Towards Automating the Coherence Verification of Multi-Level Architecture Descriptions

Abderrahman Mokni*, Marianne Huchard†, Christelle Urtado*, Sylvain Vauttier* and Huaxi (Yulin) Zhang‡
*LGI2P, Ecole des Mines Alès, Nîmes - France
Email: {Abderrahman.Mokni, Christelle.Urtado, Sylvain.Vauttier}@mines-ales.fr
†LIRMM, CNRS and Université de Montpellier 2, Montpellier - France
Email: huchard@lirmm.fr
‡INRIA, Ecole Normale Supérieure de Lyon, Lyon - France
Email: yulinz88@gmail.com

Abstract—**Component-Based Software Engineering considers off-the-shelf software component reuse as its cornerstone. In previous work, we proposed Dedal, a three level Architecture Description Language. It supports a novel modeling approach that aims at describing the specification, the implemented configuration and the running assembly of the software. This eases reuse by guiding the search for existing components. In this paper, we propose a formal approach that states the rules for component reuse and interoperability among Dedal models. The use of B, a specification language providing model-checking capabilities, enables the automatic verification of Dedal architecture descriptions. The approach is illustrated using the example of a home automation software.**

Keywords–Software architecture, component reuse, B formal models, component subtyping, component compatibility, architecture levels.

I. INTRODUCTION

Component-Based Software Engineering (CBSE) aims at engineering software from previously developed components. Expected outcomes are to increase development speed and software quality, to ease the development of software of ever increasing complexity and to decrease costs. In previous work [1], we proposed a three step approach to specify, design and deploy software architectures from existing software components. This proposal also includes means to control architecture evolution. It is supported by a three level Architecture Description Language (ADL) and component model called Dedal. The originality of this approach is to focus on component reuse by guiding the search for adequate components during the CBSE process. In this paper, we propose rules to automatically support verification and validation of Dedal's architecture descriptions which is a first step to handle reuse and architecture-centric evolution in a rigorous way. The rules are expressed in the B [2] notation, a formalism that can be automatically verified using existing provers and model checking tools. The remaining of the paper is structured as follows. Section II gives an overview of the three Dedal models and illustrates them with a home automation architecture example. Section III presents an overview of our formalization of Dedal models using the B notation. Section IV sets the intra-level rules for component substitutability and compatibility. Section V describes inter-level rules that define the relations between component descriptions in two successive description levels. Section VI depicts an overview of the experimentation of the presented formal models and rules. Section VII analyzes related work before Section VIII concludes and discusses future work.

II. OVERVIEW OF THE DEDAL MODEL

Dedal is an ADL that helps software development at three abstraction levels. These levels have been designed to support reuse-centered architecture development. In the following, we detail each of Dedal's three abstraction levels [1]. To illustrate these concepts, we propose to model a part of Home Automation Software (HAS) that manages comfort scenarios. Here, it automatically controls the building's lighting and heating in function of the time and ambient temperature. For this purpose, we propose an architecture with an orchestrator component that interacts with the appropriate devices to implement the scenario.

A. The abstract architecture specification

The *abstract architecture specification* is the first level of architecture software descriptions. It provides a generic view of the global structure of the software and states its expected functionalities according to functional requirements. An architecture specification may correspond to a prescriptive architecture, which describes the system's architecture "as-wished" at specification time, as defined by Taylor *et al.* [3]. In Dedal, the architecture specification is composed of component roles and their connections. *Component roles* represent the roles that components are expected to play in the system. They thus are abstract and partial component type specifications. They are identified by the architect in order to search for and select corresponding concrete components in the next step. Figure 1-a shows a possible architecture specification for the HAS. 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.

