Modélisation et vérification formelles en B d'architectures logicielles à trois niveaux

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

▶ **To cite this version:**

Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, Huaxi Yulin Zhang. Modélisation et vérification formelles en B d'architectures logicielles à trois niveaux. CIEL: Conférence en IngénieriE du Logiciel, Jun 2014, Paris, France. pp.71-77. hal-01244431

HAL Id: hal-01244431
https://hal.science/hal-01244431

Submitted on 1 Jun 2021

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

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

Modélisation et vérification formelles en B d'architectures logicielles à trois niveaux

Abderrahman Mokni[1], Marianne Huchard[2], Christelle Urtado[1], Sylvain Vauttier[1] and Huaxi(Yulin) Zhang[3]

[1] EMA/LGI2P, Nîmes, France
{abderrahman.mokni, christelle.urtado, sylvain.vauttier}@mines-ales.fr
[2] LIRMM, CNRS et Université de Montpellier 2, Montpellier, France
marianne.huchard@lirmm.fr
[3] ENS/INRIA, Lyon, France
yulin88@gmail.com

Résumé

La réutilisation est une notion centrale dans le développement à base de composants. Elle permet de construire des logiciels à grande échelle de meilleure qualité et à moindre coût. Afin d'intensifier la réutilisation dans les processus de développement, un ADL à trois dimensions, nommé Dedal, a été proposé. Dedal permet de décrire la spécification, l'implémentation et le déploiement d'une architecture. Chaque définition doit être cohérente, complète et, réutilisant la définition de niveau supérieur, conforme à celle-ci. Cet article présente des règles formelles permettant de préserver et de vérifier ces trois propriétés dans des définitions d'architectures décrites en Dedal. Les règles sont exprimées avec le langage formel B afin d'automatiser leur vérification.

1 Introduction

L'ingénierie du logiciel a beaucoup évolué durant les deux dernières décennies. En effet, des nouveaux besoins tels que la construction et la maintenance de systèmes larges et complexes, la diminution des coûts et temps de développement et la sûreté des logiciels, sont apparus. Le développement à base de composants, discipline du génie logiciel [8], semble être la solution la plus adéquate pour répondre à tous ces besoins. Elle propose une démarche de construction de logiciels à large maille par assemblage de blocs de composants préexistants et suffisamment découplés pour être utilisés dans de multiples contextes. Le logiciel ainsi produit est décrit sous la forme d'une architecture logicielle comprenant les composants et les connexions devant les relier. Plusieurs langages connus sous le nom d'ADL (Architecture Description Language) ont été proposés afin de décrire les architectures logicielles. Certes, la plupart des ADLs, tels que C2 [9], Wright [2] et Darwin [5], permettent de modéliser les composants, les connecteurs et les configurations mais aucun ADL ne couvre toutes les étapes de cycle de vie d'une architecture.

Dans nos travaux précédents [10, 11], un ADL à trois niveaux, nommé Dedal, a été proposé. La particularité de Dedal est de représenter explicitement trois niveaux d'abstraction dans la définition d'une architecture : la spécification, la configuration et l'assemblage. Ces niveaux correspondent respectivement aux étapes de conception, implémentation et déploiement d'un système. L'objectif de Dedal est de favoriser la réutilisation dans les processus de développement à base de composants et de supporter une évolution contrôlée des architectures logicielles. Néanmoins, Dedal manque du formalisme nécessaire pour gérer ces processus d'une manière automatique et pour vérifier la consistance des définitions d'architectures dans chaque niveau et leur conformité par rapport au niveau supérieur afin d'éviter les problèmes d'érosion et de dérive [7].

L'objectif dans cet article est de palier à ce manque en proposant une formalisation des trois niveaux de Dedal ainsi que les règles de consistance et de conformité régissant ces définitions. La formalisation est exprimée en B [1], un langage de modélisation formelle fondé sur la théorie des ensembles et la logique des prédicats.

