A formal approach for managing component-based architecture evolution

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

A formal approach for managing component-based architecture evolution

Abderrahman Mokni [a,*], Christelle Urtado [a,*], Sylvain Vauttier [a,*], Marianne Huchard [b], Huaxi Yulin Zhang [c]

[a] *LGI2P, Ecole Nationale Supérieure des Mines Alès, Nîmes, France*
[b] *LIRMM, CNRS and Université de Montpellier, Montpellier, France*
[c] *Laboratoire MIS, Université de Picardie Jules Verne, Amiens, France*

A B S T R A C T

Software architectures are subject to several types of change during the software lifecycle (*e.g.* adding requirements, correcting bugs, enhancing performance). The variety of these changes makes architecture evolution management complex because all architecture descriptions must remain consistent after change. To do so, whatever part of the architectural description they affect, the effects of change have to be propagated to the other parts. The goal of this paper is to provide support for evolving component-based architectures at multiple abstraction levels. Architecture descriptions follow an architectural model named Dedal, the three description levels of which correspond to the three main development steps – specification, implementation and deployment. This paper formalizes an evolution management model that generates evolution plans according to a given architecture change request, thus preserving consistency of architecture descriptions and coherence between them. The approach is implemented as an Eclipse-based tool and validated with three evolution scenarios of a Home Automation Software example.

Keywords:
Architecture evolution
Architecture analysis
Evolution rules
Formal models
MDE

1. Introduction

Component-based software development (CBSD) promotes a reuse-based approach to defining, implementing and composing loosely coupled independent software components into whole software systems [1]. While component reuse is crucial to shorten large-scale software systems development time, handling evolution in such processes is a significant issue [2]. Indeed, software systems have to evolve to extend their functionalities, correct bugs, improve performance and quality, or adapt to their environment. While unavoidable, software changes may engender several inconsistencies and system dysfunction if not analyzed and handled carefully. In turn, an ill-mastered evolution engenders software degradation, the loss of its evolvability and then its phase-out [3].

A famous problem of software evolution is software architecture erosion [4,5]. It arises when modifications of the software implementation violate the design principles captured by its architecture. To increase confidence in reuse-centered, component-based software systems, all architecture descriptions must remain consistent and coherent with each other after every change.

* Corresponding authors.
E-mail addresses: Abderrahman.Mokni@mines-ales.fr (A. Mokni), Christelle.Urtado@mines-ales.fr (C. Urtado), Sylvain.Vauttier@mines-ales.fr (S. Vauttier), Marianne.Huchard@lirmm.fr (M. Huchard), yulin.zhang@u-picardie.fr (H.Y. Zhang).

While a lot of work has been dedicated to architectural modeling and evolution, there still is a lack of means and techniques to tackle architectural inconsistencies, and erosion in particular. Indeed, most existing approaches to architecture evolution hardly support the whole life-cycle of component-based software and only enable evolution of early stage models by propagating change impact to runtime models while evolution of runtime models are not fully dealt with, thus increasing the risks of architecture erosion.

This paper proposes an approach and its implementation to automatically manage component-based architecture evolution at multiple abstraction levels in a manner that preserves architecture consistency and coherence all along the software lifecycle. The approach is based on the Dedal [6,7] architectural model that explicitly models architectures at three abstraction levels, each corresponding to one of the three major steps of CBSD – specification, implementation and deployment, thus granting a full evolution management process. Given a change request at any abstraction level, it transforms Dedal models into B formal models to analyze the requested change and generates an evolution plan that guarantees the consistency of architecture descriptions and the coherence between them. The proposed approach is centered on a formal evolution management model that includes the generated B models, the architecture properties to preserve and a set of evolution rules. It is implemented as an Eclipse-based tool that generates B models from diagrammatic Dedal models and uses our specific solver to resolve architecture evolution. The overall approach is illustrated with a Home Automation Software case-study.

The remainder of this paper outlines as follows: Section 2 presents the background of this work. Section 3 presents our proposal to tackle multi-level architecture evolution (*i.e.* the evolution of architecture definitions composed of multiple description levels) while Section 4 presents the implemented tool and experiments on three evolution scenarios. Section 5 discusses related work and finally, Section 6 concludes the paper and discusses future work.

2. Background

Our approach combines the use of Dedal to model software architectures and B to support automated analysis and verification. This section briefly introduces these languages.

2.1. The Dedal architecture model

2.1.1. Component-based software development by reuse

CBSD follows the *reuse-in-the-large* principle. Reusing existing (off-the-shelf) software components [8] therefore becomes the central concern during development. Traditional software development processes cannot be used as is and must be adapted to component reuse [1]. Fig. 1 illustrates our vision of such a development process which is classically divided in two:

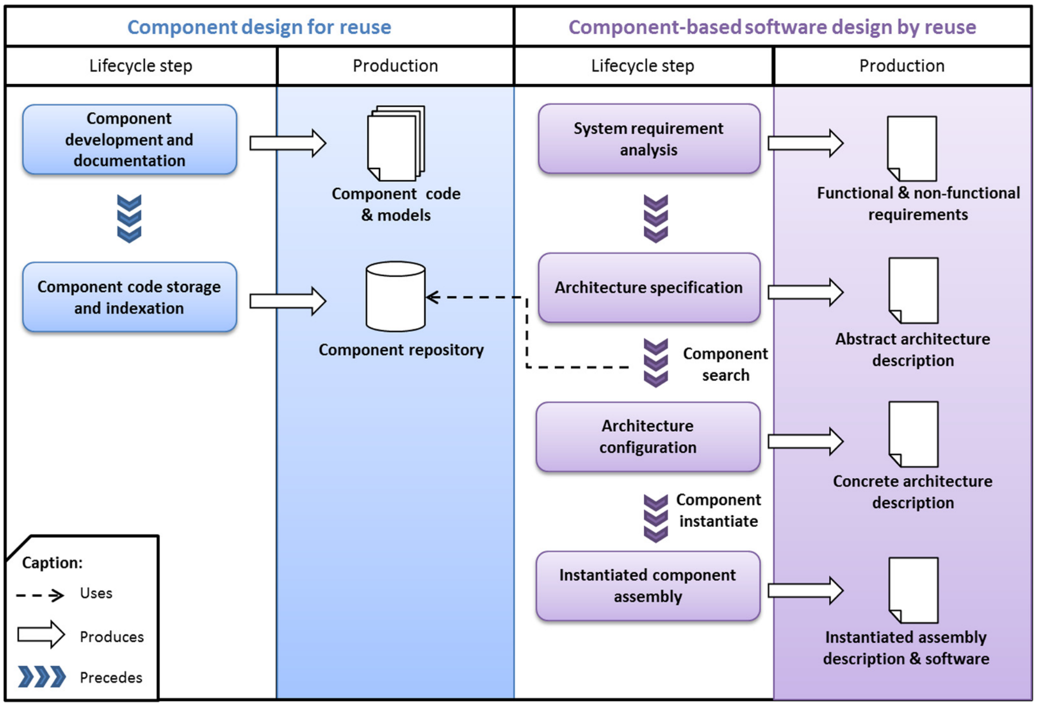
- the component development process (referred to as software component development for reuse), which will not be detailed in the sequel. This development process produces components that are stored in repositories for later use by the software development process.
- the software development process (referred to as software development by component reuse) that describes how previously developed software components can be used for software development (and how this reuse impacts the way software is built).

Dedal is a novel architectural model and ADL [6,7] that targets reuse-centered development. It covers the whole software development by component reuse life-cycle. The main idea of Dedal is to build a concrete architecture composed of stored and indexed components that are found in a component repository as candidates to satisfy the design decisions specified in an intended architecture specification. The resulting concrete architecture can then be instantiated and deployed in multiple contexts. Therefore, Dedal proposes a three-step approach for specifying, implementing and deploying software architectures.

2.1.2. Dedal abstraction levels

To illustrate the concepts of Dedal, we propose to model a home automation software (HAS) that manages comfort scenarios, which automatically controls buildings' lighting and heating depending on 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 is abstract: 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. Consequently, the provided interfaces of each role are to be connected to compatible required interfaces. Component roles are identified by the architect in order to search for and select corresponding concrete components in the next step. Fig. 2-a shows a possible HAS architecture specification. In this specification, five component roles are identified. A component playing the *HomeOrchestrator* role controls four components playing the *Light*, *Time*, *Thermometer* and *CoolerHeater* roles.



Component design for reuse
Component-based software design by reuse
Lifecycle step
Production
Lifecycle step
Production
Component development and documentation
Component code & models
Component code storage and indexation
Component repository
System requirement analysis
Functional & non-functional requirements
Architecture specification
Abstract architecture description
Component search
Architecture configuration
Concrete architecture description
Component instantiate
Instantiated component assembly
Instantiated assembly description & software
Caption:
Uses
Produces
Precedes

Fig. 1. Dedal reuse-centered development process [7].

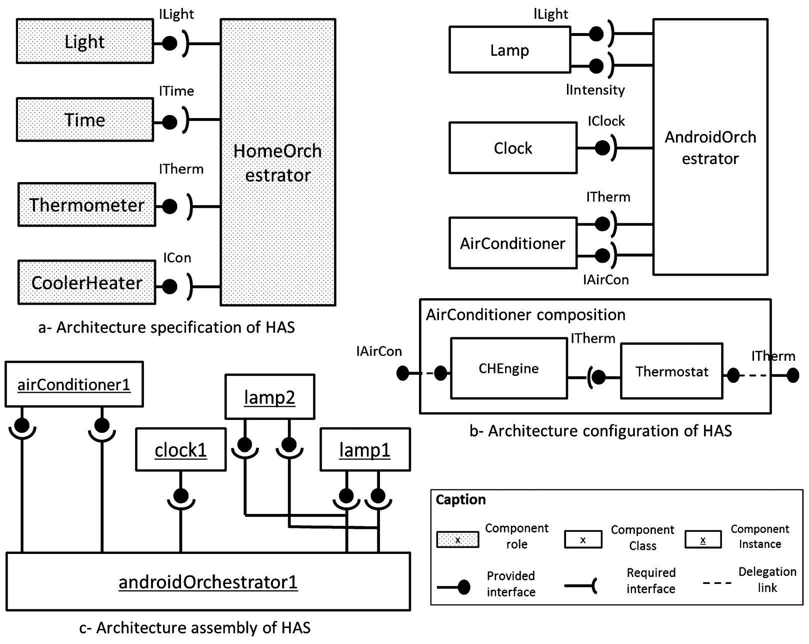


Fig. 2. Architecture specification, configuration and assembly of the HAS.

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 and connects the concrete component classes that compose a specific version of the software. In Dedal, component classes can either be 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 the unconnected interfaces of its inner components. Fig. 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

MACHINE
name and eventually parameters
INCLUDES (optional)
imported specifications
(Static/declarative part)
SETS
declaration of abstract / enumerated sets
CONSTANTS
declaration of constants
PROPERTIES
constraints on constants
VARIABLES
declaration of variables (the machine state)
INVARIANT
declaration of invariant properties of the machine
DEFINITIONS (optional)
construction of formulas / sets using the variables of the machine
(Dynamic part)
INITIALISATION
initialization of the state of the machine (all declared variables)
OPERATIONS
definition of operations that modify the state of the machine

Fig. 3. Structure of an abstract B machine.

roles from the architecture specification as with the *AirConditioner* component class, which realizes both the *Thermometer* and *CoolerHeater* roles. Conversely, a component class may provide more services than those listed in (its role in) the architecture specification as with the *Lamp* component class which provides an extra service to control the intensity of light. These extra interfaces may be left unconnected.

The *instantiated architecture assembly* describes software at runtime and holds information about its internal state. The architecture assembly models an instantiation of its 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 the maximum number of connected instances). *Component instances* document how the component classes from an architecture configuration are instantiated in the deployed software. Each component instance has an initial and a current state defined by a list of valued attributes. Fig. 2-c shows an instantiated architecture assembly for the Has example.

2.2. The B modeling language

B [9,10] is a formal modeling language and a proof-based development method for software systems. The principle of such method is to start from a very abstract model of the system and then gradually refine it. Initially designed by Abrial in 1985 to specify critical systems, B was rapidly adopted by industry and used in many case studies such as the Meteor project [11] for controlling train traffic and the Pci protocol [12]. B is also widely used and studied in academia, mainly as a formal modeling language for verification, validation and model-checking.

2.2.1. Expressiveness and semantics

B is based on Zermelo–Fraenkel (ZF) set theory and first order logic language. The B notation is very similar to mathematical language and includes all standard logical connectors (*e.g.* $\wedge$, $\vee$, $\Rightarrow$), set-theoretic operations (*e.g.* $\in$, $\cup$), closure and specific relations like injective ($\rightarrowtail$), surjective ($\twoheadrightarrow$) and bijective ($\rightarrowtail\!\!\!\!\twoheadrightarrow$) functions. B also supports sequences and the basic boolean (*BOOL*), integer (*INTEGER*) and natural (*NAT*) types.

B specifications are composed of abstract machines similar to modules (*cf.* Fig. 3). They are defined independently and can be reused as modules and refined to obtain more concrete models. An abstract machine is divided into a declarative part and a dynamic part. The declarative part contains the declaration of sets (*SETS*), constants (*CONSTANTS*), variables (*VARIABLES*) which represent the state of the machine and invariant properties (*INVARIANT*) related to variables. Optionally, it is also possible to set definitions (*DEFINITIONS*) (like macros). Definitions are useful to define extensive sets and parametrized predicates and can be reused by invariants and operations. The dynamic part contains the initialization (*INITIALISATION*) of the machine as well as operations (*OPERATIONS*) over the state (variables) of the machine. The behavior of operations is explicitly defined in B using various constructs such as preconditions (*PRE P THEN S END*), bounded choice (*CHOICE S1 OR S2*) or non-determinism (*ANY v WHERE P THEN S END*). Post-conditions are expressed by substitutions that state the new assignments of the involved variables. Output variables may also be defined as values returned by operations.