B. The concrete architecture configuration

The *concrete architecture configuration* is an implementation view of the software architecture. It results from the selection of existing component classes in component repositories. Thus, an architecture configuration lists the concrete

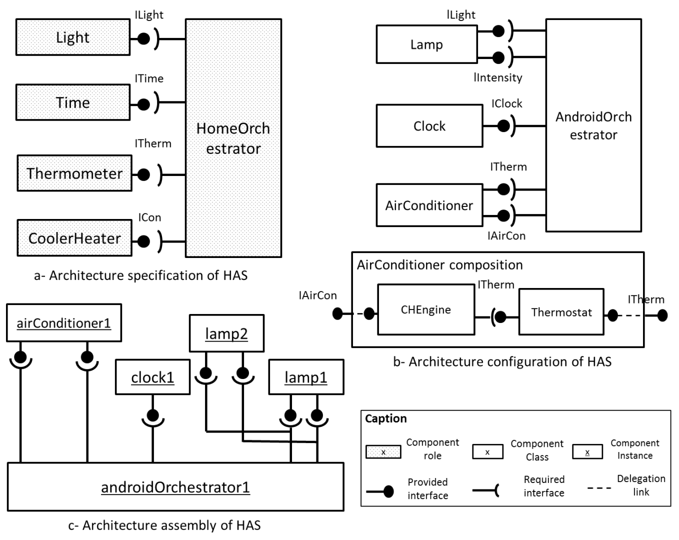

Figure 1: Architecture specification, configuration and assembly of HAS

component classes that compose a specific version of the software system.

Component classes are concrete component implementations. In Dedal, component classes can be either primitive or composite. *Primitive component classes* encapsulate executable code. *Composite component classes* encapsulate an inner architecture configuration (*i.e.,* a set of connected component classes which may, in turn, be primitive or composite). A composite component class exposes a set of interfaces corresponding to unconnected interfaces of its inner components. A *Component type* gives an abstract representation of a set of component classes. It defines the set of interfaces that a class must hold to be an implementation of this type. Component types are used to classify component classes and build indexes on the content of component repositories. To search for component classes that can be used to implement an architecture specification, component roles are matched with component types (using a classification based on specialization and substitutability in a manner similar to Arévalo *et al.* [4]). As they are implementations of their declared component types, these component classes are valid realizations of the component roles. Figure 1-b shows the architecture configuration of the HAS example as well as an example of an *AirConditioner* composite component and its inner configuration. As illustrated in this example, a single component class may realize several roles in the architecture specification as with the *AirConditioner* component class which realizes both *Thermometer* and *CoolerHeater* roles. Moreover, a component class may provide more services than those listed in the architecture specification as with the *Lamp* component class that provides an extra service to control the intensity of light.

C. The instantiated architecture assembly

The *instantiated architecture assembly* describes software at runtime and gathers information about its internal state. The architecture assembly results from the instantiation of an architecture configuration. It lists the instances of the component and connector classes that compose the deployed architecture at runtime and their assembly constraints (such as maximum numbers of allowed instances).

Component instances document how component classes from an architecture configuration are instantiated in the software. Each component instance has an initial and current state represented by a list of valued attributes. Figure 1-c shows an instantiated architecture assembly for the HAS example.

D. Motivation

The three-level Dedal model is a novel approach to component-based software development that increases reuse by supporting the search for off-the-shelf components. The associated ADL focuses on the description of architectural concepts in three separated abstraction levels but it lacks mechanisms to verify and validate architecture definitions before and after evolution. This work aims to provide mechanisms to automate the verification and validation of coherence between architecture levels from requirement to runtime. We propose a set of rules to define the relations inside each abstraction level and between two of them. The rules are expressed using B [2], a first-order logic and set-theoretic language with a rich expressiveness that can be automatically verified using existing model checkers.

III. OVERVIEW OF THE FORMALIZATION

The formalization is divided into two parts. A first part, which is generic and independent from any architectural model, consists in formalizing the most common concepts of software architectures. The second part is specific to Dedal and consists in formalizing concepts and relationships of the Dedal model. The formal model of Dedal therefore is a specialization of the generic model. In the remainder, we present the most important parts of the formalization.

A. Formalizing underlying architectural constructs

During the last decades, a consensus established that architectures were made of three main elements [5]: components (loci of computation), connectors (mediators) and configurations (topologies of the architecture). In Table I, we give the formal definition and relations between these concepts (the *arch_concepts* model).

We note that *Arch_concepts* includes an inner model called *Basic_concepts* which contains the formalization of finer grained elements (*i.e.,* interfaces and signatures). *Basic_concepts* is not presented in this paper for the sake of space.

B. Formalization of Dedal architecture levels

Dedal proposes three abstraction levels to describe architectures. Formalizing each of these levels enables to verify each of them separately but also to check the global coherence of architecture definitions.