La suite de l'article est organisée de la manière suivante. La section 2 présente un aperçu du modèle Dedal. La section 3 définit une formalisation en B des trois niveaux d'architectures de Dedal. La

section 4 propose les règles de consistance et de conformité que doivent vérifier les définitions décrites en Dedal. La section 5 clôture le papier donnant une conclusion et des perspectives à ce travail.

2 Dedal, le modèle d'architectures à trois niveaux

Pour illustrer les concepts de Dedal, on propose un exemple d'application à la domotique. Il s'agit d'un système d'orchestration de scénarios de confort à domicile (Home Automation Software HAS). L'objectif est de pouvoir gérer la luminosité et la température du bâtiment en fonction du temps et de la température ambiante. Pour celà, on propose une architecture avec un composant orchestrateur qui interagit avec les équipements adéquats afin de réaliser le scénario de confort souhaité.

2.1 Le niveau spécification

La spécification est le premier niveau de description d'une architecture. Elle permet de définir une architecture idéale répondant aux exigences du cahier des charges. La spécification est constituée de composants rôles, de leurs connexions ainsi que du comportement global de l'architecture. Chaque composant rôle remplit des fonctionnalités requises dans le système. Défini comme un types de composant abstrait, sa description permet de guider la recherche de classes de composants concrets dans les bibliothèques afin de trouver une implémentation de l'architecture. La figure 1-a montre une spécification de l'architecture du HAS composée des rôles orchestrateur (*HomeOrchestrator*), lumière (*Light*), temps (*Time*), thermomètre (*Thermometer*) et climatiseur (*CoolerHeater*).

2.2 Le niveau configuration

La configuration est le deuxième niveau de description d'une architecture. Elle représente une implémentation de l'architecture par des classes de composants concrets et est dérivée de sa spécification en utilisant la description des rôles comme critère de recherche et de sélection dans une bibliothèque de composants [3]. Les types de composants sélectionnés doivent correspondre aux rôles identifiés dans le niveau spécification. En effet, chaque classe de composant possède un nom et des attributs et implémente un type de composant qui décrit ses interfaces ainsi que son comportement. En Dedal, les classes de composants peuvent être primitives ou composites. Ainsi, un rôle peut être réalisé par une composition de classes de composants et inversement, une classe de composant peut réaliser plusieurs rôles à la fois. La figure 1-b illustre l'architecture de HAS au niveau configuration ainsi qu'un exemple de composant composite (*AirConditioner*) qui réalise en même temps les rôles *Thermometer* et *CoolerHeater*. Les classes *AndroidOrchestrator*, *Lamp*, *Clock* réalisent respectivement les rôles *HomeOrchestrator*, *Light* et *Time*.

2.3 Le niveau assemblage

L'assemblage est le troisième niveau de description d'une architecture. Il correspond au déploiement de l'architecture et à son exécution. En Dedal, l'assemblage est décrit par des instances de composants et éventuellement des contraintes telles que le nombre maximum d'instances d'une classe de composant. Les instances de composants peuvent posséder des attributs valués qui représentent leurs états initial et courant. Ceci permet de décrire des paramétrages spécifiques des états de composants permettant différentes utilisations de l'architecture. La figure 1-c présente l'architecture du HAS au niveau assemblage. On note qu'une classe de composant peut avoir plusieurs instances comme montré dans l'exemple par les deux instances *lamp1* et *lamp2* de *Lamp*.

