm1), (cr3, pintITherm)) \\
disconnect(HAS_spec, (cr4, rintICon), (cr5, pintICon)) \\
deleteRole(HAS_spec, cr4) \\
generateAddRole(HAS_spec, cr4a) \\
connect(HAS_spec, (cr4, rintILight), (cr1, pintILight)) \\
connect(HAS_spec, (cr4, rintITime), (cr2, pintITime)) \\
connect(HAS_spec, (cr4, rintITherm1), (cr3, pintITherm)) \\
connect(HAS_spec, (cr4, rintICon), (cr5, pintICon)) \\
connect(HAS_spec, (cr4a, rintIShutter), (cr6, pintIShutter))
\end{array}
$$

5.2 Propagating change to other levels

Once the change is successfully achieved at the specification level, the propagation rules are triggered in the other levels to attempt to restore coherence with the new specification architecture.

Propagating change to the HAS configuration. To restore conformity with the new HAS specification, the new configuration must realize the added *Shutter* role and its connection to the orchestrator device to perform the new required behavior. In the given example, the solution is to search for a concrete component class that realizes the *Shutter* role and can be connected to a compatible *orchestrator* class. Initially, the HAS configuration (illustrated by Figure 7) is formally instantiated as follows:

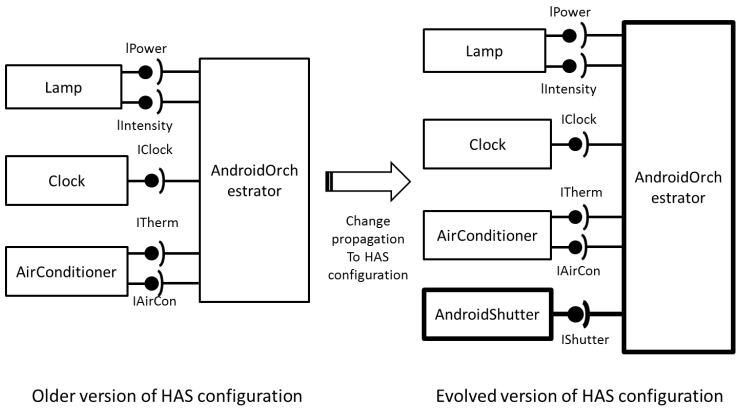

Fig. 7. Change propagation to HAS configuration

$$
\begin{array}{l}
\textbf{compClass} := \{cl1, cl2, cl3, cl4, cl3a, cl3b, cl4a, cl6\}|| \\
\textbf{comp_name} := \{cl1 \mapsto Lamp, cl2 \mapsto Clock, cl3 \mapsto AirConditioner, \\
\quad cl3a \mapsto CHEngine, cl3b \mapsto Thermostat, \\
\quad cl4 \mapsto AndroidOrchestrator, cl4a \mapsto AndroidOrchestrator, \\
\quad cl6 \mapsto AndroidShutter\}|| \\
\textbf{configuration} := \{HAS_config, AirConConfig\}|| \\
\textbf{compositeComp} := \{cl3\} \\
\textbf{composite_uses} := \{cl3 \mapsto AirConConfig\} \\
\textbf{config_components} := \{HAS_config \mapsto \{cl1, cl2, cl3, cl4\}, \\
\quad AirConConfig \mapsto \{cl3a, cl3b\}|| \\
\textbf{spec_connections} := \{HAS_config \mapsto \{ \\
\quad ((cl4, rintIPower) \mapsto (cl1, pintIPower)), \\
\quad ((cl4, rintIIntensity) \mapsto (cl1, pintIIntensity)), \\
\quad ((cl4, rintIClock) \mapsto (cl2, pintIClock)), \\
\quad ((cl4, rintITherm2) \mapsto (cl3, pintITherm2)), \\
\quad ((cl4, rintICon) \mapsto (cl3, pintICon))\}, \\
\quad AirConConfig \mapsto \{((cl3a, rintITherm1) \mapsto (cl3b, pintITherm1)), \\
\quad ((cl4, rintITime) \mapsto (cl2, pintILamp))\}|| \\
\textbf{config_clients} := \{(HAS_config \mapsto \{(cl4, rintILamp), (cl4, rintIIntensity), \\
\quad (cl4, rintIClock), (cl4, rintITherm2), (cl4, rintICon))\}\}|| \\
\textbf{config_servers} := \{(HAS_config \mapsto \{(cl1, pintILamp), (cl1, pintIIntensity), \\
\quad (cl2, pintITime), (cl3, pintITherm2), (cl3, pintICon)\})\}
\end{array}
$$

Again, we use the ProB solver giving it the following goal to restore coherence property with the new HAS specification:

$$
GOAL == selectedConfig = HAS_config \wedge configuration_consistency \wedge specConfigCoherence
$$

We note that *specConfigCoherence* is the conformity rule defined in Section 4 to check conformity between a specification and a configuration.