The *Arch_specification* model. An architecture specification inherits from the generic definition of an architecture as stated in the *Arch_concepts* model. In Dedal, an architecture specification is specifically made of a set of component roles. Roles are thus defined as specializations of components by the following property: $COMP\_ROLES \subseteq COMPS \wedge compRole \subseteq COMP\_ROLES.$

The *Arch_configuration* model. In the same way, the component class concept used in the *Arch_configuration* model

TABLE I: Formal specification of underlying concepts

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
/* A component has a name and a set of interfaces */
$component \subseteq COMPS \wedge$
$comp\_name \in component \rightarrow COMP\_NAMES \wedge$
$comp\_interfaces \in component \rightarrowtail \mathcal{P}\ (interface) \wedge$
/* A client (resp. server) is a couple of a component and an interface */
$client \in component \leftrightarrow interface \wedge$
$server \in component \leftrightarrow interface \wedge$
/* A connection is a one-to-one mapping between a client and a server */
$connection \in client \twoheadrightarrowtail server \wedge$
/* An architecture has a set of components and connections */
$architecture \subseteq ARCHS \wedge$
$arch\_components \in architecture \rightarrow \mathcal{P}\ (component) \wedge$
$arch\_connections \in architecture \rightarrow \mathcal{P}\ (connection)$

Specific B notations:
$\rightarrow$: total function $\leftrightarrow$: relation $\rightarrowtail$: injection
$\twoheadrightarrowtail$: bijection $\mathcal{P}$(<set>): powerset of <set>

is defined as a specialization of the component concept, as they share the same properties (name, interface, etc.). Table II shows the formalization of the configuration level.

TABLE II: Formal specification of the configuration level

MACHINE$Arch\_configuration$
INCLUDES $Arch\_concepts, Arch\_specification$
SETS
COMP_CLASS; *CLASS_NAME*; *ATTRIBUTES*; *CONFIGURATIONS*
CONSTANTS
COMP_TYPES
PROPERTIES
/* Component types are also a specialization of components distinct from roles */
$COMP\_TYPES \subseteq COMPS \wedge COMP\_TYPES = COMPS - COMP\_ROLES$
VARIABLES
config, *config_components*, *config_connections*, *compType*, *compClass*,
class_name, *class_attributes*, *compositeComp*, *delegatedInterface*, ...
INVARIANT
$compType \subseteq COMP\_TYPES \wedge$
/* A component class has a name and a set of attributes */
$compClass \subseteq COMP\_CLASS \wedge class\_name \in compClass \rightarrow CLASS\_NAME \wedge$
$attribute \subseteq ATTRIBUTES \wedge class\_attributes \in compClass \rightarrow \mathcal{P}(attribute) \wedge$
/* A composite component has also a configuration and is constituted of component classes */
$compositeComp \subseteq compClass \wedge composite\_uses \in compositeComp \rightarrow config \wedge$
/* A delegation is a mapping between a delegated interface and its corresponding one */
$delegatedInterface \subset interface \wedge$
$delegation \in delegatedInterface \rightarrowtail interface \wedge$
/* A configuration is a set of component classes and connections*/
$config \subseteq CONFIGURATIONS \wedge$
$config\_components \in config \rightarrow \mathcal{P}(compClass) \wedge$
$config\_connections \in config \rightarrow \mathcal{P}(connection)$

The *Arch_assembly* model. The *Arch_assembly* model captures the definition of architectures at the instance level. Component instances are mapped to initial and current states. This information is useful to audit software evolution at runtime and control dynamic reconfigurations. Next section sets Dedal's intra-level rules by defining invariant constraints on the previously defined concepts.

IV. INTRA-LEVEL RULES

A. *Component substitutability rules*

In software architectures, substitutability determines when a component can replace another while holding the architecture consistent. The notion of substitutability was firstly discussed in object-oriented languages to define subtyping and object interoperability. This notion has also been discussed in the component-based paradigm [6] [7] but there is still no consensus in defining component substitutability. Indeed, components are complex entities that can be studied from many views (syntactic, semantic, protocol, etc.). In Dedal, at least a syntactic substitutability is needed to filter components while searching for suitable ones in repositories. The corresponding rules can be extended later to take dynamic behavior into account. Figure 2 shows an example of component subtyping that illustrates the main substitutability rules. The principle that is enforced is that a subtype should provide at least the same services as its supertype and require the same or less services. For example, *Clock* can be substituted for *ClockV2* which, provides one more interface (*IInfo*) and requires one less interface (interface *ILanguage* is no more required) (*cf.* Rule 1), and the interface type *ILocation* is subtyped by *ILocation&GMT* which has one more signature *getGMT()* (*cf.* Rule 3).