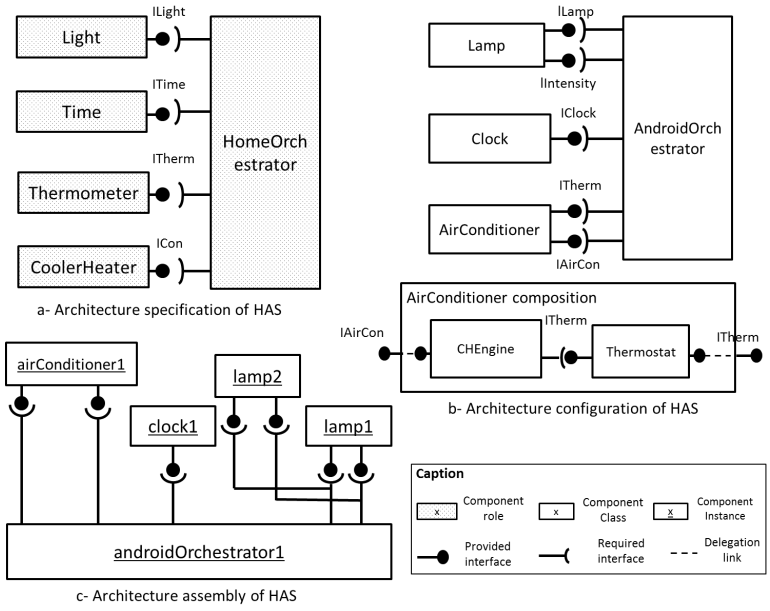

FIGURE 1 – Les trois niveaux d'architectures de Dedal

2.4 Les relations entre les trois niveaux d'architectures

La figure 2 illustre les relations entre les composants dans les trois niveaux d'architectures de Dedal. Une classe de composant réalise (*realizes*) un rôle défini dans la spécification et implémente (*implements*) un type de composant. La description de ce dernier doit correspondre à (*matches*) celle du rôle réalisé. Une classe de composant peut avoir plusieurs instances (*instantiates*) dans le niveau assemblage.

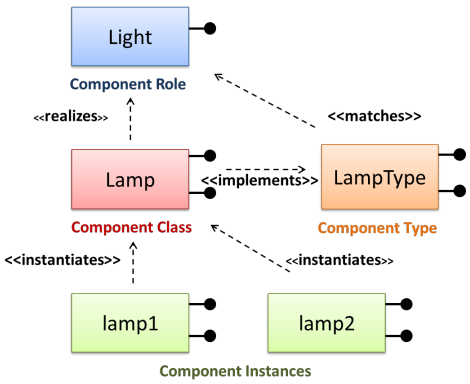

FIGURE 2 – Les relations entre les composants dans Dedal

Afin de pouvoir vérifier la consistance d'une architecture dans chaque niveau ainsi que la conformité d'un niveau d'abstraction par rapport au niveau supérieur, il est indispensable de définir formellement chaque niveau ainsi que les règles qui régissent ses relations avec le niveau supérieur. Cet objectif fait l'objet de notre proposition que nous présentons dans les sections suivantes.

3 Modélisation formelle en B de Dedal

Cette section présente la formalisation en B des concepts de Dedal. Le choix de B est motivé par le souhait d'étendre Dedal avec des définitions formelles exprimées dans un langage de prédicats et pouvant être vérifiées automatiquement moyennant des outils existants.
La formalisation comprend deux types de définitions. Des définitions génériques liées aux concepts communs à tous les ADLs à savoir les composants, les connexions et les architectures et des définitions spécifiques liées aux concepts de l'ADL Dedal. Le modèle formel générique peut servir de base pour d'autres ADL.
Vu la limite de l'espace, nous ne présentons que la partie de la formalisation que nous jugeons indispensable pour comprendre la contribution proposée (certaines déclarations d'ensembles et de variables sont donc omises dans cet article). La formalisation complète est présentée dans un travail précédent [6] qui propose les règles de substituabilité et compatibilité ainsi que les règles entre les composants de différents niveaux. Dans cet article, nous complétons les règles précédemment formulées par des règles de consistance et de conformité d'architectures.
Le modèle générique *Arch_concepts*. La table 1 présente la formalisation des concepts d'architecture, composant et connexion.

