

République Tunisienne
Ministère de l'Enseignement Supérieur,
de la Recherche Scientifique et de la
Technologie

Université de Sfax
Ecole Nationale d'Ingénieurs de Sfax



Cycle de Formation Doctorale
dans la Discipline Informatique
***Nouvelles Technologies des
systèmes informatiques dédiés***

Mémoire de MASTERE
N° d'ordre: 2007– 323

MEMOIRE

présenté à

l'École Nationale d'Ingénieurs de Sfax

en vue de l'obtention du

MASTERE

Nouvelles Technologies des Systèmes Informatiques Dédiés

par

Lamia YANGUI BOUAZIZ

UNE DEMARCHE DE VALIDATION DE
SPECIFICATIONS FORMELLES DES ARCHITECTURES
LOGICIELLES

soutenu le 3 juillet 2007, devant le jury composé de :

Mme. Hanène BEN ABDALLAH

Présidente

M. Kaïs HADDAR

Membre

M. Mohamed JMAIEL

Encadreur

Mastère en Nouvelles Technologies des Systèmes Informatiques Dédiés

Une démarche de validation de spécifications
formelles des architectures logicielles

Lamia YANGUI BOUAZIZ

Ecole Nationale des Ingénieurs de Sfax

3 juillet 2007

DEDICACES

A l'âme de ma mère et à mon père,

A mon mari *Mohamed* et ma fille *Emna*,

A toute la famille *YANGUI* et en particulier ma sœur *Dorra*,

A toute la famille *BOUAZIZ*,

Et à toi, je dédie ce qui suit.

REMERCIEMENTS

En premier lieu, je tiens à remercier tout particulièrement Monsieur Mohamed JMAIEL, maître de conférences à l'Ecole Nationale des Ingénieurs de Sfax, pour son encadrement, sa patience, et sa disponibilité permanente et pour l'aide qu'il m'a apporté afin de mener à bien réaliser ce projet et les remarques constructives qu'il m'a fournies.

Je tiens à remercier Madame Hanêne BEN ABDALLAH, maître de conférence à la Faculté des Sciences Economiques et de Gestion de Sfax, d'avoir accepté de présider le jury de ce Mastère. Ma gratitude s'adresse aussi à Monsieur Kais HADDAR, maître assistant à la Faculté de Sciences de Sfax pour avoir accepté d'évaluer ce travail.

Je tiens à remercier vivement Monsieur Riadh BEN HALIMA, assistant contractuel à l'Ecole Nationale des Ingénieurs de Sfax, pour sa collaboration, son enthousiasme, ses encouragements continus, ses précieux conseils qu'il m'a fournis le long de ce travail.

Mes plus vifs remerciements s'adressent, également, à Madame Imen AMOURI LOULOU et Madame Soumaya MARZOUK HENTATI, doctorantes, pour les nombreux conseils qu'elles m'ont donnés, les remarques qu'elles m'ont faites et qui ont participé énormément à la réalisation de ce présent travail.

Je remercie aussi toute l'équipe de l'unité de Recherche ReDCAD, pour la bonne ambiance. Je remercie particulièrement Monsieur Slim Kallel, Mademoiselle Fatma KRICHEN, Monsieur Mohamed Nadhmi MILEDI pour les discussions et l'aide qu'ils m'ont apporté tant pour le travail que pour mon moral.

Je voudrais enfin remercier ma famille et mes amis pour le soutien continuel qu'ils m'ont apporté.

Table des matières

Introduction générale.....	9
-----------------------------------	----------

Chapitre 1 : Les techniques de validation des besoins

1.1 Introduction.....	11
1.2 La validation des besoins.....	11
1.2.1 La révision.....	12
1.2.2 Le prototypage.....	13
1.2.3 Les analyses formelles.....	13
1.2.4 La simulation.....	14
1.2.3 Les scénarios.....	14
1.3 La validation des spécifications formelles.....	14
1.3.1 Problématique.....	14
1.3.2 Travaux existants.....	15
1.4 Conclusion	15

Chapitre 2 : une démarche de validation de spécifications formelles des architectures logicielles