2.2.2. Tool support for B

B is supported by powerful tools like AtelierB [13], BToolkit [14] and the more recent Bware platform [15]. These tools focus on theorem-proving but they do not enable model-checking. ProB [16] was designed for this purpose. It is a model checker and animator for B models. It automatically generates counterexamples for given assertions by exhaustively exploring the model (using state space exploration techniques). It also simulates the execution of operations on a given subset of the model and generates traces leading to some desired state. An API is also provided for developers to integrate the features of ProB in their tools.

2.3. Motivation and contribution

Component reuse helps decrease large-scale software systems time-to-market. Handling the evolution in such component-based software prevents architecture erosion and has long been identified and still remains an important thus difficult task [17,18]. To tackle this issue, this paper proposes an approach to manage the evolution of component-based software architectures based on the three-level Dedal architecture model.

Dedal is tailored for reuse [6,7] and provides as an original feature its three architecture definition levels. Indeed, specifications are the cornerstone of the concrete component search that is performed on component repositories to design, by reuse, the implementation of architectures. Along with Dedal configurations and assemblies, Dedal architecture definitions keep track of all the design decisions taken during the development process. This information is very useful to control evolution and evaluate its impact on the intentions of the architects. This is why Dedal is a choice ADL for architecture-based software evolution management.

The evolution process proposed here is driven by an evolution management model that captures changes initiated at any abstraction level, controls their impact to preserve / restore consistency and propagates them to other levels to maintain global coherence. This model is based on the B formal language which provides a rich and rigorous notation to formalize the architectural concepts and express properties over them. It supports automated analysis and model-checking thanks to the ProB tool.

In previous work [19,20], we specified Dedal models using the B modeling language and proposed an evolution management model to enable the simulation, analysis and validation of evolution scenarios at any abstraction level using ProB. At that time, evolution was not yet automated since models were specified and evolved manually and separately. In the remainder, our approach integrating both Dedal and B to automatically manage component-based architecture evolution is presented. The automated Dedal to B transformation as well as a problem-specific B solver built on top of the ProB tool are the cornerstones of the contribution of this paper.

Using our problem-specific solver enables the automatic generation of evolution plans (sequences of change operations) to leverage the impact of a change request in a problem-specific manner and maintain the architecture descriptions coherent after change. The feasibility of our approach is demonstrated by experimenting on three evolution scenarios that each addresses change in a different abstraction level.

3. The formal evolution approach

This section presents our approach to formally handle the evolution of multi-level component-based architecture descriptions produced during software development. Its key idea is to use a B solver to automatically generate evolution plans that correspond to intended changes (*cf.* Fig. 4).

Given a model in an initial state, a set of state transition rules and a goal state, a B solver finds sequences of rules that reach the goal state or proves that the goal state cannot be reached (when it does not run out of time or resources because of high computational complexity).

A first requirement is thus to transform the Dedal models produced during development into B models that can be used as an input for the solver. The principles of this transformation are detailed in Section 3.1. Architecture evolution operations along with validation properties must also be expressed as a set of rules. The resulting *Evolution Management Model* is presented in Section 3.2. Finally, initiated architecture changes must be described as goal states, as explained in Section 3.3. With these inputs, a B solver can then find an evolution plan (a sequence of rules) that achieves the intended change (reaches the goal state) while preserving the coherence of the architecture definition (enforcing properties), as presented in Section 3.4.

3.1. Dedal to Formal Dedal transformation

Dedal models need to be translated into B models, so that a B solver can calculate modifications and evaluate properties on the resulting formal architecture descriptions. Defining this transformation amounts to formalize in B the concepts of the Dedal meta-model (*cf.* Fig. 6). This way, any instance of the Dedal meta-model can be transformed into an equivalent instance of the Formal Dedal meta-model (*cf.* Fig. 5).

A meta-class is usually mapped to a B variable typed by an abstract B set while an association relation is translated into a B relation. For instance, Fig. 7 presents the formalization of the *Component* and *Interface* meta-classes and their *compInterfaces* association.

The *Component* and *Interface* meta-classes are respectively mapped to the *component* and *interface* variables and typed with the *COMPS* and *INTERFACES* abstract sets. Their *compInterfaces* association holding a one-to-many relation is translated into an injective function between the *component* variable and a non-empty set of interfaces: $\mathcal{P}_1(interface)$.

The whole Dedal meta-model formalization results in four main B machines (extracts of which are shown in Fig. 8). A generic *Arch_concepts* machine helps define the three specific *Arch_specification*, *Arch_configuration* and *Arch_assembly* ma-

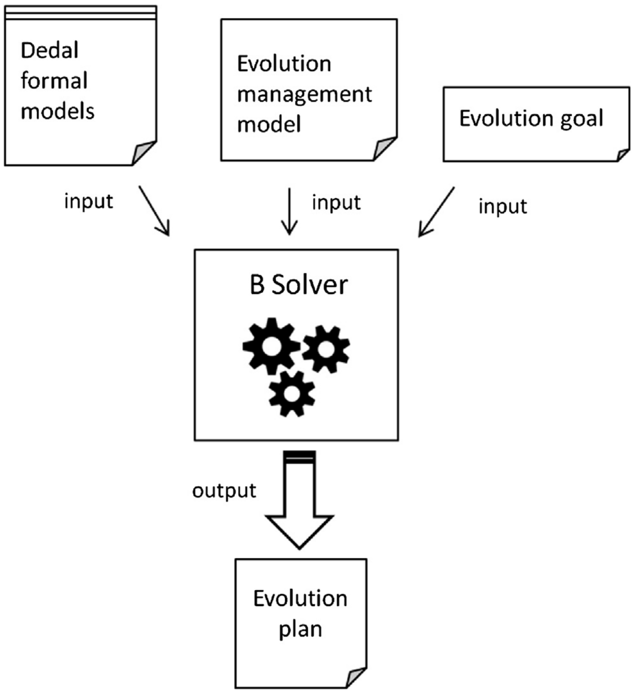


Fig. 4. Evolution management approach.

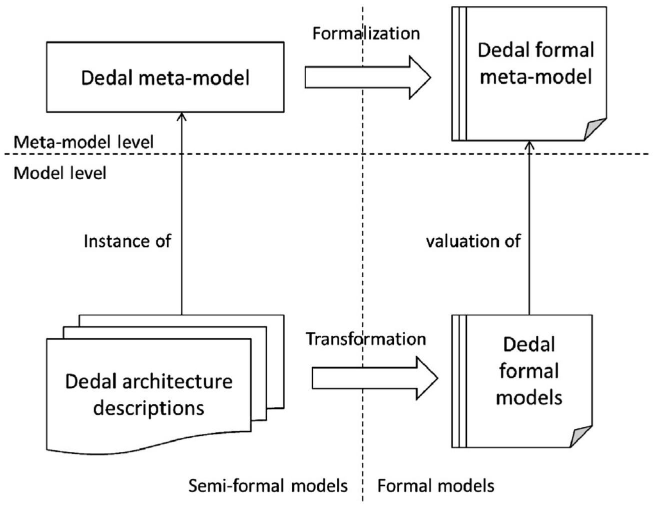


Fig. 5. Dedal to Formal Dedal transformation.

chines that each correspond to one of the three architecture description levels of Dedal. *Arch_concepts* covers the generic concepts of a software architecture (corresponding to the abstract *Component*, *Connection*, and *ArchitectureDescription* meta-classes). It includes an inner *Basic_concepts* machine that contains definitions for the finer-grained architectural elements like *Interface*, *InterfaceType*, *Signature* or *Parameter* meta-classes.

These generic definitions are reused in the three specific machines. For instance, in the *Arch_specification* machine, component roles are defined as a subset of components: $COMP\_ROLES \subseteq COMPS \wedge compRole \subseteq COMP\_ROLES$.

This corresponds to the inheritance relation between the *Component* and *CompRole* meta-classes. Consequently, all relations defined for the *component* set (such as *comp_interfaces*) also stand for the *compRole* set. The abstract B machines define a formal meta-model that can be instantiated (concrete values are given to their variables) in order to generate a Formal Dedal model. The latter is then used as an input for the B solver.

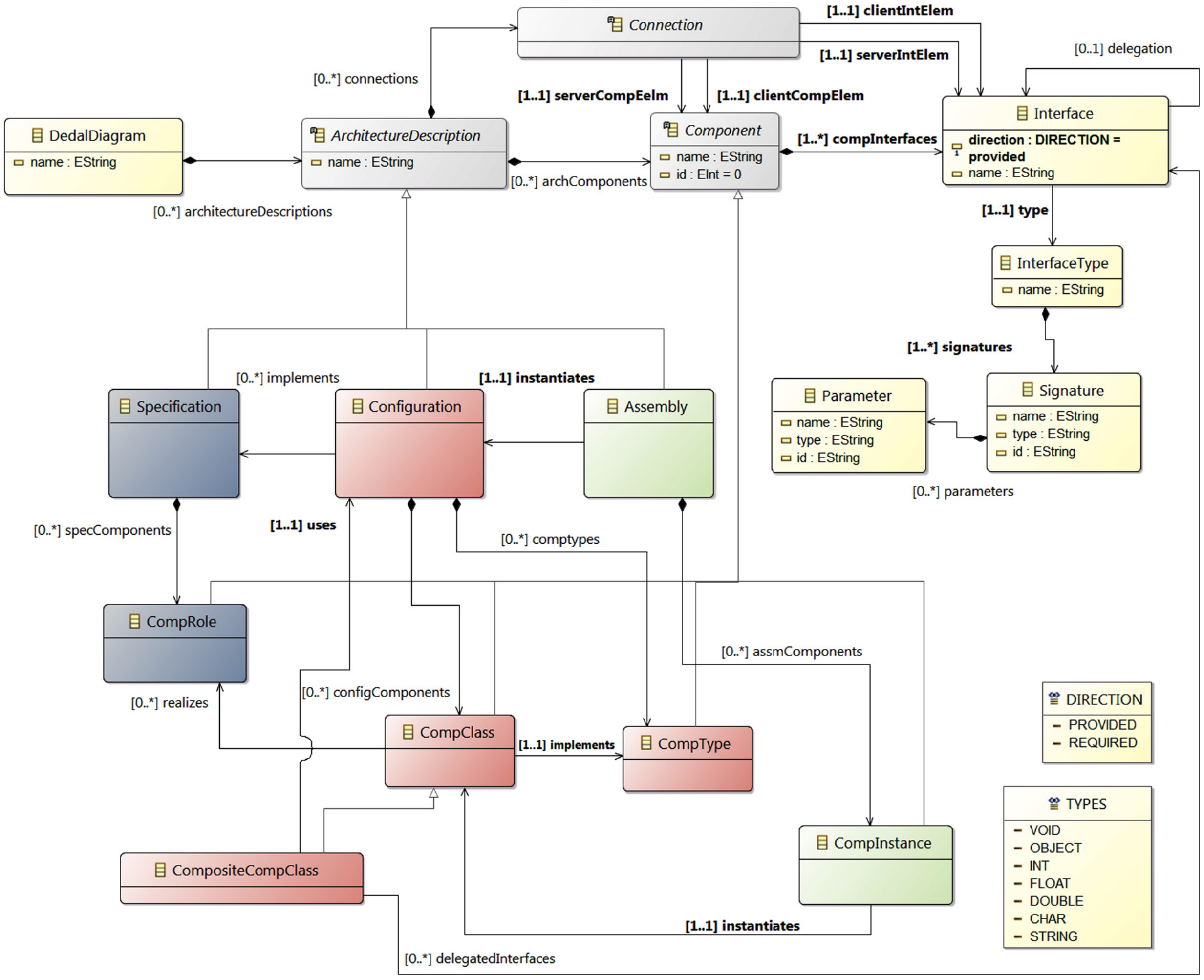


Fig. 6. Dedal meta-model.

SETS
COMPS; *INTERFACES*
VARIABLES
component, *interface*, *comp_interfaces*
INVARIANT
$component \subseteq COMPS \wedge$
$interface \subseteq INTERFACES \wedge$
$comp\_interfaces \in component \rightarrowtail \mathcal{P}_1(interface)$

Fig. 7. Formalization of meta-classes and associations in B.

3.2. *The evolution management model*

The evolution management model is composed of generic evolution rules that are used by the solver to find evolution plans satisfying given evolution goals. It consists in a B machine that defines the rules and properties that respectively enable the simulation and validation of architecture evolution at the three abstraction levels (*cf.* Fig. 9). Its main elements are detailed in the following subsections.

3.2.1. *Evolution rules*

Evolution rules are operations that control the access and the impact of architecture manipulation operations in order to manage evolution and generate consistent evolution plans (*cf.* Fig. 10). Each evolution rule embeds a corresponding architecture manipulation operation that handles the actual modification of the model, not taking into account the context of the current evolution plan.