MACHINE *Arch_concepts* **INCLUDES** *Basic_concepts*
SETS *ARCHS ; ARCH_NAMES ; COMPS ; COMP_NAMES*
VARIABLES
architecture, arch_components, arch_connections, component, ...
INVARIANT
/* Un composant possède un nom et un ensemble d'interfaces */
$component \subseteq COMPS \wedge comp\_name \in component \rightarrow COMP\_NAMES \wedge$
$comp\_interfaces \in component \rightarrowtail \mathcal{P}\ (interface) \wedge$
/* Un client (repect. serveur) est un couple de composant et une interface */
$client \in component \leftrightarrow interface \wedge server \in component \leftrightarrow interface \wedge$
/* Une connexion est une bijection entre un client et un serveur */
$connection \in client \rightarrowtail\!\!\!\!\!\rightarrow server \wedge$
/* Une architecture est composée de composants et de connexions*/
$architecture \subseteq ARCHS \wedge arch\_components \in architecture \rightarrow \mathcal{P}_1(component) \wedge$
$arch\_connections \in architecture \rightarrow \mathcal{P}(connection)$

Notations spécifiques en B :
$\leftrightarrow$: relation $\rightarrowtail$: injection $\rightarrowtail\!\!\!\!\!\rightarrow$: bijection
$\mathcal{P}$(<set>) : ensemble des parties de <set> $\mathcal{P}_1$ (<set>) = $\mathcal{P}$ (<set>) - $\varnothing$

TABLE 1 – Spécification formelle des concepts génériques

Arch_concepts inclut un autre modèle nommé *Basic_concepts* qui contient la formalisation détaillée de la notion d'interface et ses éléments (type, direction et signature) ainsi que les règles de substituabilité et de compatibilité entre interfaces.
Formalisation des concepts de Dedal. Les niveaux spécification et configuration reprennent pratiquement les même définitions que celles qui sont proposées dans le modèle générique. En effet, les composants rôles (niveau spécification) et les types de composants (niveau configuration) partagent les mêmes caractéristiques (nom et liste d'interfaces) et héritent donc du concept de composant. Les classes de composants ont une description différente du fait qu'ils peuvent avoir des attributs supplémentaires et que leurs interfaces sont déduites du type de composant implémenté. Les connexions sont décrites de la même façon dans tous les niveaux (*i.e* : un client avec interface requise doit être connecté à un serveur avec une interface fournie compatible). La table 2 montre une partie de la formalisation des trois niveaux de Dedal chacun dans un modèle B dédié (*arch_specification*, *arch_configuration* et *arch_assembly*).
Les règles intra et inter-niveaux dans Dedal. Conformément aux concepts définis précédemment, il existe deux types de règles dans Dedal : des règles intra-niveau qui sont appliquées sur les concepts d'un même niveau d'abstraction et qui définissent notamment la substituabilité et la compatibilité entre les composants et des règles inter-niveau qui définissent les relations entre deux niveaux d'abtsraction différents (voir figure 2). Nous citons à titre d'exemple la règle de réalisation entre un rôle et une classe de composant qui sera utilisée par la suite pour établir la règle de conformité :

MACHINE *Arch_specification* **USES** *Arch_concepts*
...
PROPERTIES
/* Les composants rôles (resp. les spécifications) héritent du concept de composant (resp. architectures) */
$COMP_ROLES \subseteq COMPS \wedge ARCH_SPEC \subseteq ARCHS$

MACHINE*Arch_configuration*
...
INVARIANT
/* chaque composant classe possède un nom, une liste d'attributs et implémente un composant type*/
$compClass \subseteq COMP_CLASS \wedge class_name \in compClass \rightarrow CLASS_NAME \wedge$
$class_attributes \in compClass \rightarrow \mathcal{P}(attribute) \wedge$
$class_implements \in compClass \rightarrow compType \wedge$
/* Un composant composite est un composant classe et est décrit par une configuration
$compositeComp \subseteq compClass \wedge composite_uses \in compositeComp \rightarrow config \wedge$
/* Une délégation est une relation entre une interface déléguée et une interface interne
$delegation \in delegatedInterface \rightarrow interface \wedge$
/* Une configuration est composée d'au moins un composant classe et de connexions */
$config \subseteq \quad CONFIGURATIONS \wedge config_components \in config \rightarrow \mathcal{P}_1(compClass)$
$config_connections \in config \rightarrow \mathcal{P}(connection)$