2.1 Introduction	16
2.2 Les architectures logicielles des systèmes informatiques dynamiques à base de composants abstraits.....	16
2.3 L'approche adoptée pour la spécification d'un style architectural en notation Z.....	17
2.4 Les erreurs types dans une spécification formelle d'un style architectural.....	19
2.5 Définition d'une démarche de validation de spécifications d'architectures logicielles.....	20
2. 5.1 Caractérisation des contraintes en langage naturel.....	22
2.5.1.1 Les contraintes simples.....	22
2.5.1.2 Les contraintes composées.....	23
2.5.1.3 Les contraintes avec quantificateur.....	24
2.5.2 Les scénarii proposés pour chaque type de contraintes.....	24
2.5.2.1 Les contraintes simples.....	24
2.5.2.2 Les contraintes composées.....	30

2.5.2.3 Les contraintes avec quantificateur.....	32
2.5.3 Les étapes de la démarche	33
2.6 Conclusion.....	36

Chapitre 3 : Simulateur de test d'architectures logicielles

3.1 Introduction.....	37
3.2 Présentation du simulateur.....	37
3.3 Conception du simulateur.....	41
3.3.1 Diagrammes de classes préliminaires.....	44
3.3.2 Diagrammes de séquences.....	48
3.3.3 Diagrammes de classes de conception.....	50
3.4 Implémentation du simulateur.....	53
3.4.1 Langages et environnements.....	53
3.4.2 Détails d'implémentation.....	55
3.4.3 Présentation de l'interface du simulateur.....	59
3.5 Plug-in pour l'IDE Eclipse.....	60
3.6 Conclusion.....	61

Chapitre 4 : Etude de cas « l'édition coopérative »

4.1 Introduction.....	62
4.2 L'édition coopérative.....	62
4.2.1 Le cahier de charges.....	62
4.2.2 Le schéma de style.....	62
4.2.3 La validation de la spécification.....	64
4.2.4 La spécification après la correction.....	72
4.3 Conclusion.....	73

Conclusion générale.....74

Bibliographie.....75

Table des figures

Chapitre 2 : une démarche de validation de spécifications formelles d'architectures logicielles

Figure 2.1: Description d'un style architectural.....	18
Figure 2.2: Schéma d'opération.....	18
Figure 2.3: Le processus de vérification de complétude et conformité d'une spécification.....	22
Figure 2.4: x est lié au plus à n y_i	26
Figure 2.5: y est l'image d'au plus n x_i	28
Figure 2.6: au plus n x_i font une relation R	28
Figure 2.7: au plus n y_i sont images par la relation R	29
Figure 2.8: x_i ne peut pas faire une relation R et R_I en même temps.....	29
Figure 2.9: y_i ne peut être image par R et R_I en même temps.....	30
Figure 2.10: au plus 4 x_i font une relation R	34
Figure 2.11. Le processus de vérification des contraintes.....	35

Chapitre 3 : Simulateur de test d'architectures logicielles

Figure 3.1. Architecture de l'application.....	37
Figure 3.2. Le scénario du fonctionnement du simulateur	39
Figure 3.3. Diagramme des cas d'utilisation	40
Figure 3.4. Structure d'un fichier XML représentant une architecture.....	41
Figure 3.5. Diagramme de classes préliminaires (première partie).....	45
Figure 3.6. Diagramme de classes préliminaires (deuxième partie).....	46
Figure 3.7. Diagramme de séquence : Ajouter (supprimer) un composant (une connexion).....	49
Figure 3.8. Diagramme de séquence : Tester une architecture logicielle.....	50
Figure 3.9. Diagramme de classes de conception.....	52
Figure 3.10. L'interface de Z/EVES.....	54
Figure 3.11. Simulateur de test d'architectures logicielles	60

Chapitre 4 : Etude de cas « l'édition coopérative »