A potential solution generated by the solver is:

$$
\begin{array}{l}
disconnect(HAS_config, (cl4, rintILamp), (cl1, pintILamp)) \\
disconnect(HAS_config, (cl4, rintIIntensity), (cl1, pintIItensity)) \\
disconnect(HAS_config, (cl4, rintIClock), (cl1, pintIClock)) \\
disconnect(HAS_config, (cl4, rintITherm2), (cl3, pintITherm2)) \\
disconnect(HAS_config, (cl4, rintICon), (cl3, pintICon2)) \\
deleteClass(HAS_config, cl4) \\
addClass(HAS_config, cl4a) \\
connect(HAS_config, (cl4a, rintILamp2), (cl1, pintILamp)) \\
connect(HAS_config, (cl4a, rintIIntensity2), (cl1, pintIItensity)) \\
connect(HAS_config, (cl4a, rintIClock), (cl1, pintIClock)) \\
connect(HAS_config, (cl4a, rintITherm3), (cl3, pintITherm2)) \\
connect(HAS_config, (cl4a, rintICon2), (cl3, pintICon2)) \\
connect(HAS_config, (cl4a, rintIShutter), (cl6, pintIShutter))
\end{array}
$$

Propagating change to the HAS assembly. In the same way, change is propagated to assembly level by disconnecting and deleting the instance of the old *AndroidOrchestrator* and by creating, deploying and connecting an instance of the new added *Shutter* device.
The solver is given the following goal:

$$
\begin{array}{l}
GOAL == selectedAssembly = HAS_assembly \wedge assembly_consistency \wedge \\
\quad assemblyConfigCoherence
\end{array}
$$

The *assemblyConfigCoherence* is the defined property to check coherence between an assembly and a configuration (*cf.* Section 4).
The solution generated by the solver is as follows:

$unbind(HAS_assembly, (ci4, rintILampInst), (ci1, pintILampInst1))$
$unbind(HAS_assembly, (ci4, rintIIntensityInst), (cl1, pintIItensity1Inst))$
$unbind(HAS_assembly, (ci4, rintILampInst), (ci2, pintILampInst2))$
$unbind(HAS_assembly, (ci4, rintIIntensityInst), (ci2, pintIItensityInst2))$
$unbind(HAS_assembly, (ci4, rintIClockInst), (ci1, pintIClockInst))$
$unbind(HAS_assembly, (ci4, rintITherm2Inst), (ci3, pintITherm2Inst))$
$unbind(HAS_assembly, (ci4, rintIConInst), (ci3, pintICon2Inst))$
$removeInstance(HAS_assembly, ci4)$
$deployInstance(HAS_assembly, ci4a, cl4a, \{\})$
$bind(HAS_assembly, (ci4a, rintILamp2Inst), (ci1, pintILampInst1))$
$bind(HAS_assembly, (ci4a, rintIIntensity2Inst), (ci1, pintIItensityInst1))$
$bind(HAS_assembly, (ci4a, rintIClockInst), (ci1, pintIClockInst))$
$bind(HAS_assembly, (ci4a, rintILamp2Inst), (ci2, pintILampInst2))$
$bind(HAS_assembly, (ci4a, rintITherm3Inst), (ci3, pintITherm2Inst))$
$bind(HAS_assembly, (ci4a, rintICon2Inst), (ci3, pintICon2Inst))$
$bind(HAS_assembly, (ci4a, rintIIntensity2Inst), (ci1, pintIItensityInst2))$
$bind(HAS_assembly, (ci4a, rintIShutterInst), (ci6, pintIShutterInst))$

At this stage, change is simulated and verified semi-automatically since the models are instantiated manually. Moreover, a manual checking is needed to validate the proposed evolution rules. A perspective is to fully automate the evolution management process and to study the scalability of the solver to timely handle complex goals.