The evolution rule preconditions act as a primary filter for model manipulation operations. Initialization preconditions check that all the model initialization operations have completed before starting calculating evolution plans.

```
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, ...
INVARIANT
component ⊆ COMPS ∧
comp_name ∈ component → COMP_NAMES ∧
comp_interfaces ∈ component ⤖ 𝒫1(interface) ∧
client ∈ component ↔ interface ∧
server ∈ component ↔ interface ∧
connection ∈ client ↔ server ∧
architecture ⊆ ARCHS ∧
arch_components ∈ architecture → 𝒫(component) ∧
arch_connections ∈ architecture → 𝒫(connection)
```

```
MACHINE
Arch_specification
INCLUDES
Arch_concepts
CONSTANTS
COMP_ROLES, ARCH_SPEC
PROPERTIES
COMP_ROLES ⊆ COMPS ∧
ARCH_SPEC ⊆ ARCHS
VARIABLES
compRole, specification, ...
```

```
MACHINE Arch_configuration
USES Arch_specification
SETS
COMP_CLASS; CLASS_NAME; ATTRIBUTES; ...
CONSTANTS
COMP_TYPES
PROPERTIES
COMP_TYPES ⊆ COMPS ∧
COMP_TYPES = COMPS − COMP_ROLES
VARIABLES
config, config_components, config_connections,
compType, compClass, compositeComp, class_name,
attribute, class_attributes, composite_uses,
delegatedInterface, delegation, ...
INVARIANT
compType ⊆ COMP_TYPES ∧
compClass ⊆ COMP_CLASS ∧
class_name ∈ compClass → CLASS_NAME ∧
attribute ⊆ ATTRIBUTES ∧
class_attributes ∈ compClass → 𝒫(attribute) ∧
compositeComp ⊆ compClass ∧
composite_uses ∈ compositeComp → config ∧
delegatedInterface ⊂ interface ∧
delegation ∈ delegatedInterface ⤖ interface ∧
...
```

```
MACHINE Arch_assembly
USES Arch_configuration
SETS
COMP_INSTANCES; ASSEMBLIES;
ATTRIBUTES_VALUES
VARIABLES
compInstance, assm_components, assm, current_state,
attribute_value, ...
INVARIANT
compInstance ⊆ COMP_INSTANCES ∧
attribute_value ∈ attribute → ATTRIBUTES_VALUES ∧
current_state ∈ compInstance → 𝒫(attribute_value) ∧
assm ⊆ ASSEMBLIES ∧
assm_components ∈ assm → 𝒫1(compInstance)
...
```

Fig. 8. Overview of the Dedal formal meta-model.

```
MACHINE
EvolutionManager
INCLUDES
Arch_specification, Arch_configuration, Arch_assembly
SETS
/*Enumerated set to indicate the level of change*/
  CHANGE_LEVEL = {eLevel, specLevel, configLevel, asmLevel}
VARIABLES
/*Variable to control the level of change*/
changeLevel, ...
DEFINITIONS
/*Consistency and coherence properties*/
...
global_consistency == spec_consistency ∧ config_consistency ∧ assm_consistency
global_coherence == specConfigCoherence ∧ configAssmCoherence
/*GOAL is the predicate given to the solver to find an evolution plan satisfying it*/
GOAL == global_consistency ∧ global_coherence ∧ ...
INITIALISATION
/*Initialization is used to set the initial level of change and
the initiated change*/
...
OPERATIONS
/*Initialization operations*/
...
/*Evolution rules (control the architecture manipulation operations) */
...
END
```

Fig. 9. The *EvolutionManager* machine.

```
output ← evolutionRuleName(targetArchitecture, artifacts) =
PRE
    $initialization = true \wedge$
    $changeLevel = currentChangeLevel \wedge$
    $artifacts \notin addedArtifacts \cup deletedArtifacts \wedge$
    $manipulationOperationPrecondition$
THEN
    /* execute manipulationOperationName(targetArchitecture, artifacts),
    update the sets of added artifacts and deleted artifacts,
    set the value of output parameters */
END
```

Fig. 10. Schema of an evolution rule.

```
output ← mng_addRole(spec, newRole) =
PRE
/*Initialization precondition*/
    $initialisation = \mathbf{TRUE} \wedge$
/*Change level precondition*/
    $changeLevel = specLevel \wedge$
/*Precondition to avoid cycles (inverse operation)*/
    $newRole \notin (deletedRoles \cup addedRoles) \wedge$
/*Precondition of the role addition operation*/
    $roleAdditionPrecondition$
THEN
/*Access to role addition operation*/
    addRole(spec, newRole) ||
    $addedRoles := addedRoles \cup \{newRole\}$ ||
    $output := newRole$
END;
```

Fig. 11. The component role addition evolution rule.

Initialization includes calculating and checking relations between architecture elements, such as compatibility and substitution between components and interfaces. Change level preconditions restrict access to the operations related to the current level of change (evolution is managed on one level at a time). History preconditions prevent operations that may generate cycles and then decrease the efficiency of the solver. For instance, deleting and adding the same artifact several times is unnecessary during an evolution process. Similarly, removing an added artifact results in a null operation that may be avoided. History consists of two sets: one for added artifacts and the other for deleted ones. Evolution rules also inform the solver about the artifacts that have to be manipulated after the last executed change operation. This information is used as a heuristic to increase the efficiency of the solver. Heuristics are further discussed in Section 4.1.2.

Fig. 11 gives the definition of the evolution rule that controls the role addition operation. This rule is enabled when evolution is handled at the specification level, after initialization, provided that the role has not yet been added or previously removed. If so, the precondition of the role addition operation is checked and, when it is verified, the operation is executed. Finally, the set of added component roles (*addedRoles*) is updated and the output is set to the added component role (*newRole*).

3.2.2. Model manipulation operations

A model manipulation operation is an operation that changes a target software architecture by the deletion, addition or substitution of one of its elements (components and connections). They are composed of three parts:

- the operation signature that defines the operation name and its arguments,
- preconditions that are related to the architectural model (*e.g.* a precondition that checks if substitutability between two component classes holds),
- actions (called substitutions in B) that update a set of variables related to the architectural model (*e.g.* the set of components of the architecture).

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 manipulation operations related to the specification level are the addition, deletion and substitution of a component role and the addition and deletion of connections. Fig. 12 presents the definition of the role addition operation as an example of an architecture specification manipulation operation. Its precondition first checks that arguments are soundly typed and then that the chosen role does not already belong to the architecture specification and will not name clash. Its actions update the set of component roles of the architecture specification, along with the sets of connected provided and required interfaces (respectively *spec_components*, *spec_servers* and *spec_clients*). Indeed, as only effectively used elements are defined at specification level, every interface must be connected.

```
$\mathbf{addRole}(spec, newRole) =$
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*/
$\forall cr.(cr \in compRole \wedge cr \in spec\_components(spec)$
$\Rightarrow comp\_name(cr) \neq comp\_name(newRole))$
THEN
    $\mathbf{spec\_servers}(spec) := spec\_servers(spec) \cup \text{servers}(\text{newRole}) \mid\mid$
    $\mathbf{spec\_clients}(spec) := spec\_clients(spec) \cup \text{clients}(\text{newRole}) \mid\mid$
    $\mathbf{spec\_components}(spec) := spec\_components(spec) \cup \{newRole\}$
END;
```

Fig. 12. The component role addition manipulation operation.

```
$\mathbf{replaceClass}(config, oldClass, newClass) =$
  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
    $\mathbf{config\_components}(config) := (config\_components(config) - \{oldClass\}) \cup \{newClass\}$
  END
```

Fig. 13. The component class substitution manipulation operation.

```
$\mathbf{deployInstance}(asm, inst, class, state) =$
  PRE
    $asm \in assembly \wedge class \in compClass \wedge$
/* The instance is a valid instantiation of the chosen 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 of its attributes
    of the instantiated component class*/
    $state \in \mathcal{P}\ (attribute\_value) \wedge \mathbf{card}(state) = \mathbf{card}(class\_attributes(class)) \wedge$
/* 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*/
    $initial\_state(inst) := state \mid\mid$
    $\mathbf{current\_state}(inst) := state \mid\mid$
/*updating the number of instances and the assembly architecture*/
    $\mathbf{nb\_instances}(class) := nb\_instances(class) + 1 \mid\mid$
    $\mathbf{assm\_components}(asm) := assm\_components(asm) \cup \{inst\} \mid\mid$
    $\mathbf{assm\_clients}(asm) := assm\_clients(asm) \cup clients(inst)$
  END;
```

Fig. 14. The component instance deployment manipulation operation.

Architecture configuration evolution. Change can be initiated at the configuration level, for example when new versions of software component classes are released or when component classes are not available anymore. 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. Fig. 13 presents the component class substitution operation as an example of an architecture configuration manipulation operation.

Besides checking the type of the arguments, its precondition verifies that the new component class does not already belong to the configuration and can be a substitute for the old component class (using the relations calculated during initialization). When the precondition is verified, the set of component classes composing the configuration is updated. As compared to the role addition operation presented in previous section, there is no need to update the sets of client and server interfaces (connected required and provided interfaces) here, as substitution must preserve the connections of the replaced component class (*see* § 3.2.3 for deeper insight about substitution rules).

Architecture assembly evolution. Since the architecture assembly represents the software at runtime, managing the assembly level relates to dynamic evolution issues. Indeed, some software systems have to be self-adaptive to keep providing their functions despite environmental changes (*e.g.* lack of resources, failures, user requests). Dealing with unanticipated changes is one of the most important issues in software evolution. This issue is 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 and coherence when needed. The assembly manipulation operations include component instance addition, component instance removal, component instance substitution and component instance connection / disconnection. Fig. 14 gives the definition of the component instance addition as an example of an assembly manipulation operation. After checking the types of the arguments, the precondition verifies that the instance corresponds to the chosen component class,

$$\begin{array}{l} \forall\, (cl, se).(cl \in client \wedge se \in server \Rightarrow \\ \quad ((cl, se) \in connection \Rightarrow \\ \quad \exists\, (C_1, C_2, int_1, int_2).(C_1 \in component \wedge C_2 \in component \wedge C_1 \neq C_2 \wedge \\ \quad int_1 \in interface \wedge int_2 \in interface \wedge cl = (C_1, int_1) \wedge se = (C_2, int_2) \wedge \\ \quad (int_1, int_2) \in int\_compatible))) \end{array}$$

Fig. 15. Interface consistency property.

that it does not already belong to the assembly and that another instance of the class can be added in the assembly. It also verifies that the chosen initial state is valid.

When executed, the operation adds the instance in the assembly, updates the count of instances of the component class and updates the set of client interfaces. The set of server interfaces will be updated later, as client interfaces are automatically connected by the evolution manager to maintain the consistency of the assembly (*see* § 3.2.3). Manipulation operations constitute the dynamic aspect of the architectural formal models. They enable to change the state of a model which must therefore be validated thanks to consistency and coherence properties exposed in the following sections.

3.2.3. *Consistency properties*

Consistency properties maintain the correctness of each architecture description level during the evolution process. Taylor et al. [21] define consistency as an internal property intended to ensure that different elements of an architecture model do not contradict one another. They point out five kinds of inconsistencies that may occur in architecture models: name, interface, behavior, interaction and refinement. Our consistency properties deal with the following inconsistencies:

- *Name consistency* ensures that each component holds a unique name to avoid conflicts when selecting components.
- *Interface consistency* ensures that all architecture connections are correct (*i.e.* a required interface is always connected to a compatible provided interface).
- *Interaction consistency* ensures that the architecture realizes its functional objectives (components are able to soundly cooperate through their connected interfaces). In our approach, this property is implemented as a verification that each required interface is connected to a compatible provided one. Moreover, in architecture specifications, all server interfaces must also be connected (no unused feature is described at this level). Besides, every architecture definition must be composed of a connected graph, so that no part of the architecture is isolated.

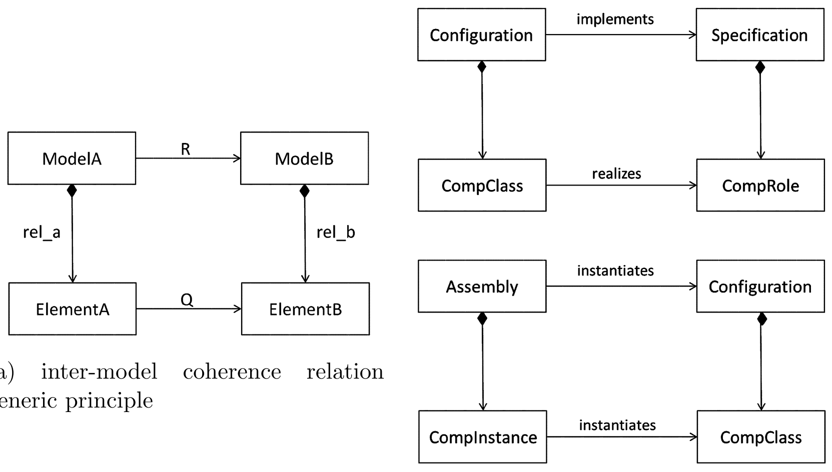
Behavior consistency is out of the scope of the work presented in this paper which only considers static type definitions, for now. *Refinement consistency* is handled separately by our coherence properties (*cf.* Section 3.2.4). As an example, the formalization of our interface consistency property is presented in Fig. 15.

This property states that a required (client) interface is properly connected to a provided (server) interface when these two interfaces belong to different components and have compatible types.

Consistency properties are based on commonly adopted syntactic typing rules that state compatibility and substitution between finer grained entities such as components and interfaces. These rules transpose the well studied typing principles used in the object-oriented paradigm to the component-oriented paradigm. As usual, the main principle is that a component that belongs to a subtype can substitute for a component that belongs to a supertype (*i.e.* be connected at the same place in the same architecture). This entails that a component subtype must define a set of interfaces that can replace all the interfaces defined in its supertype (identical interfaces or interfaces belonging to subtypes). Moreover, a component subtype cannot define extra required interfaces, as they correspond to extra connection requirements that break the substitution guarantee with the supertype. Conversely, extra provided interfaces can be defined in a subtype as they do not imply mandatory extra connections.

Comparing component types thus amounts to comparing interface types. Interface type hierarchies are built with respect to the same substitution principle: an interface subtype must define a set of operations that can replace those of its supertypes. Usual specialization rules are applied to provided interface types, that are comparable to object types. A provided interface subtype must define at least the same operations as its supertypes or specialized operations that can replace them. Classically, an operation specializes another one when it has the same name, a contravariant set of input parameters (at most as many parameters, with identical or more generic data types) and a covariant set of output parameters (at least as many parameters, with identical or more specific data types). With these rules, it is always possible to call a more specialized operation with the input values of a more generic one and then to use the output values of the more specialized operation in place of the output value of the more generic one.