Figure 4.1 : Le schéma de style de l'application.....	63
Figure 4.2 : Contrainte numéro 5, scénario 1.....	64
Figure 4.3 : Contrainte numéro 5, scénario 2.....	64
Figure 4.4 : Contrainte numéro 8, scénario 1.....	65
Figure 4.5 : Contrainte numéro 8, scénario 2.....	65
Figure 4.6 : Contrainte numéro 8, scénario 3.....	66
Figure 4.7 : Contrainte numéro 10, scénario 1.....	67
Figure 4.8 : Contrainte numéro 10, scénario 2.....	67
Figure 4.9 : Contrainte numéro 11, scénario 1.....	68
Figure 4.10 : Contrainte numéro 11, scénario 2.....	68
Figure 4.11 : Contrainte numéro 11, scénario 3.....	69
Figure 4.12 : Contrainte numéro 11, scénario 4.....	69
Figure 4.13 : Contrainte numéro 12, scénario 1.....	70
Figure 4.14 : Contrainte numéro 12, scénario 2.....	71
Figure 4.15 : Contrainte numéro 14, scénario 1.....	71
Figure 4.16 : la spécification du schéma de style rectifiée.....	72
Figure 4.17: une configuration architecturale type.....	73

Liste des Tableaux

Chapitre 2 : une démarche de validation de spécifications formelles d'architectures logicielles

Tableau 2.1: le système a au plus n composants de type X	24
Tableau 2.2: le système a au moins n composants de type X	25
Tableau 2.3: le système a au moins n et au plus m composants de type X	25
Tableau 2.4: x est lié au plus à n composants de type Y	6
Tableau 2.5: x est lié au moins à n composants de type Y	27
Tableau 2.6: x est lié au moins à n et au plus à m composants de type Y	27

Chapitre 3 : Simulateur de test d'architectures logicielles

Tableau 3.1. Tableau explicatif des diagrammes de classes.....	47
Tableau 3.2. Certaines balises d'un fichier .zev et leur signification	54

Introduction générale

Un grand nombre d'études [13,20] ont montré que les échecs dans la mise en œuvre et l'utilisation des systèmes informatiques sont dus à une mauvaise compréhension des besoins auxquels ces systèmes tentent de répondre.

Boehm [4] de son côté a rapporté que la correction d'une erreur durant la phase de codage coûte entre cinq et dix fois plus que sa correction durant la phase de spécification des besoins.

Afin de corriger cette situation, il est nécessaire de définir des méthodes, des techniques et des outils pour élucider, valider et représenter de manière adéquate et structurée les besoins relatifs aux systèmes à développer. C'est l'objectif que s'est fixé l'ingénierie des besoins.

Dans ce mastère, nous nous intéressons à la validation de spécifications formelles des architectures logicielles dynamiques des systèmes informatiques à base de composants abstraits.

Nous cherchons à vérifier que toutes les contraintes exprimées en langage naturel dans le cahier de charges sont spécifiées conformément aux exigences et attentes de l'utilisateur. Nous utilisons une approche formelle utilisant la notation Z et un outil de spécifications formelles fournissant un support de preuves. Ceci nous permet d'avoir une spécification formelle non ambiguë, claire et consistante. Toutefois, nous remarquons que la complétude et la conformité d'une spécification formelle par rapport aux besoins ne sont pas toujours garanties. Nous constatons aussi que certaines erreurs (sémantiques) échappent à l'outil de spécification. De plus, nous ne pouvons pas détecter ces erreurs par la révision de la spécification vue sa complexité. La simulation des spécifications formelles [5] paraît une solution mais elle demeure incomplète si elle ne guide pas l'utilisateur pendant le processus de validation des besoins.

Pour remédier à ce problème, nous proposons une démarche de validation de spécifications formelles des architectures logicielles basée sur le test de scénarios et utilisant un simulateur de test d'architectures logicielles.

Notre démarche définit un ensemble de scénarios (configurations architecturales) à tester par rapport au style architectural spécifié pour chaque type de contraintes informelles et selon les résultats, l'utilisateur constate s'il y a erreur dans la spécification ou non.

Ce mémoire présente dans le premier chapitre un état de l'art pour la validation des besoins, définit la démarche proposée pour la validation des spécifications formelles des architectures logicielles dans le chapitre suivant. Nous présentons le simulateur, sa conception et son implémentation dans le troisième chapitre. Enfin, nous illustrons notre démarche par une étude de cas : nous considérons l'application de " l'édition coopérative ".

Les techniques de validation des besoins

1.1 Introduction

Un grand nombre d'études [13,20] ont montré que les échecs dans la mise en œuvre et l'utilisation des systèmes informatiques sont dus à une mauvaise compréhension des besoins auxquels ces systèmes tentent de répondre. Un exemple édifiant est celui de la refonte du système d'information du SAM londonien (gestion des ambulances d'urgence) qui, à cause d'une mauvaise compréhension des besoins, a abouti à plusieurs décès.

