A three-level formal model for software architecture evolution

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

+LGI2P, Ecole Nationale Supérieure des Mînes 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. This papers gives an overview of our formal approach to address the architecture-centric evolution at the three main steps of component-based software development: specification, implementation and deployment. We illustrate our proposal with an example of software evolution that leads to erosion and we demonstrate how our evolution process can resolve this problem.

Keywords: Software architecture, architecture levels, reuse, software evolution, B formal models

1 Introduction

Software evolution has gained a lot of interest during the last years [1]. Indeed, as software ages, it needs to evolve and be maintained to fit new user requirements. This avoids to build a new software from scratch and hence save time and money. Handling evolution in component-based software systems is non trivial since an ill-mastered change may lead to architecture inconsistencies and incoherence between design and implementation. Many ADLs (Architecture Description Languages) were proposed to support architecture modeling and analysis. Examples include C2SADL [2], Wright [3] and Darwin [4]. Although, some ADLs integrate architecture modification languages, handling and controlling architecture evolution in the overall software lifecycle is still an important issue. In this paper, we attempt to provide a solution to the architecture-centric evolution that preserves consistency and coherence between architecture levels. We propose an architecture evolution process based on the formal foundations of our three-level ADL Dedal [5]. The process is then illustrated by an evolution scenario on a simplified Home Automation Software architecture. The remainder of this paper is organized as follows: Section 2 briefly discusses examples of existing ADLs. Section 3 gives an overview of Dedal and its formal foundations. Section 4 describes the evolution process based on Dedal. Section 5 presents an evolution scenario that illustrates the proposed evolution process before Section 6 concludes and discusses future work.

2 Existing ADLs

Over the two last decades, a number of ADLs were proposed [6]. Most of them provide textual notations to describe architectural entities (*i.e.* components, interfaces and connections). Initially, ADLs were domain-specific. Examples include C2SADL [2] for the design of concurrent systems and Wright [3] and Darwin [4] for the design and analysis of distributed architectures. Later on, attempts to unify ADLs and make them general-purpose were made. For example, ACME [7] was designed for such purpose. It consists in a common interchange description language that offers annotation facilities to support architecture descriptions in other languages. Another relevant example is xADL 2.0 [8]. It was designed to support various types of systems. The strength of xADL 2.0 resides in its extensibility since it is XML-based. It offers then an easy way for architects to adapt its use to any kind of architectures.

Although a lot of effort was dedicated to improve the expressiveness of ADLs and promote their use to model software architectures, several important issues are not taken into account. First, existing ADLs hardly support all the development steps of component-based software architectures (*i.e.* specification, implementation and deployment). Most of them cover only one or at most two levels which harden their integration in development processes. Second, they hardly support architecture evolution and handle architecture inconsistencies such as drift or erosion [9]. C2SADL and Darwin are exceptions. They include language to describe changes. However, they do not support reverse evolution and they do not cover all steps of component-based development either.

In this work, we address architecture evolution in the whole component-based development process. We show how architectural erosion could be avoided thanks to reverse evolution.

3 Overview of Dedal

3.1 The three architecture 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 in a reuse-based process [10].

To illustrate the three architecture levels of Dedal, we propose an example of a Home Automaton Software (HAS). Figure 1 presents the HAS architecture at three abstraction levels:
The abstract architecture specification (*cf.* Figure 1-a) is the first level of architecture software descriptions. It represents the architecture as designed by the architect and after analyzing the requirements of the future software. In Dedal, the architecture specification is composed of component roles and their connections. 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.