Regarding required interfaces, opposite specialization rules are used. Indeed, a required interface corresponds to dependencies. Thus, a required interface subtype cannot define more operations than its supertypes, in order not to add extra dependencies. It cannot define less operations either, as this can impair interactions with other components. A required interface subtype must then implement the same operations as its supertypes, or more generic operations (*i.e.* operations with the same name, at least as many input parameters of identical or more specific data types and at most as many output parameters with identical or more generic data types). Requiring more generic operations than its supertypes, a more specialized required interface can replace a more generic required interface. Dedal typing rules are discussed and detailed in previous work [22].



(a) inter-model coherence relation generic principle

(b) coherence relations between the Dedal architecture levels

Fig. 16. Coherence relations between architecture levels.

$$\begin{array}{l} coherence(model_A, elem_A, model_B, elem_B, rel_a, rel_b, R, Q) == \\ \forall(M_a, M_b).(M_a \in model_A \wedge M_b \in model_B \Rightarrow ((M_a, M_b) \in R \\ \quad \Leftrightarrow \\ (\forall e_b.(e_b \in elem_B \wedge (M_b, e_b) \in rel_b \Rightarrow \\ \quad \exists e_a.(e_a \in elem_A \wedge (M_a, e_a) \in rel_a \wedge (e_a, e_b) \in Q))))) \end{array}$$

Fig. 17. Generic coherence rule.

$$\begin{array}{l} implements \in configuration \leftrightarrow specification \wedge \\ coherence(configuration, compClass, specification, compRole, \\ \quad config\_components, spec\_components, implements, realizes) \end{array}$$

Fig. 18. Implementation coherence property using the generic rule.

Compatibility is calculated thanks to the aforementioned typing rules. Basically, a required interface is compatible with a provided interface when they have the same type (*i.e.* are defined by the same set of operations). The required interface is also compatible with a provided interface that belongs to a subtype of its type (because of the substitution principle). Compatibility rules are also detailed in [22].

3.2.4. Coherence properties

Coherence properties prevent architecture erosion (mismatches between the different description levels) so as to maintain the global correctness of architecture definitions. Coherence properties maintain the relations that must exist between the specification, configuration and assembly defining an architecture (*cf.* Fig. 16-b): its configuration must be a valid implementation of its specification; its assembly must be a valid instance of its configuration. These relations between description levels rely on typing relations between their composing elements. The component classes composing the configuration of an architecture must implement the component roles of its specification. In the same way, the component instances composing its assembly must be valid instances of the component classes of its configuration. This relates to a generic principle (*cf.* Fig. 16-a) that a relation between two kinds of models implies a relation between their composing elements (and possibly reciprocally under restrictive conditions). For instance, a model can be considered as a specialization of another model only when its composing elements specialize the elements of the other model. The generic principle can be formalized by the generic coherence rule depicted in Fig. 17.

In our work, two properties are defined in the *Evolution Management Machine* to assert the coherence of an architecture definition: coherence between configuration and specification and coherence between assembly and configuration.

Coherence between configuration and specification. 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 configuration is coherent with a specification when two properties hold:

- all component roles from the specification are realized by component classes in the configuration. This results in a many-to-many relation as several component roles may be realized by a single component class while, conversely, several component classes may be needed to realize a single role. Using the generic coherence rule (*cf.* Fig. 17), this first property can be expressed as shown in Fig. 18.

$$
\begin{array}{|l}
implements \in configuration \leftrightarrow specification \wedge \\
\forall\,(Conf, Spec).(Conf \in configuration \wedge Spec \in specification \Rightarrow \\
(Conf, Spec) \in implements \\
\Leftrightarrow \\
\forall\, CR.(CR \in compRole \wedge CR \in spec\_components(Spec) \Rightarrow \\
\exists\, CL.(CL \in compClass \wedge CL \in config\_components(Conf) \wedge \\
(CL, CR) \in realizes)))
\end{array}
$$

Fig. 19. Implementation coherence property (expanded).

$$
\begin{array}{|l}
conform \in specification \leftrightarrow configuration \wedge \\
coherence(configuration, server, specification, server, \\
\quad config\_servers, spec\_servers, conform, int\_substituion') \\
\text{where:} \\
(s, s') \in int\_substitution' \Leftrightarrow (serverInterfaceElem(s), serverInterfaceElem(s')) \in int\_substitution
\end{array}
$$

Fig. 20. Provided interface connection coherence property.

$$
\begin{array}{|l}
instantiates \in assembly \rightarrow configuration \wedge \\
coherence(assembly, compInstance, configuration, compClass, \\
\quad assm\_components, config\_components, instantiates, comp\_instantiates) \\
\wedge \\
coherence(configuration, compClass, assembly, compInstance, \\
\quad config\_components, assm\_components, instantiates^{-1}, comp\_instantiates^{-1}) \\
\text{where:} \\
instantiates^{-1} \text{ and } comp\_instantiates^{-1} \text{ are the respective reverse relations of } instantiates \text{ and } comp\_instantiates
\end{array}
$$

Fig. 21. Configuration instantiation coherence property.

To illustrate the instantiation of the generic coherence rule, we give the expansion of the implementation coherence property in Fig. 19. In the remainder (Fig. 20 and Fig. 21), only the generic coherence rule is used.

- each connected provided (*server*) interface in the configuration is defined in the specification. This prevents having a configuration that implements extra functions not specified at the higher level which leads to architectural drift or erosion (*cf.* Fig. 20).

Coherence between assembly and configuration. As the definition of an assembly is not obtained from a configuration by an instantiation process (assemblies are defined at design-time), coherence between assembly and configuration descriptions must be checked *a posteriori* explicitly. An assembly is coherent with a configuration when every class of the configuration is instantiated at least once in the assembly and, conversely, every component instance in the assembly is a valid instance of a component class of the configuration (*cf.* Fig. 21).

3.3. Evolution goal

The evolution goal (GOAL) consists in a predicate definition that the solver will attempt to satisfy by searching for a valid sequence of evolution rules (evolution plan) to execute on the architecture. The evolution goal consists of a static and a variable part. The static part contains all the consistency (*global_consistency*) and coherence (*global_coherence*) properties: the calculated evolution plan must maintain the validity of the architecture. The variable part contains the arguments of the initiated change: the evolution plan must achieve the intended change. For example, if the initiated change consists in the addition of a component role *cr* in a specification *spec*, the evolution goal would be the following:

$$GOAL == global\_consistency \wedge global\_coherence \wedge cr \in spec\_components(spec)$$

3.4. Evolution plan generation

Our evolution process distinguishes two kinds of change: initiated change and triggered change. *Initiated changes* have an external source: they originate from a user action or from the execution environment. *Triggered changes* are induced by the evolution manager to restore architecture consistency at each level (they are called *local changes*) and / or global architecture coherence (they are called *propagated changes*), after they have been impacted by an initiated change.

Evolution is handled as a three step process (*cf.* Fig. 22). First, the initiated changes that compose a change request are all processed. These changes all affect a given level of architecture description (called the changed architecture level). In a second step, the impact of these initiated changes are calculated at the changed architecture definition level, thanks to the consistency properties. Maintaining consistency may imply additional (triggered) changes. Finally, the impact of these changes on the other architecture definition levels are calculated thanks to the coherence properties. Maintaining coherence may also imply additional (propagated) changes on the other architecture definition levels.

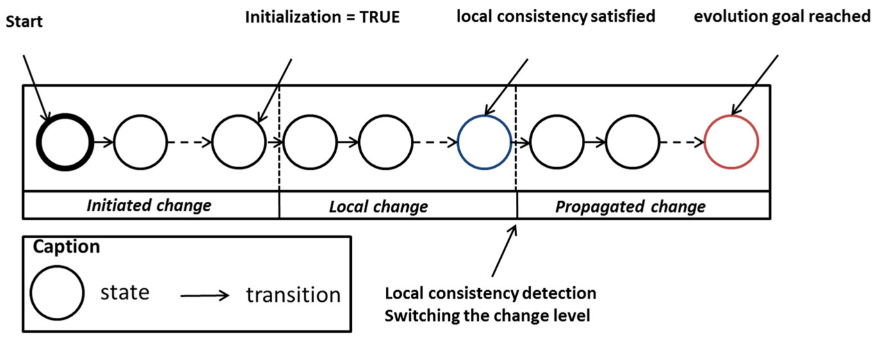


Fig. 22. Evolution plan generation process.

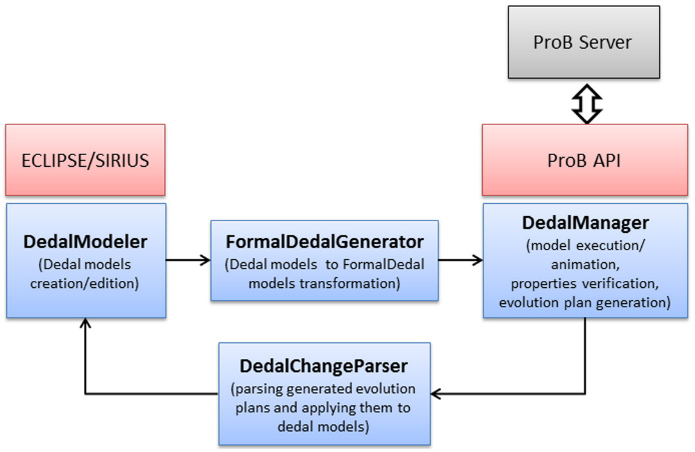

ProB Server
ECLIPSE/SIRIUS
ProB API
DedalModeler
(Dedal models creation/edition)
FormalDedalGenerator
(Dedal models to FormalDedal models transformation)
DedalManager
(model execution/ animation, properties verification, evolution plan generation)
DedalChangeParser
(parsing generated evolution plans and applying them to dedal models)

Fig. 23. Architecture of the Dedal modeling and evolution management environment.

4. Implementation and experimentation

To support our approach, we have implemented DedalStudio, a CASE tool which provides a Dedal modeler, a Formal Dedal generator and an evolution manager based on a solver. Three experiments are then presented in this section to assert the feasibility of our formal evolution approach. Each evolution scenario illustrates a change propagation issue that starts at a different abstraction level, in order to cover the three kinds of multi-level evolution: top-down, bottom-up and mixed. Finally, we evaluate the performance of our solver on the basis of the three experiments.

4.1. DedalStudio

To validate our approach, we have implemented DedalStudio, an Eclipse-based modeling and evolution management environment for Dedal.

4.1.1. Architecture of the tool suite

DedalStudio, the architecture of which is shown in Fig. 23, enables the creation of architecture definitions, using a graphical concrete syntax designed for the Dedal meta-model, composed of Specification Diagrams (SD), Configuration Diagrams (CD) and Assembly Diagrams (AD). The diagram editor (*DedalModeler*), shown in Fig. 24 is based on SIRIUS,[1] a generic platform that enables the creation of graphical modeling tools on top of EMF (Eclipse Modeling Framework).[2] The *FormalDedalGenerator* creates Formal Dedal models corresponding to Dedal diagrams. The *DedalManager* handles the evolution process and the generation of evolution plans. It implements a customized solver built upon the ProB API[3] that enables the animation and model-checking of B models. Finally, the *DedalChangeParser* parses the generated evolution plans and apply the manipulation operations on the Dedal models. All theses tools, except for *DedalModeler* which is targeted to the architect, are fully automatic.

1 https://eclipse.org/sirius/.
2 https://eclipse.org/modeling/emf/.
3 http://stups.hhu.de/ProB/w/ProB_Java_API.

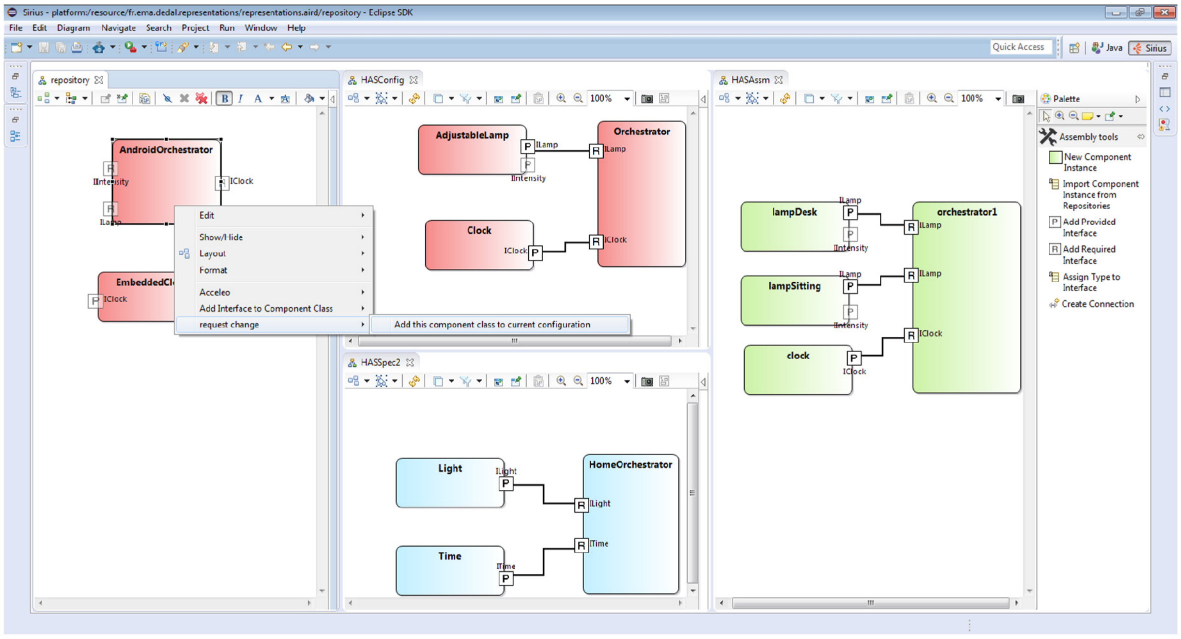


Fig. 24. The DedalModeler tool.