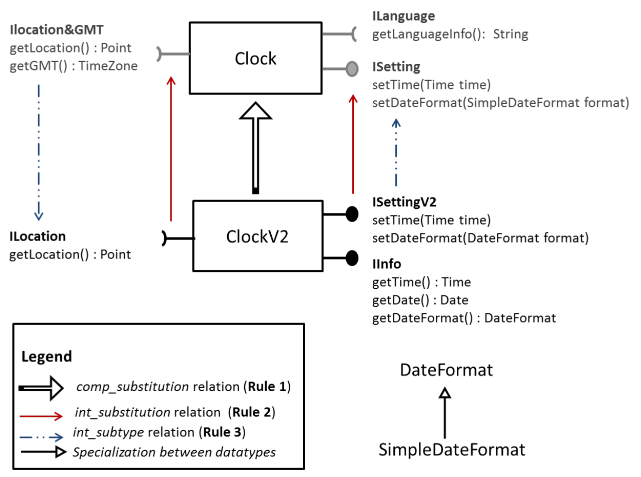
Ilocation&GMT
getLocation() : Point
getGMT() : TimeZone
Clock
ILanguage
getLanguageInfo(): String
ISetting
setTime(Time time)
setDateFormat(SimpleDateFormat format)
ILocation
getLocation() : Point
ClockV2
ISettingV2
setTime(Time time)
setDateFormat(DateFormat format)
IInfo
getTime() : Time
getDate() : Date
getDateFormat() : DateFormat
Legend
comp_substitution relation (Rule 1)
int_substitution relation (Rule 2)
int_subtype relation (Rule 3)
Specialization between datatypes
DateFormat
SimpleDateFormat

Figure 2: Example of component substitutability

Rule 1: Component substitutability. A component *C_sup* can be substituted for a component *C_sub* iff there exists an injection *inj1* between the set of interfaces of *C_sup* and the set of interfaces of *C_sub* such that *int* can be substituted for *inj1*(*int*), *int* being a provided interface of *C_sup*, and there exists an injection *inj2* between the set of interfaces of *C_sub* and the set of interfaces of *C_sup* such as *inj2*(*int*) can be substituted for *int*, *int* being a required interface of *C_sub*. Formally:

$comp\_substitution \in component \leftrightarrow component \wedge$
$\forall\ (C\_sup\ ,\ C\_sub).$
$(C\_sup \in component \wedge C\_sub \in component \wedge C\_sup \neq C\_sub$
$\Rightarrow$
$(C\_sub \in comp\_substitution\ [\{\ C\_sup\}]$
$\Leftrightarrow$
$\exists\ (inj1\ ,\ inj2).$
$(inj1 \in providedInterfaces\ (C\_sup) \rightarrowtail providedInterfaces(C\_sub) \wedge$
$\forall\ (int).$
$(int \in interface \wedge int \in providedInterfaces(C\_sup)$
$\Rightarrow$
$inj1(int) \in int\_substitution\ [\{int\}]) \wedge$

$inj2 \in requiredInterfaces\ (C\_sub) \rightarrowtail requiredInterfaces\ (C\_sup) \wedge$
$\forall\ (int).$
$(int \in interface \wedge int \in requiredInterfaces\ (C\_sub) \wedge$
$\Rightarrow$
$int \in int\_substitution\ [\{inj2\ (int)\}]))))$

According to Rule 1, the component subtype can have more provided and fewer required interfaces than its supertype. This rule depends on interface substitutability which we define as follows:

Rule 2: Interface substitutability. Interface substitutability depends on the interface type and direction. Interface substyping is given by Rule 3. When both interfaces are provided, substitutability is covariant with interface subtyping (*i.e.,* a provided interface *int_sup* is substituted for a provided interface *int_sub* iff the type of *int_sub* is a subtype of *int_sup*'s type). In the second case where the two interfaces are required, substitutability is contravariant with interface subtyping (*i.e.,* a required interface *int_sup* is substituted for a required interface *int_sub* iff the type of *int_sup* is a subtype of *int_sub*'s type).