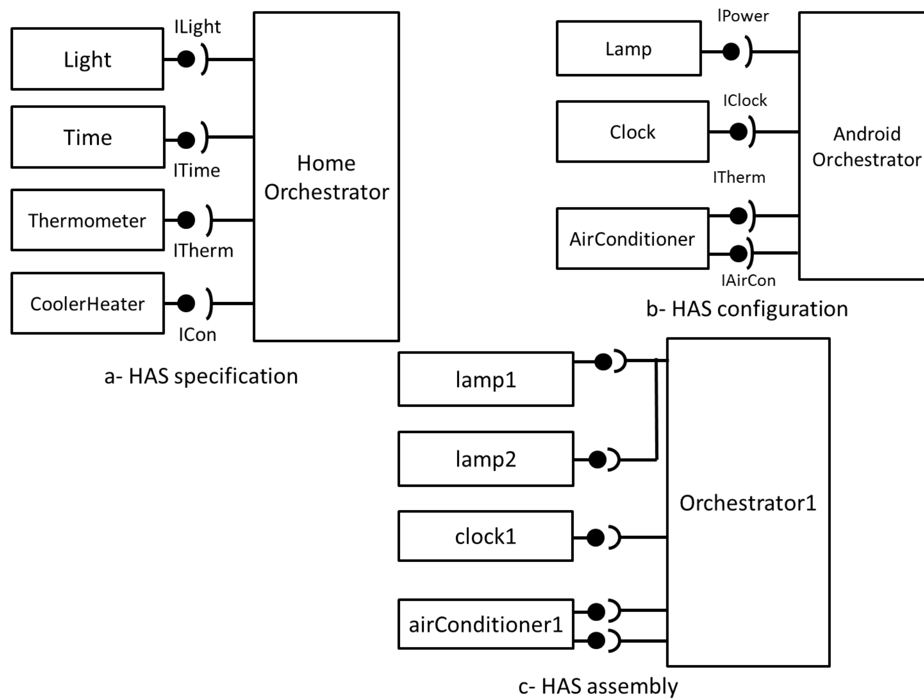

Fig. 1. Illustrative Example

The concrete architecture configuration (*cf.* Figure 1-b) is an implementation view of the software architecture. 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. *Primitive component classes* encapsulate executable code. *Composite component classes* encapsulate 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.

The instantiated architecture assembly (*cf.* Figure 1-c) 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).

3.2 The formal foundations of Dedal

Dedal is formalized using B [11], a set-theoretic and first order logic formalism. The formalization [12] covers all the concepts of Dedal and includes a set of rules that defines the relations between the different artifacts into and over each architecture level of Dedal (*cf.* Figure 2). These rules are classified into two categories: the intra-level rules and the inter-level rules.

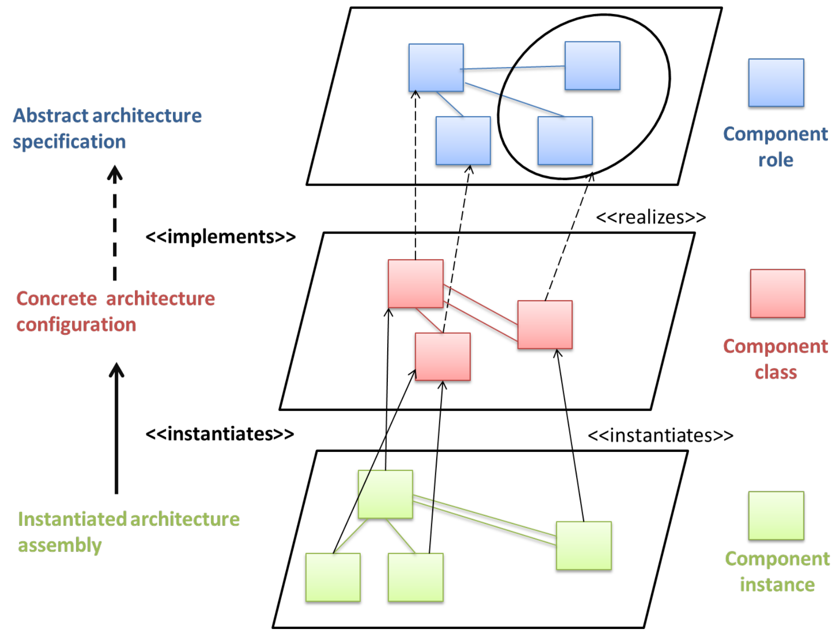

Fig. 2. Inter-level relations in Dedal

Intra-level rules in Dedal consist in substitutability and compatibility between components of the same abstraction level (component roles, concrete component types, instances). Defining intra-level relations is necessary to check the architecture consistency. For instance, the components must be correctly connected to each other (*i.e.* each required interface is connected to a compatible provided one).