4.1.2. *The DedalManager solver*

Evolution management starts when a change to the architecture model is requested (for instance, a component class addition is requested in the configuration). The *DedalManager* receives the request, identifies the change level and deduces the evolution goal. It then invokes its solver, that conforms to the design principles presented in Section 3. The resolution algorithm implemented in the solver explores the search space to find a sequence of evolution rules leading to the chosen goal. If a solution is found, the *DedalManager* generates an evolution plan that can then be committed by user. Otherwise (*i.e.* in case of failure), the *DedalManager* rejects the change request.

In previous work [20], we have made an evaluation of the performances of the ProB solver to generate evolution plans by state space exploration. The tested strategies were Depth-First (DF), Breadth-First (BF) and mixed (DF/BF) [23]. In most cases DF performed best, better than DF/BF and BF. The ProB solver, however, is general-purpose and increasing resolution time (over 3 minutes) is necessary when models become complex. To try and overcome this problem, this paper proposes an alternative: the implementation of a customized solver, using the API provided with ProB. It also consists in a depth-first search algorithm but enhanced with two specific heuristics: the artifact-oriented heuristic and the operation-oriented heuristic.

The artifact-oriented heuristic. The idea of artifact-oriented heuristic is to prioritize the operations manipulating the artifacts that are more likely to satisfy the evolution goal (thereafter called the main artifacts). For instance, adding a new component usually entails several connection operations on that component to restore architecture consistency. Main artifacts are determined at each iteration of the search process by the output of last executed evolution rule.

The operation-oriented heuristic. The operation-oriented heuristic adopts an opposite point of view. It delays the use of operations that engender unsatisfied dependencies between the components of the architecture and hence more evolution operations to be found in order to reestablish architecture consistency. Addition operations are the most concerned ones. They are therefore ordered as the least priority operations while performing the search process.

The search algorithm. Listing 1 describes the search algorithm of our customized solver. Lines 1–14 define and initialize the main variables of the algorithm. `Transitions` refers to the set of all the evolution rules instances in the current state of the architecture model. The set of already explored transitions is stored in `visited`, in order to avoid cycles in the search process. The current sequence of executed transitions is stored in `pl`, to collect the candidate evolution plan. The traversal of the search graph is handled by `stack`. At each step of the search process, the set of all the enabled transitions (*i.e.* the evolution rule instances whose preconditions are verified) is pushed on the stack in order to explore them in the next steps. Transitions are pushed on the stack along with the current state of the architecture model and the current evolution plan. This enables to backtrack to previous nodes in the search graph and explore other paths when dead ends are reached. The main artifact `a` is used in the evaluation of the artifact-oriented heuristics. The `initialMainArtifact` references the artifact modified by the initiated change. It is calculated from the post-conditions of the corresponding operations.

At each iteration of the search process (lines 17–33), the top of the `stack` is popped (line 19), setting a context consisting of an architecture model state (`s`), an evolution plan (`pl`) and an enabled transition (e_i). If the transition has already

```
1  // initialisation step
2  s = initialState;
3  a = initialMainArtifact;
4  pl = null;
5  stack = null;
6  visited = ∅;
7  enabledTransitions = {e_i ∈ Transitions where pre(e_i) == true};
8  priorTransitions = {e_i ∈ enabledTransitions where h1(e_i) == true};
9  lowpriorTransitions = ∅;
10 enabledTransitions = enabledTransitions - priorTransitions;
11
12 // organizing stack
13 stack.push(s, pl, enabledTransitions);
14 stack.push(s, pl, priorTransitions);
15
16 // starting forward, DF search
17 while (stack ≠ ∅)
18  {
19  (s, pl, e_i) = stack.pop();
20  if ((s, e_i) ∉ visited)
21   {
22   visited = visited ∪ {(s, e_i)};
23   s = execute(e_i);
24   pl = pl+e_i;
25   if (goal == true) return pl;
26   a = output(e_i);
27   enabledTransitions = {e_i ∈ Transitions where pre(e_i) == true};
28   priorTransitions = {e_i ∈ enabledTransitions where h1(e_i) == true};
29   lowpriorTransitions = {e_i ∈ enabledTransitions where h2(e_i) == true};
30   enabledTransitions = enabledTransitions - (priorTransitions ∪ lowpriorTransitions);
31   stack.push(s, pl, lowpriorTransitions);
32   stack.push(s, pl, enabledTransitions);
33   stack.push(s, pl, priorTransitions);
34   }
35  }
36 return null; // no solution for this change request
```

Listing 1: Search algorithm of the specific solver.

been visited from this state (line 20), another context is popped from the `stack` (this happens when a state can be reached by several paths of the search tree). If the transition has not been explored, it is listed as `visited` (line 22) and executed (line 23), updating the state of the architecture model. The last executed transition is appended to the evolution plan (line 24). If the `goal` is satisfied, an evolution plan has been found and it is returned (line 25). Otherwise, the set of the enabled transitions in the current state is calculated (line 27) as is the set of higher priority enabled transitions (line 28) based on the artifact-oriented heuristic (`h1`). This uses the main artifact defined as the output of the last executed transition (line 26). The set of lower priority enabled transitions is also calculated (line 29), based on the operation-oriented heuristic (`h2`). This enables to push on the `stack` the enabled transitions to be explored depending on the priority determined by our heuristics (lines 31–33). The use of a `stack` enables a DF traversal of the graph: the next iteration of the search process will pop one of the currently enabled transitions, from the current architecture state, trying to extend the search path down to the `goal`. When a dead end is reached (no transitions are enabled in the current state), the search process implicitly backtracks to a previous graph node by popping from the top of the `stack` a previously pushed context. This enables the complete traversal of the search graph (breadth search). The search process is iterated until the `goal` is reached or there is no more transition to explore (line 17). In this latter case, the requested change is rejected (line 36). Three examples of evolution plans calculated by our solver are presented in the next sections.

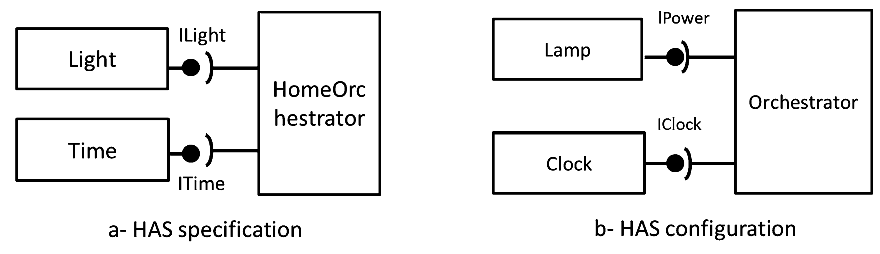
4.2. *First experiment: requirement change*

The first scenario addresses a requirement change. The initial HAS architecture enables to switch on / off the lights at specific hours (*cf.* Fig. 25). However, it does not enable any control on light intensity. To add this new functionality, an architect should modify the HAS specification. This corresponds to a top-down evolution since the change starts at the highest abstraction level. A solution is to replace the *Light* component role by a new one (*Luminosity*) that enables intensity control. Fig. 26 presents the initial architecture specification and the evolved one. An extract of the instantiation of the *Arch_specification* machine corresponding to the HAS is presented in Fig. 27.

4.2.1. *Evolution goal and initiated change*

The initiated change consists in replacing the *Light* component role (*cr1*) by the *Luminosity* component role (*cr1a*). This corresponds to the execution of the role substitution operation on the HAS specification:

spec_replaceRole(*HAS_spec*, *cr*1, *cr*1*a*)



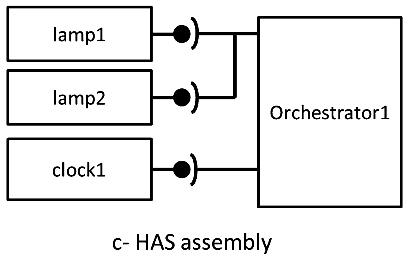


Fig. 25. Architecture definitions of the HAS.

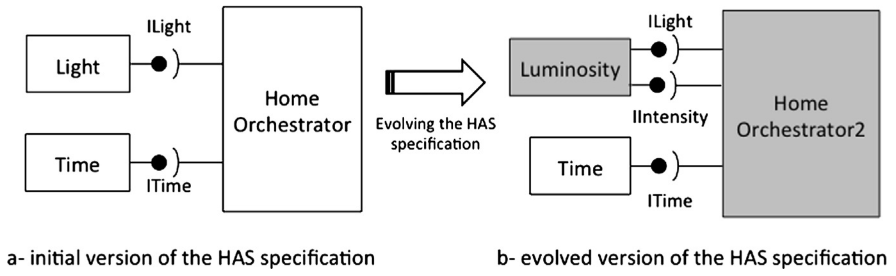

ILight
Light
Home Orchestrator
Time
ITime
Evolving the HAS specification
Luminosity
ILight
IIntensity
Home Orchestrator2
Time
ITime
a- initial version of the HAS specification
b- evolved version of the HAS specification

Fig. 26. Evolving the HAS specification by role replacement.

compRole $:= \{cr1, cr1a, cr2, cr3, cr3a\}$ ||
comp_name $:= \{cr1 \mapsto Light, cr1a \mapsto Luminosity, cr2 \mapsto Time,$
$cr3 \mapsto HomeOrchestrator,$
$cr3a \mapsto HomeOrchestrator2\}$ ||
arch_spec $:= \{HAS\_spec\}$ ||
spec_components $:= \{HAS\_spec \mapsto \{cr1, cr2, cr3\}\}$ ||
spec_connections $:= \{HAS\_spec \mapsto \{$
$((cr3, rintILight) \mapsto (cr1, pintILight)),$
$((cr3, rintITime) \mapsto (cr2, pintITime)), \}\}$ ||
spec_clients $:= \{(HAS\_spec \mapsto \{(cr3, rintILight), (cr3, rintITime)\}\}$ ||
spec_servers $:= \{(HAS\_spec \mapsto \{(cr1, pintILight), (cr2, pintITime)\}$

Fig. 27. Instantiation of the *Arch_specification* machine for the HAS.

The following goal is thus given to the solver, based on the post-conditions of the substitution operation, defining the change that must be achieved by the evolution process:

$GOAL == global\_consistency \wedge global\_coherence \wedge cr1a \in spec\_components(HAS\_spec) \wedge cr1 \notin spec\_components(HAS\_spec)$

The solver then calculates an evolution plan that can restore the consistency and coherence of the architecture that may have been altered by the initial change.

4.2.2. Triggered change

The intended role substitution entails the addition of a new server interface (the *IIntensity* provided interface) which must be connected to restore the consistency of the HAS specification (all interfaces must be connected at specification level). The solver generates the plan presented in Fig. 28 to restore the consistency of the HAS specification.

Change entails the disconnection of all the required interfaces, the deletion of the initial orchestrator (*cr3*), the addition of a new orchestrator (*cr3a*) and finally the connection of all the required interfaces (this is enough to get all the interfaces connected and satisfy the interaction consistency property at specification level). After consistency is verified for specification, change is propagated to the configuration in order to restore the coherence of the architecture definition.

```
spec_disconnect(HAS_spec, (cr3, rintILight), (cr1a, pintILight))
spec_disconnect(HAS_spec, (cr3, rintITime), (cr2, pintITime))
spec_deleteRole(HAS_spec, cr3)
spec_addRole(HAS_spec, cr3a)
spec_connect(HAS_spec, (cr3a, rintILight2), (cr1a, pintILight2))
spec_connect(HAS_spec, (cr3a, rintITime2), (cr2, pintITime))
spec_connect(HAS_spec, (cr3a, rintIIntensity), (cr1a, pintIIntensity))
```

Fig. 28. HAS specification consistency restoration plan.

```
compClass := {cl1, cl1a, cl2, cl3, cl3a, cl2a} ||
comp_name := {cl1 ↦ Lamp, cl1a ↦ AdjustableLamp, cl2 ↦ Clock,
        cl3 ↦ Orchestrator, cl3a ↦ AndroidOrchestrator,
        cl2a ↦ AndroidClock} ||
configuration := {HAS_config} ||
config_components := {HAS_config ↦ {cl1, cl2, cl3}}
config_connections := {HAS_config ↦ {
        ((cl3, rintIPower) ↦ (cl1, pintIPower)),
        ((cl3, rintIClock) ↦ (cl2, pintIClock))}
```

Fig. 29. Initial HAS configuration in Formal Dedal.

```
config_replaceClass(HAS_config, cl1, cl1a)
config_disconnect(HAS_config, (cl3, rintILamp), (cl1, pintILamp))
config_disconnect(HAS_config, (cl3, rintIClock), (cl2, pintIClock))
config_deleteClass(HAS_config, cl3)
config_addClass(HAS_config, cl3a)
config_connect(HAS_config, (cl3a, rintILamp2), (cl1a, pintILamp2))
config_connect(HAS_config, (cl3a, rintIClock2), (cl2, pintIClock))
config_connect(HAS_config, (cl3a, rintIIntensity), (cl1a, pintIIntensity))
```

Fig. 30. Coherence restoration plan for the HAS configuration.

```
compInstance := {ci11, ci12, ci1a1, ci1a2, ci2, ci2a, ci3, ci3a} ||
comp_instantiates := {ci11 ↦ cl1, ci12 ↦ cl1, ci1a1 ↦ cl1a
        ci1a2 ↦ cl1a, ci2 ↦ cl2, ci2a ↦ cl2
        ci3 ↦ cl3, ci3a ↦ cl3} ||
compInstance_name := {ci11 ↦ lamp1, ci12 ↦ lamp2, ci1a1 ↦ adjustableLamp1,
        ci1a2 ↦ adjustableLamp2, ci2 ↦ clock1
        ci3 ↦ orchestrator1, ci3a ↦ androidOrchestrator1,
        ci2a ↦ androidClock1} ||
assembly := {HAS_assembly} ||
assm_components := {HAS_assembly ↦ {ci11, ci12, ci2, ci3}}
assm_connections := {HAS_assembly ↦ {
        ((ci3, rintIPowerInst) ↦ (ci11, pintIPowerInst)),
        ((ci3, rintIClock) ↦ (ci2, pintIClockInst)), ...}
```

Fig. 31. Initial HAS architecture assembly.