Rule 3: Interface subtyping. An interface type *intTypeSub* is a subtype of an interface type *intTypeSup* iff there exists an injection *inj* between the signature set of *intTypeSup* and the signature set of *intTypeSub* such that for each signature *sig* of *intTypeSup*, *inj*(*sig*) specializes *sig*.

```
int_subtype ∈ interfaceType ↔ interfaceType ∧
∀ (intTypeSup,intTypeSub).
(intTypeSup ∈ interfaceType ∧ intTypeSub ∈ interfaceType ∧
intTypeSup ≠ intTypeSub
      ⇒
   ((intTypeSup, intTypeSub) ∈ int_subtype
       ⇔
    ∃ inj.
  (inj ∈ int_signatures(intTypeSup) ⤖ int_signatures(intTypeSub) ∧
      ∀ (sig).
      (sig ∈ signature ∧ sig ∈ int_signatures(intTypeSup)
         ⇒
      inj(sig) ∈ sig_subtype[{sig}]))
   )
)
```

According to Rule 3, interface subtyping allows to add new signatures. This is why this relation is used both in a covariant way on provided interfaces and in a contravariant way (to require less signatures) on required interfaces in Rule 2. Interface subtyping in turn relies on signature specialization.

Rule 4: Signature specialization. Signature specialization conforms to method overriding in the object-oriented paradigm. A specialized signature must have contravariant specialization of parameter types and covariant specialization of return type as it must require less information when invoked and provide richer results. To define signature specialization, we first consider parameter specialization.

Rule 4.1. A signature *sig_sub* is parameter subtype of a signature *sig_sup* iff there exists an injection *inj* between the parameters of *sig_sup* and the parameters of *sig_sub* and for each parameter *param* of *sig_sup*, *inj*(*param*) has the same name as *param* and the type of *inj*(*param*) is a subtype of *param*'s type.

```
∀ (sig_sup, sig_sub).
(sig_sup ∈ signature ∧ sig_sub ∈ signature ∧ sig_sup ≠ sig_sub
⇒ (
   (sig_sup, sig_sub) ∈ param_subtype
   ⇔
   ∃ inj.( inj ∈ parameters(sig_sup) ⤖ parameters(sig_sub) ∧
           ∀ param.(param ∈ parameter ∧
                        param ∈ parameters (sig_sup)
           ⇒
           param_name (param) = param_name (inj (param)) ∧
           parameter_type (inj( param ))
                     ∈ closure (subtype)[{parameter_type (param)}]))
           )
   )
```

Rule 4.2. A signature *sig_sub* specializes a signature *sig_sup* if and only if they have the same name and *sig_sup* is parameter subtype of *sig_sub* and the return type of *sig_sub* is a subtype of the return type of *sig_sup*.

```
∀ (sig_sup, sig_sub).
( sig_sup ∈ signature ∧ sig_sub ∈ signature ∧ sig_sup ≠ sig_sub
⇒ (
   (sig_sup, sig_sub) ∈ sig_subtype
   ⇔ (
      sig_name (sig_sup) = sig_name (sig_sub) ∧
      (sig_sub, sig_sup) ∈ param_subtype ∧
      sig_return (sig_sub) ∈ closure (subtype)[{sig_return (sig_sup)}])
      )
   )
```

B. Component compatibility rules

Components compatibility relies on interface compatibility. Two components can interact if and only if they have at least two compatible (connectable) interfaces.

Rule 5: Interface compatibility. A provided interface *intA* and a required interface *intB* are compatible iff the type of *intA* is a subtype of *intB*'s.

In other words, a provided interface should declare the same, a specialization of and possibly extra signatures than the required interface to ensure that all the required functionalities can be supplied.

Substitutability and compatibility rules are defined for general-purpose. In Dedal, they are used to check intra-level relations between components of the same kind (*i.e.,* roles, types, classes or instances). In the remainder, we focus on the inter-level rules which enable to check the global coherence between the multiple architecture definitions.