Inter-level rules are specific to Dedal and consist in relations between components at different abstraction levels as shown in Figure 2. Defining inter-level rules is mandatory to decide about coherence between two architecture descriptions at different abstraction levels. For instance, the realization rule is used to check that a given configuration is a valid implementation of a given specification and the instantiation rule is used to check if an assembly correctly instantiates a given configuration.

4 Software architecture evolution in Dedal

Handling software evolution in Dedal is quite advantageous. Indeed, Dedal covers the whole life-cycle of software systems and hence all descriptions can be kept up-to-date for further reuse, reimplementation and deployment in different contexts. To keep architecture descriptions coherent, change must be propagated from where it is initiated to the other abstraction levels descriptions. As a solution, we propose an evolution process based on Dedal to enable change in software systems in a manner that preserves architecture consistency and coherence.

The evolution process in Dedal lies on two kinds of rules: (1) static rules to check architecture consistency and coherence between the different descriptions and (2) evolution rules to trigger the change at each abstraction level and propagate it to the other levels descriptions. Figure 3 presents the condition diagram of the evolution process.

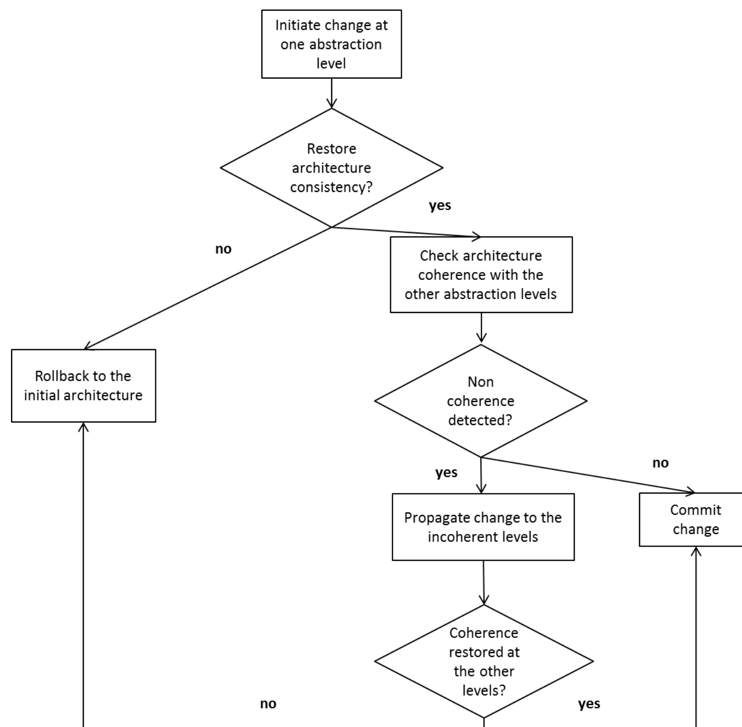

Fig. 3. Condition diagram of the evolution process

4.1 Static rules

Static rules in Dedal are classified into consistency rules and coherence rules. Consistency rules are name uniqueness, completeness, connection correctness and graph consistency. These properties are verified at the same abstraction level by the evolution manager to check weather the architecture is structurally consistent or a change must be triggered to restore consistency. Coherence rules are used to check if software descriptions at the three abstraction levels of Dedal are coherent. If an incoherence is detected, the evolution manager propagates change to the other levels to restore coherence. Coherence rules include the verification of the following properties:

- A configuration *Conf* is an implementation of a given specification *Spec*.
- A specification *Spec* is a documentation of a given configuration *Conf*.
- An assembly *Asm* is an instantiation of a given configuration *Conf*.
- A configuration *Conf* is instantiated by a given assembly *Asm*.