4.2.3. Change propagation to the configuration

Coherence is altered due to the new requirement defined by the specification. Indeed, the initial HAS configuration (*cf.* Fig. 25) does not correctly implement all the roles of the evolved HAS specification. Fig. 29 details the instantiation of the *Arch_configuration* machine corresponding to the initial HAS configuration. Change propagation is therefore needed to restore coherence. The restoration plan found by the solver is presented in Fig. 30.

It first consists in replacing the *Lamp* component class by the *AdjustableLamp* component class. This operation does not require any modification of the connections, as it is based on the substitution principle between the two component classes (the *AdjustableLamp* class is a specialization of the *Lamp* class). The situation is different regarding the *Orchestrator* component class. It cannot be simply replaced by the existing *AndroidOrchestrator* component class, which is a valid implementation of the *HomeOrchestrator2* role. Indeed, as it holds an extra required interface, the *AndroidOrchestrator* component class is not a specialization of the *Orchestrator* component class. Nonetheless, the solver is able to find a suitable plan to restore consistency in this more difficult situation. The *Orchestrator* component class is disconnected and removed. The *AndroidOrchestrator* component class is then added and connected. This way, the configuration is consistent (all required interfaces are connected and the configuration is composed of a unique connected graph of components) and coherent with the specification (every role is implemented in the configuration).

4.2.4. Change propagation to the assembly

After coherence is reached in the configuration, change is propagated to the architecture assembly. Here again, coherence is altered because the current HAS assembly is not a valid instantiation of the evolved HAS configuration. Fig. 31 details the initial state of the corresponding *Arch_assembly* machine.

The coherence restoration plan presented in Fig. 32 is generated by the solver to propagate changes. First, the client interfaces of the *Orchestrator* component instance are disconnected. Then, the two *Light* component instances are replaced by *AdjustableLight* component instances (as allowed by the substitution principle). The *Orchestrator* component instance is

assm_unbind(*HAS_assembly*, (*ci*3, *rintILampInst*), (*ci*11, *pintILampInst*1))
assm_unbind(*HAS_assembly*, (*ci*3, *rintILampInst*), (*ci*2, *pintILampInst*2))
assm_unbind(*HAS_assembly*, (*ci*3, *rintIClockInst*), (*ci*12, *pintIClockInst*))
assm_replaceInstance(*HAS_assembly*, *ci*1, *ci*1*a*1)
assm_replaceInstance(*HAS_assembly*, *ci*12, *ci*1*a*2)
assm_removeInstance(*HAS_assembly*, *ci*3)
assm_deployInstance(*HAS_assembly*, *ci*3*a*)
assm_bind(*HAS_assembly*, (*ci*3*a*, *rintILamp*2*Inst*), (*ci*1*a*1, *pintILampInst*1))
assm_bind(*HAS_assembly*, (*ci*3*a*, *rintIIntensity*2*Inst*), (*ci*1*a*1, *pintIIntensityInst*1))
assm_bind(*HAS_assembly*, (*ci*3*a*, *rintIClockInst*), (*ci*2, *pintIClockInst*))
assm_bind(*HAS_assembly*, (*ci*3*a*, *rintILamp*2*Inst*), (*ci*1*a*2, *pintILampInst*2))
assm_bind(*HAS_assembly*, (*ci*3*a*, *rintIIntensity*2*Inst*), (*ci*1*a*2, *pintIItensityInst*2))

Fig. 32. Coherence restoration plan for the HAS architecture assembly.

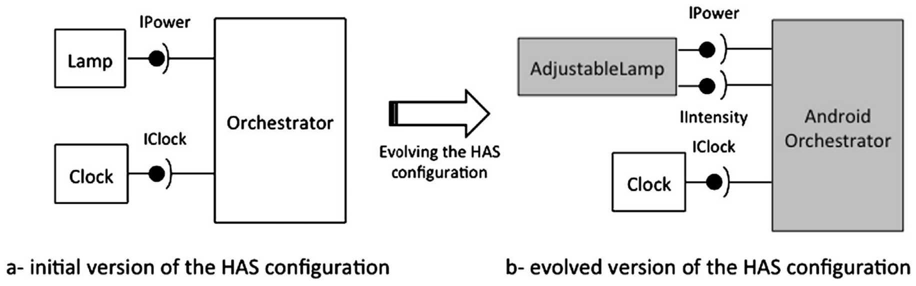

IPower
Lamp
Orchestrator
IClock
Clock
Evolving the HAS configuration
IPower
AdjustableLamp
IIntensity
IClock
Android Orchestrator
Clock
a- initial version of the HAS configuration
b- evolved version of the HAS configuration

Fig. 33. Evolving the HAS configuration by component substitution.

removed and an *AndroidOrchestrator* component instance is added. As explained for the configuration coherence restoration, substitution is not possible because of the extra required interfaces of the *AndroidOrchestrator* component. Fortunately, an evolution plan can still be found so that every component class in the configuration is instantiated at least once in the assembly. Finally, all the required interfaces are connected to compatible provided interfaces, maintaining a consistent assembly.

4.3. *Second experiment: implementation change*

The second scenario addresses an implementation change. The objective is to enable the control of the building through a mobile device (running Android OS for example). To adapt the current implementation to Android, the *Orchestrator* component class (*cl3*) should be removed and replaced with an Android compatible one (*cl3a*). Change is initiated at the configuration level, which entails a mixed evolution: bottom-up because the change has to be propagated to the higher level specification and top-down because it has to be propagated also to the lower assembly level. Fig. 33-a shows the initial implementation of the HAS while Fig. 33-b shows the evolved one.

4.3.1. *Initiated change*

Change is initiated by deleting the initial orchestrator (*cl3*) and adding the Android compatible one (*cl3a*). This is processed by the following sequence of operations:

config_disconnect(*HAS_config*, (*cl*3, *rintILamp*), (*cl*1, *pintILamp*))
config_disconnect(*HAS_config*, (*cl*3, *rintIClock*), (*cl*2, *pintIClock*))
config_deleteClass(*HAS_config*, *cl*3)
config_addClass(*HAS_config*, *cl*3*a*)

To start the evolution process, the following goal is given to the solver:

$GOAL == global\_consistency \wedge global\_coherence \wedge cl3a \in config\_components(HAS\_config) \wedge cl3 \notin config\_components(HAS\_config)$

4.3.2. *Triggered change*

The generated triggered change is listed in Fig. 34. To restore consistency, all component classes must be correctly connected. The *AndroidOrchestrator* component class requires an additional server interface to control the intensity of light. The *Lamp* component class (*cl1*) is suitably replaced with *AdjustableLamp* (*cl1a*) that provides the *IIntensity* server interface. This is another illustration of the solving capabilities of our approach. After configuration consistency is verified, change is propagated to the architecture specification.

4.3.3. *Change propagation to the specification*

The current HAS specification is not any more a good design model of the new version of the HAS configuration. This corresponds to an erosion problem as light intensity control is not included in the current specification. Hence, a new

```
config_connect(HAS_config, (cl3a, rintIClock2), (cl2, pintIClock))
config_replaceClass(HAS_config, cl1, cl1a)
config_connect(HAS_config, (cl3a, rintIPower2), (cl1a, pintIPower2))
config_connect(HAS_config, (cl3a, rintIIntensity), (cl1a, pintIIntensity))
```

Fig. 34. HAS configuration consistency restoration plan.

```
spec_disconnect(HAS_spec, (cr3, rintILight), (cr1, pintILight))
spec_disconnect(HAS_spec, (cr3, rintITime), (cr2, pintITime))
spec_deleteRole(HAS_spec, cr3)
spec_addRole(HAS_spec, cr3a)
spec_replaceRole(HAS_spec, cr1, cr1a)
spec_connect(HAS_spec, (cr3a, rintILight2), (cr1a, pintILight2))
spec_connect(HAS_spec, (cr3a, rintIIntensity), (cr1a, pintIIntensity))
spec_connect(HAS_spec, (cr3a, rintITime2), (cr2, pintITime))
```

Fig. 35. HAS specification coherence restoration plan.

```
assm_unbind(HAS_assembly, (ci3, rintILampInst), (ci11, pintILampInst1))
assm_unbind(HAS_assembly, (ci3, rintILampInst), (ci12, pintILampInst2))
assm_unbind(HAS_assembly, (ci3, rintIClockInst), (ci2, pintIClockInst))
assm_removeInstance(HAS_assembly, ci3)
assm_deployInstance(HAS_assembly, ci3a, cl3a)
assm_replaceInstance(HAS_assembly, ci11, ci1a1)
assm_replaceInstance(HAS_assembly, ci12, ci1a2)
assm_bind(HAS_assembly, (ci3a, rintILamp2Inst), (ci1a1, pintILampInst1a))
assm_bind(HAS_assembly, (ci3a, rintIIntensity2Inst), (ci1a1, pintIItensityInst1))
assm_bind(HAS_assembly, (ci3a, rintIClockInst), (ci2, pintIClockInst))
assm_bind(HAS_assembly, (ci3a, rintILamp2Inst), (ci1a2, pintILampInst2a))
assm_bind(HAS_assembly, (ci3a, rintIIntensity2Inst), (ci1a2, pintIItensityInst2))
```

Fig. 36. HAS assembly coherence restoration plan.

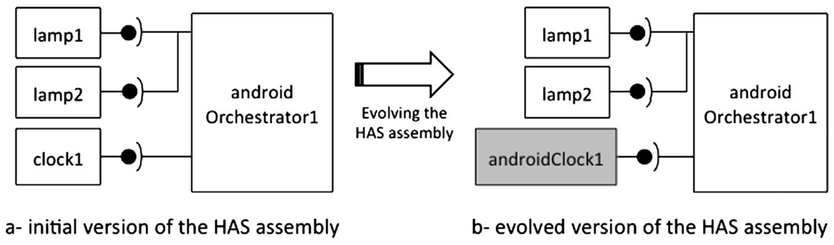

lamp1
lamp2
clock1
android Orchestrator1
Evolving the HAS assembly
lamp1
lamp2
androidClock1
android Orchestrator1
a- initial version of the HAS assembly
b- evolved version of the HAS assembly

Fig. 37. Evolving the HAS assembly by component instance substitution.

specification version is required to keep architecture descriptions coherent. Change is propagated to the HAS specification (*cf.* Fig. 35) by replacing the *HomeOrchestrator* role (*cr3*) with the *HomeOrchestrator2* (*cr3a*). To do so, the *HomeOrchestrator* role is disconnected and deleted. Then the *HomeOrchestrator2* role is added. On the other way, the *Luminosity* role (*cr1a*) can be directly substituted for the *Light* role (*cr1*). This enforces coherence between the specification and the configuration. Finally, the connection of all client interfaces is sufficient to restore the consistency of the specification (no pending interfaces; a unique connected component graph).

4.3.4. *Change propagation to the assembly*

The current version of the HAS assembly is no more a valid instantiation of the evolved HAS configuration. Change has to be propagated at assembly level to restore coherence (*cf.* Fig. 36). In a similar way to specification coherence restoration, the *Orchestrator* component instance (*ci3*) is disconnected and deleted. An *AndroidOrchestrator* component instance (*ci3a*) is added to the assembly. Two *AdjustableLight* component instances (*ci1a1*) and (*ci1a2*) are substituted for the existing *Light* component instances (*ci11*) and (*ci12*). This restores the coherence of the assembly with the configuration. The server interfaces of the components are then bound to compatible provided interfaces, so that the assembly remains consistent (no pending server interfaces; a unique connected component graph).

4.4. *Third experiment: runtime change*

The third scenario addresses a runtime change. It corresponds to a bottom-up evolution since the change is initiated at the lowest abstraction level. Because of a dry battery, the clock device in the building is out of service. This environmental change induces the dysfunction of the *clock1* driver (*ci2*). The objective is to find a solution to dynamically repair the architecture in order to maintain the functionalities of the system. Fig. 37 shows the initial and evolved version of the HAS assembly.

Table 1
Performance evaluation.

	Change level	DF (ms)	H-DF (ms)
Exp 1	specLevel (initial)	3260	2100
	configLevel	3254	1393
	asmLevel	26738	1926
Exp 2	configLevel (initial)	4712	2537
	specLevel	8733	1896
	asmLevel	TIME-OUT	1927
Exp 3	asmLevel (initial)	4747	1184
	configLevel	TIME-OUT	2351
	specLevel (not affected)	–	–

4.4.1. Initiated change

The *clock1* (*ci2*) component instance must be replaced by another component instance that provides the same services. An instance of the *AndroidClock* component class, *androidClock1* (*ci2a*), is thus chosen to replace *clock1*. The initiated change is handled by the following operations:

$replaceInstance(HAS\_assembly, ci2, ci2a)$

The solver then searches an evolution plan that reaches the following goal:

$GOAL == global\_consistency \wedge global\_coherence \wedge ci2a \in assm\_components(HAS\_assembly) \wedge ci2 \notin assm\_components(HAS\_assm)$

4.4.2. Triggered change

The component instance replacement does not alter the consistency of the assembly architecture. However, coherence with the configuration architecture has to be reestablished. Indeed, the evolved assembly architecture is not a valid instantiation of the current configuration architecture since the *ci2a* component instance does not instantiate the *cl2* component class.