V. Inter-level rules

Specifying inter-level rules is a crucial step to ensure coherence between architecture levels from requirements to runtime (*cf.* Figure 3). In order to go from the specification of an architecture to an implemented configuration, the architect must select suitable concrete component classes that realize the specified roles. The implementation can then be instantiated and deployed in multiple contexts. Inter-level rules are the foundations to automate such a reuse process in component-based software development.

A. Relations between the specification and configuration levels

Two main relations are considered between the specification and configuration levels: the matching relation

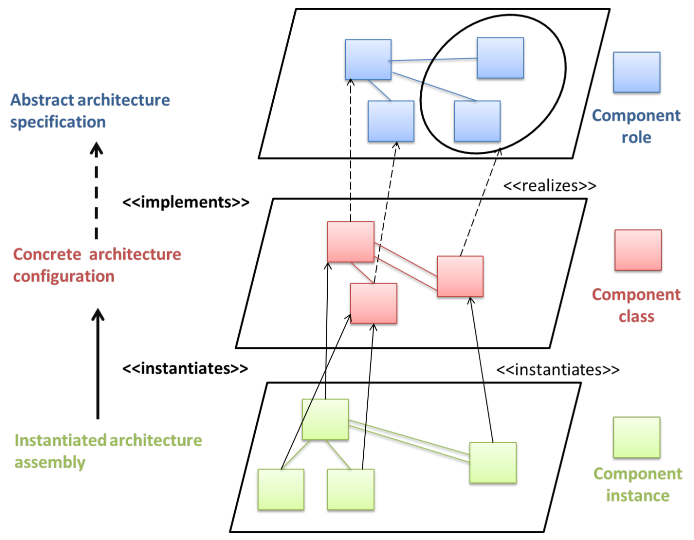

Figure 3: Relations between architecture levels

between component roles and concrete component types and the realization relation between component roles and component classes.

Rule 6: Component type matching. A component type *CT* matches with a component role *CR* iff it exists an injection *inj* between the set of interfaces of *CR* and the set of interfaces of *CT* such that *int* can be substituted for *inj*(*int*), *int* being an interface of *CR*. Formally:

$$
\begin{aligned}
&matches \in compType \leftrightarrow compRole \wedge \\
&\forall\,(CT, CR).(CT \in compType \wedge CR \in compRole \\
&\Rightarrow \\
&((CT,CR) \in matches \\
&\quad\Leftrightarrow \\
&\exists(inj).(inj \in comp\_interfaces\ (CR) \rightarrowtail comp\_interfaces\ (CT) \wedge \\
&\qquad \forall\ (int).(int \in interface \Rightarrow inj\ (int) \in int\_substitution\ [\{int\}]) \\
&)))
\end{aligned}
$$

As stated in Section II, component role descriptions are specified by the architect to guide the search for existing component classes. Hence, there are several ways to find a concrete realization of component roles. A component class can realize several roles at once or a composition of component classes (composite component) can complement each other to realize a given role. This holds a many-to-many mapping between component roles and concrete component types.

Rule 7: Component implementation. To draw an analogy with object-oriented programming, the relation between a component class and a component type is similar to the relation between a class and an interface. A component class must implement all the provided interfaces of the component type. Implementation details (that depend on decisions of the architect) are out of the scope of the abstract aspects that we intend to validate. However, an abstract formalization of the implementation is compulsory to make our formal model coherent. Component implementation is defined as follows:

$$class\_implements \in compClass \rightarrow compType$$

Rule 8: Component realization. The relation between a component class and a component role is a corollary of the matching relation (Rule 6) and the implementation relation (Rule 7). Indeed, a component class *CL* realizes a component role *CR* iff it exists a component type *CT* implemented by *CL* and that matches with *CR*. Formally:

$$
\begin{aligned}
&realizes \in compClass \leftrightarrow compRole \wedge \\
&\forall\ (CL, CR).(CL \in compClass \wedge CR \in compRole \\
&\Rightarrow \\
&\quad((CL, CR) \in realizes \\
&\qquad\Leftrightarrow \\
&\quad\exists\ CT.(CT \in compType \wedge (CT\ , CR) \in matches \wedge \\
&\quad(CL,CT) \in class\_implements)) \\
&)
\end{aligned}
$$