4.2 Evolution rules

An evolution rule is an operation that makes change in a target software architecture by the deletion, addition or substitution of one of its constituent elements (components and connections). Each rule is composed of three parts: the operation signature, preconditions and actions. Specific evolution rules are defined at each abstraction level to perform change at the corresponding formal description. These rules are triggered by the evolution manager when a change is requested. Firstly, a sequence of rule triggers is generated to reestablish consistency at the formal description of the initial level of change. Afterward, the evolution manager attempts to restore coherence between the other descriptions by executing the adequate evolution rules. The following role addition rule is an example of evolution rules at specification level:

```
/* Operation signature takes as arguments an instance of the architecture
   specification(spec) and the instance of the new role(newRole)*/
addRole(spec, newRole) =
/* preconditions */
PRE
spec ∈ arch_spec ∧ newRole ∈ compRole ∧ newRole ∉ spec_components(spec) ∧
/* spec does not contain a role with the same name*/
∀ cr.(cr ∈ compRole ∧ cr ∈ spec_components(spec)
   ⇒ comp_name(cr) ≠ comp_name(newRole))
THEN
/* actions */
/* update the set of clients (required interfaces), the set of
   servers (provided interfaces) and the set of component roles */
      spec_servers(spec) := spec_servers(spec) ∪ servers(newRole) ||
      spec_clients(spec) := spec_clients(spec) ∪ clients(newRole) ||
      spec_components(spec) := spec_components(spec) ∪ {newRole}
END;
```

5 Evolution scenario

5.1 Motivation

To illustrate the evolution approach, we propose an example of evolving the Has architecture. The objective is to enable the control of the house through a mobile

device running under Android OS. The change is initiated at the configuration level and attempts to adapt the current HAS implementation to an android device.

Figure 4-a shows the initial implementation of HAS while Figure 4-b shows the evolved one. Two main changes are noticed in the new configuration: the orchestrator is substituted for a new one compatible with android. Since a new service to control the intensity of lamp is required, the component *Lamp* is replaced by the component *AdjustableLamp* with additional provided interface (*IIntensity*) to adjust the luminosity.

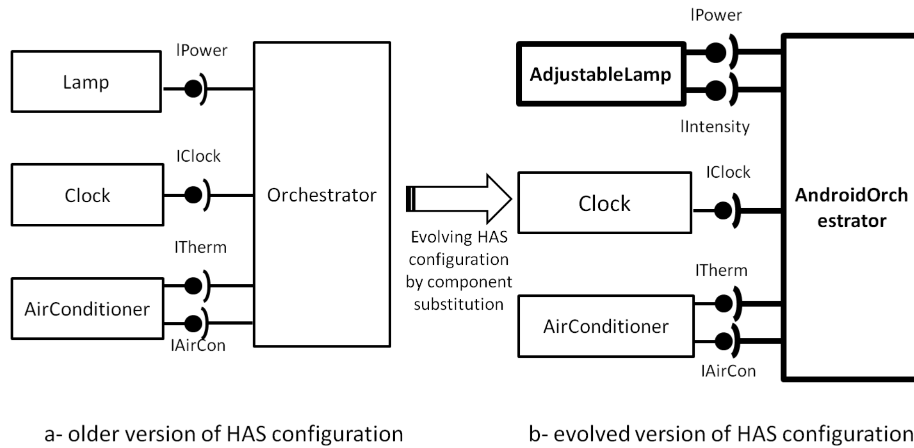

Fig. 4. Evolving the HAS configuration

5.2 Tool support overview

At this stage of work, the evolution process is assisted using ProB [13], an animation tool of B models. We manually instantiate the B formal models corresponding to the Has architecture and execute evolution rules at each abstraction level. We control the evolution process by checking consistency and coherence properties thanks to the evaluation console of ProB. This first step provides a proof feasability of our work. Ongoing work is to automate the generation of Dedal formal models and use the ProB solver to automate the evolution process.