MACHINE *Arch_assembly*
...
INVARIANT
/* Un composant instance est désigné par un nom et possède un état initial et un état courant*/
$compInstance \subseteq COMP_INSTANCES \wedge compInstance_name \in compInstance \rightarrow INSTANCE_NAME \wedge$
$initiation_state \in compInstance \rightarrow \mathcal{P}\ (attribute_value) \wedge$
$current_state \in compInstance \rightarrow \mathcal{P}\ (attribute_value) \wedge$
/* Un assemblage est constitué d'instances et de connexions*/
$assm \subseteq ASSEMBLIES \wedge assm_components \in assm \rightarrow \mathcal{P}_1(compInstance) \wedge$
$assm_connection \in assm \rightarrow \mathcal{P}(connection)$

TABLE 2 – Formalisation des trois niveaux de Dedal

Règle de réalisation : Une classe de composant CL réalise un rôle CR si le type de CL est conforme à (*matches*) CR. La description du type de CL doit contenir au moins syntaxiquement les même interfaces que le rôle CR ou des spécialisations des interfaces de CR. Ceci est énoncé formellement comme suit :

$$
\begin{array}{l}
realizes \in compClass \leftrightarrow compRole \wedge \\
\forall\ (CL,CR).(CL \in compClass \wedge CR \in compRole \\
\Rightarrow \\
\quad ((CL,CR) \in realizes \\
\qquad \Leftrightarrow \\
\quad \exists\ CT.(CT \in compType \wedge (CT\ ,\ CR) \in matches \wedge (CL,CT) \in class_implements)) \\
)
\end{array}
$$

$$
\begin{array}{l}
matches \in compType \leftrightarrow compRole \wedge \\
\forall\ (CT,CR).(CT \in comType \wedge CR \in compRole \\
\Rightarrow \\
((CT,CR) \in matches \\
\quad \Leftrightarrow \\
\exists (inj).(inj \in comp_interfaces(CR) \rightarrowtail\!\!\!\!\!\!+\;\; comp_interfaces(CT) \wedge \\
\quad \forall\ (int).(int \in interface \\
\qquad \Rightarrow \\
\quad inj(int) \in int_substitution[\{int\}]) \\
)))
\end{array}
$$

La formalisation des trois niveaux de Dedal est un pas essentiel pour établir les règles de consistance et de conformité entre les niveaux. Ces règles constituent la deuxième proposition de ce papier que nous présentons dans la section suivante.

4 Vérification des architectures à trois niveaux

A l'instar de la formalisation du modèle Dedal, cette section présente des règles génériques pouvant être appliquées sur une architecture en général à savoir les règles de consistance et des règles spécifiques au modèle Dedal qui définissent les relations entre chaque niveau en vue de vérifier la conformité d'un niveau par rapport à un autre.

4.1 Règles de consistance d'une architecture

Une architecture est dite consistante si elle répond à toutes les fonctionnalités décrites dans le cahier des charges (complétude) et si tous ses composants sont correctement connectés (exactitude).
Complétude : Une architecture *arch* est dite complète si pour chacun de ses clients *cl*, il existe une connexion *conn* tel que *cl* est dans l'une des extrémités de *conn*. Formellement :

$$
\begin{array}{l}
\forall\ (arch,\ cl).\\
\quad (arch \in architecture \wedge cl \in client \wedge cl \in arch\_clients(arch)\\
\qquad \Rightarrow\\
\quad \exists\ conn.\\
\quad\ (conn \in connection \wedge conn \in arch\_connections(arch) \wedge\\
\quad\ cl \in \mathbf{dom}(\{conn\})))
\end{array}
$$

La règle d'exactitude complète la règle précédente en imposant que toutes les connexions soient correctes (*i.e* : entre un client et un serveur compatibles).
Exactitude : Pour tout client *cl* et tout serveur *se*, si *cl* et *se* sont connectés, alors leurs interfaces *int1* et *int2* sont compatibles. Formellement :

$$
\begin{array}{l}
\forall\ (cl,\ se).\\
\quad (cl \in client \wedge se \in server\\
\qquad \Rightarrow\\
\quad\ ((cl,\ se) \in connection\\
\qquad \Rightarrow\\
\quad\ \exists\ (C1,\ C2,\ int1,\ int2).\\
\quad\ (C1 \in component \wedge C2 \in component \wedge C1 \neq C2 \wedge int1 \in interface \wedge int2 \in interface \wedge\\
\quad\ cl = (C1,\ int1) \wedge se = (C2,\ int2) \wedge (int1,\ int2) \in int\_compatible)))
\end{array}
$$