Rule 9: Relation between an architecture specification and its configuration. An architecture configuration *Conf* realizes an architecture specification *Spec* iff for each component role *CR* in *Spec* it exists a component class *CL* in *Conf* such that *CL* realizes *CR*.

$$
\begin{aligned}
&implements \in config \leftrightarrow arch\_spec \wedge \\
&\forall\ (Conf, Spec).(Conf \in config \wedge Spec \in arch\_spec \\
&\quad\Rightarrow \qquad (Conf, Spec) \in implements \\
&\qquad\qquad\Leftrightarrow \\
&\quad\forall\ CR.(CR \in compRole \wedge CR \in spec\_components\ (Spec) \Rightarrow \\
&\qquad\exists\ CL.(CL \in compClass \wedge CL \in config\_components\ (Conf) \wedge \\
&\qquad(CL, CR) \in realizes)))
\end{aligned}
$$

B. Relation between the configuration and assembly levels

An architecture assembly is composed of instances of the component classes that are in the architecture configuration. The instantiation depends on many technical choices that should be made by the architect (*e.g.*, the choice of the runtime framework) and should not be considered at such an abstract level of formalization.

$$comp\_instantiates \in compInstance \rightarrow compClass$$

The instantiation is a total function between the set of component instances and the set of component classes. This means that each component instance instantiates one and only one component class. Conversely, a component class can have several instances (the number of instances can be specified through assembly constraints).

Consequently, an architecture assembly *Asm* instantiates an architecture configuration *Conf* iff every component class *CL* of *Conf* is instantiated at least once by a component instance *CI* in *Asm* and every component instance *CI* in *Asm* is an instance of a component class in *Conf*:

$$
\begin{aligned}
&instantiates \in assm \rightarrow config \\
&\forall\ (Asm,Conf).(Asm \in assm \wedge Conf \in config \\
&\Rightarrow \\
&((Asm,Conf) \in instantiates \\
&\quad\Leftrightarrow \\
&\forall\ CL.(CL \in compClass \wedge CL \in config\_components(Conf) \\
&\quad\Rightarrow \\
&\quad\exists\ CI.(CI \in compInstance \wedge CI \in assm\_components(Asm) \wedge \\
&\quad(CI,CL) \in comp\_instantiates) \wedge \\
&\forall\ CI.(CI \in compInstance \wedge CI \in assm\_components(Asm) \\
&\quad\Rightarrow \\
&\quad\exists\ CL.(CL \in compClass \wedge CL \in config\_components(Conf) \wedge \\
&(CI,CL) \in comp\_instantiates))\ )))
\end{aligned}
$$

VI. Experimentation Overview

In order to validate the proposed rules, formal models are manually instantiated using simple tests covering the main cases. Each test corresponds to a specific instantiation of a given architectural model to check if one of the defined rules meets the required definition. Models are checked using ProB [8], a model checker of B that shows invariant violations and the current state of the given model. In case a violation is detected, either instantiation is wrong or the defined rules have to be revisited. At this stage of work, all rules have been manually validated and can be used later to automate the analysis process.

In future work, we aim to automatically generate the specification of model instances from the graphical or textual descriptions of architectures. The derived models will then be passed to the model checker for automatic verification of the architectural descriptions.

VII. Related work

Over two decades ago, many researches focused on giving ADL's a formal representation. A classification of the major ADL's was proposed by Medvidovic *et al.* [5]. Although, most of these ADL's provide the required features to describe an architecture, they often are either domain-specific or lack formal foundations to support automatic analysis and dynamic evolution. Some ADL's like Wright [9] and Rapide [10] focus on the specification and verification of architectural behavior. Wright uses CSP, a formal language based on process algebra while Rapide uses partially ordered sets (posets) of events to model behavior and enable formal reasoning on architecture specifications. Both Wright and Rapide, however, focus on architecture behavior and do not consider its structure. They do not provide any mechanism for component reuse and do not support multiple abstraction levels either.