5.3 Evolving the HAS configuration

The change is initiated by disconnecting and deleting the old orchestrator. Then, the one compatible with android is added and connected. The old Lamp is replaced by the adjustable one and connected to the android orchestrator. The evolution manager performs the following operations:

$disconnect(HAS_config, (cl4, rintIPower), (cl1, pintIPower))$
$disconnect(HAS_config, (cl4, rintIClock), (cl2, pintIClock))$
$disconnect(HAS_config, (cl4, rintITherm), (cl3, pintITherm))$
$disconnect(HAS_config, (cl4, rintIAirCon), (cl3, pintIAirCon))$
$deleteClass(HAS_config, cl4)$
$addClass(HAS_config, cl4a)$
$replaceClass(HAS_config, cl1, cl1a)$
$connect(HAS_config, (cl4a, rintIPower2), (cl1a, pintIPower2))$
$connect(HAS_config, (cl4a, rintIIntensity), (cl1a, pintIIntensity))$
$connect(HAS_config, (cl4a, rintIClock2), (cl2, pintIClock))$
$connect(HAS_config, (cl4a, rintITherm2), (cl3, pintITherm))$
$connect(HAS_config, (cl4a, rintIAirCon2), (cl3, pintIAirCon2))$

We note that *cl1*, *cl2*, *cl3* and *cl4* refer respectively to the component classes *Lamp*, *Clock*, *AirConditioner* and *Orchestrator*. *cl1a* and *cl4a* refer respectively to the new component classes *AdjustableLamp* and *AndroidOrchestrator*. Their Interface names are prefixed with *rint*(for a required interface) and *pint*(for provided interface) followed by the interface type name and eventually a number when there is several instances of the same interface

5.4 Propagating change to the HAS specification

The current HAS specification is no longer a good documentation of the new version of the HAS configuration and thus, we have a problem of erosion. Indeed, the control of the light intensity is not included in the current specification. Hence, a new documentation version is required to keep both descriptions coherent. Figure 5 shows the initial and evolved version of the HAS specification after the change propagation.

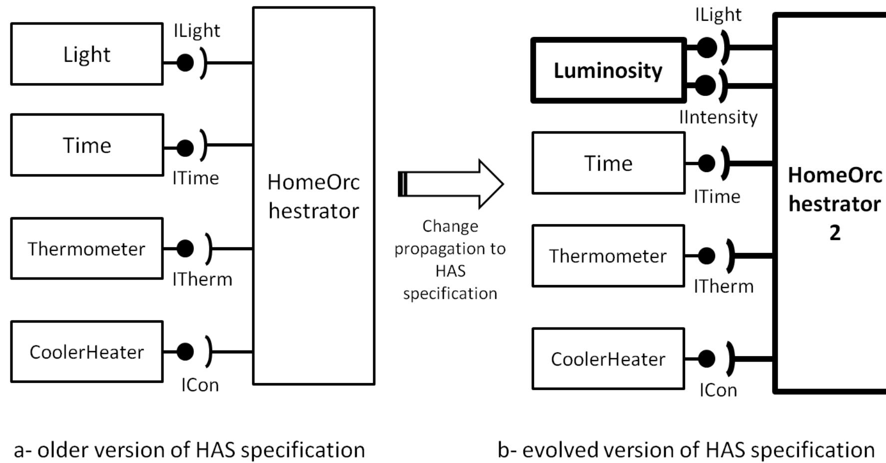

Fig. 5. Evolving the HAS specification by change propagation

The change is propagated to the HAS specification by replacing the role *HomeOrchestrator*(cr4) with *HomeOrchestrator2*(cr4a) and the role *Light*(cr1)

by *Lunminosity*(cr1a). The following operations are performed by the evolution manager:

$disconnect(HAS_spec, (cr4, rintILight), (cr1, pintILight))$
$disconnect(HAS_spec, (cr4, rintITime), (cr2, pintITime))$
$disconnect(HAS_spec, (cr4, rintITherm1), (cr3, pintITherm))$
$disconnect(HAS_spec, (cr4, rintICon), (cr5, pintICon))$
$deleteRole(HAS_spec, cr4)$
$addRole(HAS_spec, cr4a)$
$replaceRole(HAS_spec, cr1, cr1a)$
$connect(HAS_spec, (cr4a, rintILight2), (cr1a, pintILight2))$
$connect(HAS_spec, (cr4a, rintIIntensity), (cr1a, pintIIntensity))$
$connect(HAS_spec, (cr4a, rintITime), (cr2, pintITime))$
$connect(HAS_spec, (cr4a, rintITherm1), (cr3, pintITherm))$
$connect(HAS_spec, (cr4a, rintICon), (cr5, pintICon))$

5.5 Propagating change to the HAS assembly

The current HAS assembly violates the instantiation rule according to the new version of the HAS configuration. This violation is detected by the evolution manager and change is triggered at the assembly level to restore coherence. Figure 6 illustrates the changes applied on the assembly architecture.

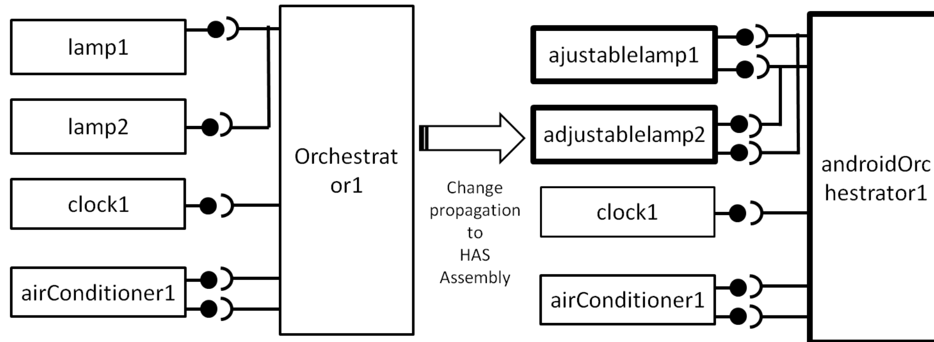

Fig. 6. Evolving the HAS assembly

6 Conclusion and future work

In this paper, we give an overview of our three-level ADL Dedal and its formal model. At this stage, a set of evolution rules is proposed to handle architecture change during the three steps of software lifecycle: specification, implementation and deployment. The rules were tested and validated on sample models using a B model checker. As future work, we aim to manage the history of architecture changes in Dedal descriptions as a way to manage software system versions. Furthermore we are considering to automate evolution by integrating Dedal and evolution rules into an eclipse-based platform.

References

1. Mens, T., Serebrenik, A., Cleve, A., eds.: Evolving Software Systems. Springer (2014)
2. 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 (Viewpoints '96) on SIGSOFT '96 Workshops, New York, USA, ACM (1996) 24–27
3. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM TOSEM **6**(3) (July 1997) 213–249
4. Magee, J., Kramer, J.: Dynamic structure in software architectures. In: Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering. SIGSOFT '96, New York, NY, USA, ACM (1996) 3–14
5. Zhang, H.Y., Urtado, C., Vauttier, S.: Architecture-centric component-based development needs a three-level ADL. In: Proceedings of the 4th European Conference on Software Architecture. Volume 6285 of LNCS., Copenhagen, Denmark, Springer (August 2010) 295–310
6. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering **26**(1) (January 2000) 70–93
7. Garlan, D., Monroe, R., Wile, D.: Acme: An architecture description interchange language. In: Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research. CASCON '97, IBM Press (1997) 7–
8. Dashofy, E., van der Hoek, A., Taylor, R.: A highly-extensible, xml-based architecture description language. In: Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on. (2001) 103–112
9. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Software Engineering Notes **17**(4) (October 1992) 40–52
10. 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
11. Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York, USA (1996)
12. 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 International Conference on Software Engineering Advances, Nice, France (October 2014)
13. 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