4.2 Les règles de conformité dans Dedal

Les règles de conformité sont basées sur les relations entre les niveaux d'architectures en Dedal (*cf.* Figure 2). Les règles relatives à ces relations sont détaillées dans [6].
Conformité d'une configuration par rapport à sa spécification : Une configuration *Conf* implémente une spécification *Spec* si pour tout rôle *CR* de *Spec*, il existe au moins une classe de composant *CL* dans *Conf* qui le réalise. Formellement :

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

On note qu'un composant rôle peut être réalisé à travers un composant composite qui n'existe pas initialement dans la bibliothèque à composants mais dont la composition est calculée à travers d'autres classes existantes.
Conformité d'un assemblage par rapport à une configuration : Un assemblage *Asm* instancie une configuration *Conf* si toutes les classes de composants de *Conf* possèdent au moins une instance dans *Asm* et toutes les instances qui sont dans *Asm* sont des instances de classes de composants qui sont dans *Conf*. Formellement :

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

5 Conclusion et perspectives

Dedal est un modèle architectural qui permet une représentation explicite et distincte des spécifications, configurations et assemblages constituant les trois niveaux d'une architecture logicielle. Cet article a proposé des règles formelles en B définissant la consistance d'architectures logicielles ainsi que la conformité entre les trois niveaux de Dedal. Les règles ont été validées grâce à l'animateur ProB [4] et en réalisant des vérifications sur des instances de modèles B de Dedal relatives à l'exemple HAS. L'un des futurs objectifs est d'automatiser la transformation en modèles formels de Dedal de leurs descriptions textuelles ou graphiques (tel que UML) afin de permettre une vérification automatique des architectures logicielles. Une perspective de court terme consiste ausssi à établir des règles d'évolution permettant (1) de faire évoluer une architecture à un niveau d'abstraction donné tout en préservant sa consistance et (2) de propager l'impact du changement vers les autres niveaux afin de préserver la conformité et d'éviter les problèmes de dérive et d'érosion.
Sur le plan pratique, nous envisageons de développer un outil intégré autour d'Eclipse permettant l'édition (textuelle et graphique) et la vérification des descriptions d'architectures en Dedal et la gestion automatique de l'évolution architecturale.

Références

[1] Jean-Raymond Abrial. *The B-book : Assigning Programs to Meanings*. Cambridge University Press, New York, USA, 1996.

[2] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3) :213–249, July 1997.

[3] Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. FCA-based service classification to dynamically build efficient software component directories. *International Journal of General Systems*, pages 427–453, 2008.

[4] Michael Leuschel and Michael Butler. ProB : An automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2) :185–203, February 2008.

[5] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1996.

[6] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Huaxi (Yulin) Zhang. Formal modeling of software architectures at three abstraction levels. *Rapport technique*, 2014.

[7] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4) :40–52, October 1992.

[8] Ian Sommerville. *Software engineering (9th edition)*. Addison-Wesley, 2010.

[9] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., and Jason E. Robbins. A component- and message-based architectural style for GUI software. In *Proceedings of the 17th ICSE*, pages 295–304, New York, USA, 1995. ACM.

[10] Huaxi (Yulin) Zhang, Christelle Urtado, and Sylvain Vauttier. Architecture-centric component-based development needs a three-level ADL. In *Proceedings of the 4th ECSA*, volume 6285 of *LNCS*, pages 295–310, Copenhagen, Denmark, August 2010. Springer.

[11] Huaxi (Yulin) Zhang, Lei Zhang, Christelle Urtado, Sylvain Vauttier, and Marianne Huchard. A three-level component model in component-based software development. In *Proceedings of the 11th GPCE*, pages 70–79, Dresden, Germany, September 2012. ACM.