6 Conclusion and future work

In this paper, we proposed a set of rules to evolve software architectures. These rules defined as a B formal model of our three-level Dedal ADL that covers the whole lifecycle of software systems. Our approach enables simulation and early validation of software evolution at design time (specification and implementation) as well as runtime (deployment). At this stage, the proposed consistency properties and evolution rules are checked and validated using a B animator and model checker. As a future work, we aim to extend the use of the proposed evolution rules in order to consider the semantics of changes. Another perspective is to generate multiple candidate evolution paths that can be evaluated using some criteria (*e.g.* quality of service, cost, change priority) as proposed by Barnes *et al.* [21].
We are also considering several MDE techniques to develop an eclipse-based environment for Dedal that automatically manages software architecture evolution.

References

1. Lientz, B.P., Swanson, E.B., Tompkins, G.E.: Characteristics of application software maintenance. Communication of the ACM **21**(6) (June 1978) 466–471
2. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Software Engineering Notes **17**(4) (October 1992) 40–52
3. de Silva, L., Balasubramaniam, D.: Controlling software architecture erosion: A survey. Journal of Systems and Software **85**(1) (January 2012) 132–151
4. Zhang, H.Y., Urtado, C., Vauttier, S.: Architecture-centric component-based development needs a three-level ADL. In: Proceedings of the 4th ECSA. Volume 6285 of LNCS., Copenhagen, Denmark, Springer (August 2010) 295–310

5. Zhang, H.Y., Zhang, L., Urtado, C., Vauttier, S., Huchard, M.: A three-level component model in component-based software development. In: Proceedings of the 11th GPCE, Dresden, Germany, ACM (September 2012) 70–79
6. Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York, USA (1996)
7. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE TSE **26**(1) (January 2000) 70–93
8. Medvidovic, N.: ADLs and dynamic architecture changes. In: Joint Proceedings of the Second International Software Architecture Workshop and International Workshop on Multiple Perspectives in Software Development on SIGSOFT '96 Workshops, New York, USA, ACM (1996) 24–27
9. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM TOSEM **6**(3) (July 1997) 213–249
10. Luckham, D.C., Kenney, J.J., Augustin, L.M., Vera, J., Bryan, D., Mann, W.: Specification and analysis of system architecture using rapide. IEEE TSE **21** (1995) 336–355
11. Garlan, D., Monroe, R., Wile, D.: ACME: An architecture description interchange language. In: Proceedings of CASCON, IBM Press (1997)
12. Magee, J., Kramer, J.: Dynamic structure in software architectures. ACM SIGSOFT Software Engineering Notes **21**(6) (1996) 3–14
13. Oquendo, F.: {Pi-ADL}: An architecture description language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures. SIGSOFT Software Engineering Notes **29**(3) (May 2004) 1–14
14. Shaw, M., Garlan, D.: Formulations and formalisms in software architecture. In Leeuwen, J., ed.: Computer Science Today. Volume 1000 of LNCS. Springer (1995) 307–323
15. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM **21**(8) (August 1978) 666–677
16. Joolia, A., Batista, T., Coulson, G., Gomes, A.T.A.: Mapping ADL specifications to an efficient and reconfigurable runtime component platform. In: Proceedings of the 5th WICSA, Washington, USA, IEEE (2005) 131–140
17. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I. Information and Computation **100**(1) (September 1992) 1–40
18. Mokni, A., Huchard, M., Urtado, C., Vauttier, S., Zhang, H.Y.: Towards automating the coherence verification of multi-level architecture descriptions. In: Proceedings of the 9th ICSEA, Nice, France (October 2014)
19. Taylor, R., Medvidovic, N., Dashofy, E.: Software architecture: Foundations, Theory, and Practice. Wiley (2009)
20. Leuschel, M., Butler, M.: ProB: An automated analysis toolset for the b method. International Journal on Software Tools for Technology Transfer **10**(2) (February 2008) 185–203
21. Barnes, J., Garlan, D., Schmerl, B.: Evolution styles: foundations and models for software architecture evolution. Software and Systems Modeling **13**(2) (2014) 649–678

View publication stats