4.4.3. Change propagation to the configuration

Change propagation induces the substitution of the *AndroidClock* component class (*cl2a*) for the *Clock* component class (*cl2*), which amounts to the following evolution plan:

$replaceClass(HAS\_config, cl2, cl2a)$

As connections are preserved by the substitution operation, the consistency of the configuration is also preserved. The evolution plan thus includes no other operation.

4.4.4. Change propagation to the specification

The component class substitution preserves the coherence between the specification and the configuration. Indeed, when a component class implements a given role, any component subclass, as a substitute, also implements the role. As a consequence, no change needs to be propagated to the specification.

4.5. Performance evaluation

The performance of the solver has been measured during the three experiments, in order to evaluate the influence of our proposed heuristics. Tests were run on a standard PC (2.5 GHZ Intel Core i5, 8 GB SDRAM) under Windows 7. Test of the three evolution scenarios are then performed first using *DF* and then using *DF* enhanced with heuristics (*H-DF*) to compare the results. Table 1 shows the average time in milliseconds of 5 runs for each evolution scenario, using depth-first search without heuristics (*DF*) and with heuristics (*H-DF*).

Timeout is set to 3 minutes. Results doubtlessly show the benefits of a custom solver that integrates specific heuristics. The order and number of evolution rules may differ from a generated evolution plan to another (our algorithms are not deterministic as they make random choices when sets of equivalent elements are considered, such as a set of candidate main artifacts) but all generated plans are valid and lead to the same goal state.

A more precise performance evaluation, based on a larger set of experiments and a theoretical study of the combinatorial complexity of the search space is needed. Performance is indeed an inherent limitation for search-based software engineering, as the resolution time of solvers generally grows exponentially depending on the size of the problems. Designing and integrating new heuristics to cut down resolution time is promising (we can for instance preferentially choose transitions that generate no or little incoherence in the architecture model).

5. Related work

This section presents three areas of related work. The first area is that of software architecture evolution which is the main theme of this work. It presents a survey of the main state-of-the-art evolution approaches our work can be compared to. The second area is that of formal modeling languages. It presents a brief comparison of seven formal modeling languages including B. The third area describes other approaches based on model transformation and integration of semi-formal and formal methods. These approaches do not necessarily focus on architecture evolution but they present interesting alternatives from the technical point of view.

5.1. Software architecture evolution

Most of the approaches dealing with architecture evolution adopt an Adl to model architectures and propose a mapping between the Adl and a runtime framework in order to implement the change and enable dynamic evolution. C2-SADEL [24], Darwin [25], ArchWare [26] and Plastik [27] fall into this category. C2-SADEL models architectures in the C2 style [28] and provides multiple component subtyping mechanisms to favor reuse and enable architecture evolution. Its tool support is Dradel, an environment that enables the mapping between architectural description and the implementation by translating them into Java code. The tool supports static evolution by applying changes on architectural descriptions first and then implementing them. The architecture analysis however is limited since no powerful analysis techniques were integrated. Darwin and ArchWare (which provides π-Adl [29] as an Adl) focus on modeling dynamic structures. They both rely on π-calculus to define the semantics of architecture constructs and guarantee a reliable interaction between components and compile architecture descriptions into code. ArchWare also proposes π-Arl [30] an architecture refinement language to evolve architecture descriptions by stepwise refinement. Plastik was also proposed to deal with dynamic reconfigurations. It relies on Armani, an extension of the ACME [31] Adl to enable invariants expression and reconfigurations properties. Compared to the previous approaches, Plastik has the advantage to map its Adl to OpenCOM [32], a runtime component model dedicated to component-based programming and proposing built-in reconfiguration operations. The main shortcoming of these approaches is that they don't consider changes as first-class elements and focus more on how to implement architecture evolution rather than specify, analyze and propagate it. Moreover, adopted Adls hardly cover the entire Cbsd process. The specification level (necessary to guide reuse) and assembly level (that describes the software at runtime) are often missing. Finally, the coherence between architectural descriptions and implementation is not guaranteed since evolution is processed top-down only.

Recent work by Sanchez et al. [33] proposes an architecture-based re-engineering approach to evolve and maintain legacy software. The principle is to produce a high level architecture description of the legacy system so that it becomes easy to reason about change and then reversely use the produced knowledge to modify source code. The approach is guided through a bidirectional transformation and relies on Archery [34], an Adl for modeling architecture patterns corresponding to translated code parts. Targeted at legacy system re-engineering, this work is different from our proposal on the evolution of component-based software systems developed by a reuse-based process.

Other recent approaches show a particular interest to specifying architecture evolution as first-class entities. A first example is the work of Tamazalit, Le Goaer et al. [35,36]. The authors introduce the notion of *evolution styles*, first-class entities that can be specified and classified for reuse to evolve a particular family of systems. Evolution styles include evolution operations that can be specialized, composed and instantiated to deal with change. Barnes et al. [37] adopt a wider definition of evolution styles and introduce the concept of evolution paths as a way to plan the evolution of domain-specific software systems. A path is an evolution trace leading from an initial architecture to a desired target architecture. An evolution style refers to a family of evolution paths sharing common properties. It includes operations, constraints and functions to evaluate paths according to quality metrics. Path constraints can be formally specified using the *path constraint language*, a specific extension of LTL (Linear Temporal Logic). While the computability of the language was proved, as far as we know, there is no existing model checker to support the automated analysis of path constraints. The authors also propose a solution [38] to automate evolution planning using PDDL [39] (the Planning Domain Definition Language). However, this approach still lacks automation since no translation from any Adl to PDDL specification was proposed. Moreover, the evolution is specified and planned beforehand. In our approach, changes are not necessarily expected and the architect intervenes only to validate the work of the evolution manager.

Another closely related work is the one of Hansen, Ingstrup and others [40,41]. The authors propose an approach to model and analyze runtime architectural change. They opt for a runtime architecture model that closely maps to the OSGi[4] platform to facilitate implementation and for Alloy [42] as a relational first-order logic modeling language to formalize the static and dynamic (operations) concepts of the architecture model. The choice of Alloy is motivated by its support for object-oriented modeling and its accompanying analyzer that enables automated verification. The objective is to apply architectural changes without violating some predefined properties. For this purpose, the authors model the reconfiguration planning as a predicate satisfaction problem with pre- and post-conditions. Then, they run the Alloy SAT solver to find sequences of the model instances satisfying the problem where the first instance satisfies the pre-conditions and the last

[4] http://www.osgi.org/Main/HomePage.

instance satisfies the post-conditions. This work is similar to ours in the sense that both aim to provide a reliable and automated way to handle architectural changes. It proposes an interesting alternative for resolving evolution using the constraint-solving technique. However, this work focuses only on one level of change which is runtime. Moreover, the formalized architecture model is dependent on OSGi. Finally, the work lacks automation, since no automatic translation from ADL models to Alloy models was proposed.

5.2. Comparison of formal modeling languages

Formal modeling brings abstraction, precision and rigor to software systems. It intervenes at the very early stages of software development to give a formal specification of system requirements. Resulting models constitute unambiguous descriptions that enable software analysis, verification and validation. Several languages and methods were proposed to aid formal modeling. Formal languages provide abstractions to represent concepts, properties over them and possibly behavior. However, they differ in expressiveness, underlying semantics and purpose. Some languages focus more on descriptions and how to make formal modeling more accessible whereas others focus more on automated analysis neglecting expressiveness. A good formal language must be a compromise between both aspects. In the following, we compare seven formal modeling languages. These languages are B [9], Z [43], OCL [44], Alloy [42], VDM [45], Coq [46] and Agda [47].

B, Z and VDM are quite similar in term of expressiveness since they were basically designed for theorem-proving. All of them enable to express properties practically in the same way and support almost the same types (In addition, VDM supports real numbers). However there are some subtle differences between them. Z is more abstract while VDM and B are more low level and intended to be refined into code. Both VDM and B adopt a similar structure that realizes abstract state machines. They explicitly separate the declarative (structure) from the dynamic (operations) part and, unlike Z, they separate pre-conditions from post-conditions. B has the particularity to modify variables by assignments like in programming languages while in VDM and Z, pre and post states must be explicit.

Coq and Agda are proof assistants designed for the verification of functional programs. Unlike the previously mentioned formal modeling languages, Coq and Agda are implementations of type theories rather than set theory. They support higher order logic, polymorphism, dependent types, as well as inductive types. Set theoretic operators (*e.g.* $\cup$, $\cap$), for instance, are not directly predefined in such systems. Unlike B and VDM, these languages do not implement state machines. Therefore, there is no built-in structure that explicitly defines variables, invariants and operations.

OCL and Alloy are different and were designed for different purposes. OCL was basically developed to express constraints that cannot be expressed using graphical notations on UML diagrams. It has an object-oriented notation and heavily relies on navigation. Hence predicate expressions are sometimes verbose comparing to the mathematical notation adopted by the other languages. Alloy is a structural modeling language inspired by Z. It was designed for supporting fully automated analysis. Being strictly first-order, Alloy is less expressive than the other languages [48]. For instance, set of sets and predicates over relations are not directly expressible with Alloy.

Regarding analysis support, all these languages are typed and hence support type-checking. Theorem-proving is supported by Coq, Agda, Z, B and VDM which were basically designed for software correctness. Model-checking and constraint solving is only supported by B, with the ProB tool, and Alloy, with the Alloy analyzer. To some extent, Jaza [49], an animator for Z, enables constraint-solving on small domains. However, Z is limited in terms of model-checking capabilities. This is due to the high abstract nature of the Z language making its handling challenging [50]. Nevertheless, continuous attempts to build a model checker for Z are undertaken [51].

B seems to be the best compromise between expressiveness and analysis support. Alloy could also be a good alternative in our case. However, regardless its expressiveness, it presents another shortcoming. As witnessed in Torlak et al. [52], Alloy lacks support of partial instances. Partial instances are explicit representations of instances included in the specification of the model. This is central in our approach since instances are generated automatically from graphical models and injected in B specifications (so-called deep embedding technique [53]). Montaghami et al. [54] argued that this feature enables a number of capabilities such as test-driven development, regression testing, modeling by example, and combined modeling and meta-modeling. The authors also proposed a syntax extension of Alloy to support partial instance definition but, as far as we know, this feature is not yet integrated in the last version of Alloy [55].

5.3. Alternative formal approaches

Integration between semi-formal and formal methods is gaining more and more interest in software engineering. On the one hand, semi-formal languages, such as UML [56], offer graphical notations that significantly ease modeling. On the other hand, formal modeling languages provide a strong support for automated software analysis. Several works benefit from combining both kinds of notation to validate their approaches.

Ledru et al. [57] propose an approach based on the transformation of UML into B to validate security policies for information systems. They use their B4MSecure[5] tool to generate B specifications corresponding to a security model. Conjointly, they use ProB to validate security policy scenarios.

[5] http://b4msecure.forge.imag.fr/.

Keznikl et al. propose the ARCAS method [58], an automated approach to generate connections solutions for middleware architectures. Given a connector specification, the approach translates it into a corresponding Alloy model and performs constraint-solving to find connector instances that realize the specification.

Macedo et al. propose Echo [59], an Eclipse-based tool for model repair and transformation using model finding. Given a set of meta-models with internal constraints (specified using OCL) and a set of inter-model consistency rules (specified using QVT-R [60] transformations), Echo can detect inconsistencies on derived models and keep them consistent with their corresponding meta-models and between them as well. The detection and repair mechanism is based on translating MDE [61] artifacts (meta-models and their annotations with OCL and QVT-R rules) to Alloy. The output is then analyzed using a procedure built on top of Alloy solver that generates consistent models as close as possible to the original ones.

6. Conclusions and future work

Managing software architecture evolution throughout the whole software lifecycle is a significant issue. This paper proposes an approach to manage the evolution of component-based software architectures. Thanks to the three-level Dedal architecture model, our approach handles change at three abstraction levels of software architectures: specification, implementation and deployment. The evolution process is driven by an evolution management model that captures changes initiated at any abstraction level, controls their impact to preserve / restore consistency and propagates them to other levels to maintain global coherence.

The proposed evolution management model is based on the B formal language. Using our solver built on top of the ProB tool, it enables the generation of reliable evolution plans as sequences of change operations. The feasibility of our approach is demonstrated by experimenting on three evolution scenarios that each addresses change in a different abstraction level.

The limitation of this work is its scalability. This limitation is classical in comparable works as architecture descriptions can be considered as graphs (of connected software components) the size of which can theoretically be arbitrarily big. Establishing evolution plans therefore amounts to exploring all possible change action combinations on these graphs to restore properties that can be seen as (local or global) constraints on these graphs. Scalability issue is an inherent limitation for search-based software engineering problems. However, such limitation is mitigated by two factors. First, architecture descriptions are often limited in size as architects prefer to split them in intelligible parts of moderate size using hierarchical composition, an asset of CBSD [1]. Secondly, instead of using an off-the-shelf agnostic B solver, we proposed our own solver that integrates problem-specific heuristics that decrease the calculation time.

Threats to the validity of our approach lie in the example scenarios that we have considered for experimental validation. Although, the examples cover all kinds of scenarios, experimenting with real architecture descriptions might reveal unforeseen issues (scalability, efficiency of heuristics, *etc.*). Further experiments on real case studies is therefore necessary to fully validate our approach.

As future work, we would like to extend our definition of the consistency property in order to include behavioral consistency as described in Taylor et al. [21] and thus cover all their identified five kinds of consistency. This would amount in considering architectural protocols and component behavior.