Other close works are the formalization and analysis of architectural styles using a formal language. Kim and Garlan [11] propose an approach for modeling and analyzing architectural styles using Alloy. These works address architecture styles rather than architecture constructs and aim to provide a generic formal model for several styles like C2 [12] or the pipe and filter style. Our focus is slightly different since we address the structure of architectures independently from its style.

Our work has also drawn inspiration from type theory in object languages [13]. Like objects, components can have subtyping relations that enable reuse and software evolution. However, components are more complex than plain objects and they do not obey the same rules. To our knowledge, there was no real attempt, except for Medvidovic *et al.* to adapt the theory of objects to components. Medvidovic *et al.* propose a type theory for software architectures by multiple component subtyping and have the architect decide about which properties (name, interface, behavior or implementation) he wants to specialize. They applied their theory on their C2SADEL [6] ADL. In our three level component model, we need different typing rules to define relations between components into and between each levels of architecture descriptions. A part of our subtyping rules is also inspired from our previous work on building component directories using Formal Concept Analysis [14]. In fact, rules for specializing functionality signatures were defined to guide the search for compatible or substitutable components in a yellow-page like component directory.

VIII. Conclusion and future work

This paper proposes mechanisms to automate component reuse and inter-level coherence checking in a component-based development process. The outlined approach consists in coupling a three-level ADL called Dedal with B formal models. These models were reinforced with invariant constraints to set substitutability and compatibility rules into each abstraction level and inter-level rules to enable (1) reuse by guiding the search for concrete component classes and (2) coherence checking between abstraction levels. This work sets the basis for the definition of evolution rules which will be the next step of our contribution. Indeed, the proposed mechanisms will be used to automatically handle software evolution and propagate changes among architecture descriptions to preserve coherence.

A practical issue of future work will be to provide a toolset for Dedal, our three-level ADL. Indeed, we plan to map Dedal to UML and provide a visual modeling tool. Furthermore, we are considering the integration of existing model checkers and animation tools to automate verification and realize simulations and early validations of evolution scenarios.

References

[1] H. Y. Zhang, C. Urtado, and S. Vauttier, "Architecture-centric component-based development needs a three-level ADL," in Proc. of the 4th ECSA, ser. LNCS, vol. 6285. Copenhagen, Denmark: Springer, August 2010, pp. 295–310.

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

[3] R. Taylor, N. Medvidovic, and E. Dashofy, Software architecture: Foundations, Theory, and Practice. Wiley, 2009.

[4] G. Arévalo, N. Desnos, M. Huchard, C. Urtado, and S. Vauttier, "FCA-based service classification to dynamically build efficient software component directories," International Journal of General Systems, 2008, pp. 427–453.

[5] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," IEEE TSE, vol. 26, no. 1, Jan. 2000, pp. 70–93.

[6] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor, "A language and environment for architecture-based software development and evolution," in Proc. of the 21st ICSE, Los Angeles, USA, 1999, pp. 44–53.

[7] A. Beugnard, J.-M. Jezequel, N. Plouzeau, and D. Watkins, "Making components contract aware," Computer, vol. 32, no. 7, Jul 1999, pp. 38–45.

[8] M. Leuschel and M. Butler, "ProB: An automated analysis toolset for the B method," International Journal on Software Tools for Technology Transfer, vol. 10, no. 2, Feb. 2008, pp. 185–203.

[9] R. Allen and D. Garlan, "A formal basis for architectural connection," ACM TOSEM, vol. 6, no. 3, Jul. 1997, pp. 213–249.

[10] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and analysis of system architecture using rapide," IEEE TSE, vol. 21, 1995, pp. 336–355.

[11] J. S. Kim and D. Garlan, "Analyzing architectural styles," Journal of Systems and Software, vol. 83, no. 7, Jul. 2010, pp. 1216–1235.

[12] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., and J. E. Robbins, "A component- and message-based architectural style for GUI software," in Proc. of the 17th ICSE. Seattle, USA: ACM, 1995, pp. 295–304.

[13] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," ACM TOPLAS, vol. 16, no. 6, 1994, pp. 1811–1841.

[14] N. A. Aboud, G. Arévalo, J.-R. Falleri, M. Huchard, C. Tibermacine, C. Urtado, and S. Vauttier, "Automated architectural component classification using concept lattices," in Proc. WICSA/ECSA, Cambridge, UK, September 2009, pp. 21–31.