Another interesting research direction would be to integrate the notion of evolution style [36] in our evolution management model. The idea is to enable the generation of multiple candidate evolution plans that can be evaluated considering non-functional properties (*e.g.* quality, cost, time) as proposed by Barnes et al. [37].

Regarding the technical aspect, we are investigating new heuristics to improve the performance of our solver and reduce complexity.

Acknowledgements

The authors warmly thank the anonymous reviewers for their in-depth reading of the paper and their helpful comments that made it possible to greatly improve its quality.

References

[1] H.V. Vliet, Software Engineering: Principles and Practice, 3rd edition, Wiley Publishing, 2008.

[2] H.P. Breivold, I. Crnkovic, M. Larsson, A systematic review of software architecture evolution research, Inf. Softw. Technol. 54 (1) (2012) 16–40, http://dx.doi.org/10.1016/j.infsof.2011.06.002, http://www.sciencedirect.com/science/article/pii/S0950584911001376.

[3] T. Mens, S. Demeyer, Software Evolution, Springer, 2008.

[4] D.E. Perry, A.L. Wolf, Foundations for the study of software architecture, SIGSOFT Softw. Eng. Notes 17 (4) (1992) 40–52.

[5] L. de Silva, D. Balasubramaniam, Controlling software architecture erosion: a survey, J. Syst. Softw. 85 (1) (2012) 132–151.

[6] H.Y. Zhang, C. Urtado, S. Vauttier, Architecture-centric component-based development needs a three-level ADL, in: Proceedings of the 4th European Conference of Software Architecture, Copenhagen, Denmark, in: Lecture Notes in Computer Science, vol. 6285, Springer, 2010, pp. 295–310.

[7] H.Y. Zhang, L. Zhang, C. Urtado, S. Vauttier, M. Huchard, A three-level component model in component-based software development, in: Proceedings of the 11th International Conference on Generative Programming: Concepts and Experiences, Dresden, Germany, ACM, 2012, pp. 70–79.

[8] I. Crnkovič, S. Sentilles, A. Vulgarakis, M. Chaudron, A classification framework for software component models, IEEE Trans. Softw. Eng. 37 (5) (2011) 593–615.

[9] J.-R. Abrial, The B-book: Assigning Programs to Meanings, Cambridge University Press, New York, USA, 1996.

[10] D. Cansell, D. Méry, Foundations of the B method, Comput. Inform. 22 (2003) 1–31.

[11] P. Behm, P. Benoit, A. Faivre, J.-M. Meynadier, Meteor: a successful application of B in a large project, in: J. Wing, J. Woodcock, J. Davies (Eds.), FM'99 — Formal Methods, in: Lecture Notes in Computer Science, vol. 1708, Springer, Berlin, Heidelberg, 1999, pp. 369–387.
[12] D. Cansell, G. Gopalakrishnan, M. Jones, D. Méry, A. Weinzoepflen, Incremental proof of the producer/consumer property for the PCI protocol, in: D. Bert, J. Bowen, M. Henson, K. Robinson (Eds.), ZB 2002: Formal Specification and Development in Z and B, in: Lecture Notes in Computer Science, vol. 2272, Springer, Berlin, Heidelberg, 2002, pp. 22–41.
[13] Atelier B, ClearSy, Aix-en-Provence (F), http://www.atelierb.eu/, accessed 09/01/2015.
[14] The B-Toolkit User's Manual, B-Core (UK) Limited.
[15] D. Delahaye, C. Dubois, C. Marché, D. Mentré, The BWare project: building a proof platform for the automated verification of B proof obligations, in: Y. Ait Ameur, K.-D. Schewe (Eds.), Abstract State Machines, Alloy, B, TLA, VDM, and Z, in: Lecture Notes in Computer Science, vol. 8477, Springer, Berlin, Heidelberg, 2014, pp. 290–293.
[16] M. Leuschel, M. Butler, ProB: an automated analysis toolset for the B method, Int. J. Softw. Tools Technol. Transf. 10 (2) (2008) 185–203.
[17] D. Garlan, R. Allen, J. Ockerbloom, Architectural mismatch: why reuse is so hard, IEEE Softw. 12 (6) (1995) 17–26, http://dx.doi.org/10.1109/52.469757.
[18] D. Garlan, R. Allen, J. Ockerbloom, Architectural mismatch: why reuse is still so hard, IEEE Softw. 26 (4) (2009) 66–69, http://dx.doi.org/10.1109/MS.2009.86.
[19] A. Mokni, M. Huchard, C. Urtado, S. Vauttier, H.Y. Zhang, Formal rules for reliable component-based architecture evolution, in: Formal Aspects of Component Software – 11th International FACS Symposium Revised Selected Papers, Bertinoro, Italy, 2014, pp. 127–142.
[20] A. Mokni, M. Huchard, C. Urtado, S. Vauttier, H.Y. Zhang, An evolution management model for multi-level component-based software architectures, in: The 27th International Conference on Software Engineering and Knowledge Engineering, SEKE 2015, July 6–8, 2015, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, 2015, pp. 674–679.
[21] R. Taylor, N. Medvidovic, E. Dashofy, Software Architecture: Foundations, Theory, and Practice, Wiley, 2009.
[22] A. Mokni, M. Huchard, C. Urtado, S. Vauttier, H.Y. Zhang, Towards automating the coherence verification of multi-level architecture descriptions, in: Proceedings of the 9th ICSEA, Nice, France, 2014, pp. 416–421.
[23] M. Leuschel, J. Bendisposto, Directed model checking for B: an evaluation and new techniques, in: J. Davies, L. Silva, A. Simao (Eds.), Formal Methods: Foundations and Applications, in: Lecture Notes in Computer Science, vol. 6527, Springer, Berlin, Heidelberg, 2011, pp. 1–16.
[24] N. Medvidovic, D.S. Rosenblum, R.N. Taylor, A language and environment for architecture-based software development and evolution, in: Proceedings of the 21st ICSE, 1999, pp. 44–53.
[25] J. Magee, J. Kramer, Dynamic structure in software architectures, ACM SIGSOFT Softw. Eng. Notes 21 (6) (1996) 3–14.
[26] F. Oquendo, B. Warboys, R. Morrison, R. Dindeleux, F. Gallo, H. Garavel, C. Occhipinti, ArchWare: architecting evolvable software, in: F. Oquendo, B. Warboys, R. Morrison (Eds.), Software Architecture, in: Lecture Notes in Computer Science, vol. 3047, Springer, Berlin, Heidelberg, 2004, pp. 257–271.
[27] A. Joolia, T. Batista, G. Coulson, A.T.A. Gomes, Mapping ADL specifications to an efficient and reconfigurable runtime component platform, in: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, IEEE, Washington, USA, 2005, pp. 131–140.
[28] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead Jr., J.E. Robbins, A component- and message-based architectural style for GUI software, in: Proceedings of the 17th International Conference on Software Engineering, ICSE '95, ACM, New York, USA, 1995, pp. 295–304.
[29] F. Oquendo, Pi-ADL: an architecture description language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures, SIGSOFT Softw. Eng. Notes 29 (3) (2004) 1–14.
[30] F. Oquendo, Pi-ARL: an architecture refinement language for formally modelling the stepwise refinement of software architectures, SIGSOFT Softw. Eng. Notes 29 (5) (2004) 1–20, http://dx.doi.org/10.1145/1022494.1022517.
[31] D. Garlan, R. Monroe, D. Wile, ACME: an architecture description interchange language, in: Proceedings of Centre for Advanced Studies Conference, IBM Press, 1997, p. 7.
[32] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, J. Ueyama, A component model for building systems software, in: Software Engineering and Applications, SEA '04, Cambridge, 2004, pp. 684–689.
[33] A. Sanchez, N. Oliveira, L.S. Barbosa, P. Henriques, A perspective on architectural re-engineering, Sci. Comput. Program. 98 (4) (2015) 764–784, http://dx.doi.org/10.1016/j.scico.2014.02.026, http://www.sciencedirect.com/science/article/pii/S0167642314000938.
[34] A. Sanchez, L. Barbosa, D. Riesco, Bigraphical modelling of architectural patterns, in: F. Arbab, P. Ölveczky (Eds.), Formal Aspects of Component Software, in: Lecture Notes in Computer Science, vol. 7253, Springer, Berlin, Heidelberg, 2012, pp. 313–330.
[35] D. Tamzalit, M. Oussalah, O.L. Goaer, A. Seriai, Updating software architectures: a style-based approach, in: Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, vol. 1, SERP 2006, Las Vegas, Nevada, USA, June 26–29, 2006, 2006, pp. 336–342.
[36] O.L. Goaer, D. Tamzalit, M. Oussalah, A. Seriai, Evolution shelf: reusing evolution expertise within component-based software architectures, in: Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2008, 28 July – 1 August 2008, Turku, Finland, 2008, pp. 311–318.
[37] J.M. Barnes, D. Garlan, B. Schmerl, Evolution styles: foundations and models for software architecture evolution, Softw. Syst. Model. 13 (2) (2014) 649–678.
[38] J.M. Barnes, A. Pandey, D. Garlan, Automated planning for software architecture evolution, in: Proceedings of 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11–15, 2013, 2013, pp. 213–223.
[39] D. McDermott, PDDL—the planning domain definition language, Technical report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.
[40] M. Ingstrup, K.M. Hansen, Modeling architectural change: architectural scripting and its applications to reconfiguration, in: Joint Working IEEE/IFIP Conference on Software Architecture, WICSA/ECSA, 2009, pp. 337–340.
[41] K.M. Hansen, M. Ingstrup, Modeling and analyzing architectural change with alloy, in: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10, ACM, New York, NY, USA, 2010, pp. 2257–2264.
[42] D. Jackson, Alloy: a lightweight object modelling notation, ACM Trans. Softw. Eng. Methodol. 11 (2) (2002) 256–290.
[43] J.M. Spivey, The Z Notation: A Reference Manual, Prentice Hall International (UK) Limited, 1992.
[44] OCL, 2.3.1 specification, http://www.omg.org/spec/OCL/2.3.1/, accessed 09/01/2015.
[45] C.B. Jones, Systematic Software Development Using VDM, 2nd ed., Prentice-Hall, 1990.
[46] Y. Bertot, P. Castéran, G. Huet, C. Paulin-Mohring, Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, Texts in Theoretical Computer Science, Springer, Berlin, New York, 2004, données complémentaires http://coq.inria.fr, http://opac.inria.fr/record=b1101046.
[47] U. Norell, Dependently typed programming in Agda, in: Proceedings of the 6th International Conference on Advanced Functional Programming, AFP'08, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 230–266, http://dl.acm.org/citation.cfm?id=1813347.1813352.
[48] D. Jackson, Software Abstractions: Logic, Language, and Analysis, Appendix E: Alternative approaches, The MIT Press, 2006.
[49] M. Utting, Jaza user manual and tutorial, http://www.cs.waikato.ac.nz/~marku/jaza/, accessed 03/08/2015.
[50] G. Smith, L. Wildman, Model checking Z specifications using SAL, in: H. Treharne, S. King, M. Henson, S. Schneider (Eds.), ZB 2005: Formal Specification and Development in Z and B, in: Lecture Notes in Computer Science, vol. 3455, Springer, Berlin, Heidelberg, 2005, pp. 85–103.

[51] J. Derrick, S. North, A. Simons, Z2SAL: a translation-based model checker for Z, Form. Asp. Comput. 23 (1) (2011) 43–71, http://dx.doi.org/10.1007/s00165-009-0126-7.
[52] E. Torlak, D. Jackson, Kodkod: a relational model finder, in: O. Grumberg, M. Huth (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, in: Lecture Notes in Computer Science, vol. 4424, Springer, Berlin, Heidelberg, 2007, pp. 632–647.
[53] J. Svenningsson, E. Axelsson, Combining deep and shallow embedding for EDSL, in: H.-W. Loidl, R. Peña (Eds.), Trends in Functional Programming, in: Lecture Notes in Computer Science, vol. 7829, Springer, Berlin, Heidelberg, 2013, pp. 21–36.
[54] V. Montaghami, D. Rayside, Extending alloy with partial instances, in: J. Derrick, J. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, E. Riccobene (Eds.), Abstract State Machines, Alloy, B, VDM, and Z, in: Lecture Notes in Computer Science, vol. 7316, Springer, Berlin, Heidelberg, 2012, pp. 122–135.
[55] Alloy language reference, http://alloy.mit.edu/alloy/documentation/book-chapters/alloy-language-reference.pdf, accessed 25/07/2015.
[56] UML, 2.5 specification, http://www.omg.org/spec/UML/2.5/Beta2/, accessed 09/01/2015.
[57] Y. Ledru, A. Idani, J. Milhau, N. Qamar, R. Laleau, J.-L. Richier, M.-A. Labiadh, Taking into account functional models in the validation of IS security policies, in: C. Salinesi, O. Pastor (Eds.), Advanced Information Systems Engineering Workshops, in: Lecture Notes in Business Information Processing, vol. 83, Springer, Berlin, Heidelberg, 2011, pp. 592–606.
[58] J. Keznikl, T. Bureš, F. Plášil, P. Hnětynka, Automated resolution of connector architectures using constraint solving (ARCAS method), Softw. Syst. Model. 13 (2) (2014) 843–872, http://dx.doi.org/10.1007/s10270-012-0274-8.
[59] N. Macedo, T. Guimaraes, A. Cunha, Model repair and transformation with Echo, in: IEEE/ACM 28th International Conference on Automated Software Engineering, ASE, 2013, pp. 694–697.
[60] OMG, MOF 2.0 Query/View/Transformation (QVT), version 1.1, http://www.omg.org/spec/QVT/1.1/, accessed 24/07/2015.
[61] D.C. Schmidt, Guest editor's introduction: model-driven engineering, Computer 39 (2) (2006) 25–31.