Les efforts requis pour corriger les erreurs découlant de cette mauvaise compréhension des besoins sont très importants. Afin de corriger cette situation, il est nécessaire de définir des méthodes, des techniques et des outils pour élucider, valider et représenter de manière adéquate et structurée les besoins relatifs aux systèmes à développer. Ce travail devrait permettre, dans le futur, de développer des systèmes plus fiables. C'est l'objectif que s'est fixé l'ingénierie des besoins.

Dans ce chapitre, nous allons mettre l'accent sur les techniques de validation des besoins et en particulier la validation des spécifications formelles afin d'aboutir à une solution permettant de valider les spécifications formelles des architectures logicielles.

1.2 La validation des besoins

L'ingénierie des besoins est un ensemble d'activités pour découvrir, analyser, documenter, valider et maintenir un ensemble d'exigences pour un système [25].

Zave [28] fournit également une des plus claires définitions de l'ingénierie des besoins :

" Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behaviour, and to their evolution over time and across software families."

Dans notre travail, nous allons nous concentrer sur la validation des besoins.

La validation des besoins est le processus établissant l'adéquation, la complétude et la consistance d'une spécification des besoins avec les attentes de l'utilisateur. Puisque la validation des besoins est placée dans les premières phases du cycle de vie d'un logiciel et puisque une erreur dans les besoins découverte en un stade tardé est très coûteuse à fixer [4], la validation des besoins est l'activité de vérification et validation la plus décisive pour un logiciel.

La validation des besoins est difficile pour deux raisons [23]. La première est de nature philosophique et concerne la question de vérité et connaissance. La seconde raison est sociale et concerne la difficulté d'atteindre l'accord de plusieurs utilisateurs avec des opinions incompatibles.

Il y a cinq approches standards [23,24] pour évaluer l'adéquation, la complétude et la consistance des spécifications des besoins par rapport aux exigences et attentes de l'utilisateur :

- mener des révisions pour les spécifications des besoins : inspection des besoins.
- utiliser des fonctions de test et analyse intégrées pour les outils de spécification (analyses automatiques de complétude et consistance).
- implémenter et évaluer un prototype.
- faire une simulation ou encore animer les spécifications.
- utiliser des scénarios.

Les techniques telles que l'inspection et l'analyse formelle se concentrent sur la cohérence des descriptions des besoins : sont elles consistantes et sont elles structurellement complètes ?

Par contre, les techniques telles que l'implémentation de prototype, l'animation des spécifications et l'utilisation des scénarios testent la correspondance avec les besoins réels de l'utilisateur : est ce que toutes les attentes de l'utilisateur sont couvertes ?

1.2.1 La révision

La technique de révision des spécifications implique un groupe de personnes examinant un document (la spécification) dans le but de trouver des fautes ou découvrir des anomalies.

Toutes les conclusions et décisions du groupe sont collectées et données aux responsables aux corrections. Ils existent plusieurs techniques de révision : les inspections [8], les révisions techniques (eng technical reviews) [10].

Bien que la plupart des auteurs soient d'accord sur le fait que la révision doit être un processus formel, il n'y a aucun accord sur une méthode ou procédure à suivre lors de l'inspection d'une spécification des besoins.

1.2.2 Le prototypage

Un prototype d'un système logiciel est une implémentation incomplète du système qui imite le comportement que désire l'utilisateur. Les utilisateurs essaient le prototype comme s'il s'agit du système à développer.

Nous ne pouvons pas considérer ces deux techniques pour la validation des besoins. En effet, les systèmes logiciels sont de plus en plus complexes et les spécifications de leurs besoins sont de plus en plus difficiles à réviser. En plus, les prototypes sont utiles principalement pour valider l'interface de l'utilisateur et le comportement interactif du système mais ils sont moins susceptibles pour vérifier la complétude et la consistance des besoins.

1.2.3 Les analyses formelles

Ils existent des approches basées sur les méthodes formelles (les analyses formelles) et elles ont prouvées leur efficacité pour la validation des besoins. En particulier, nous citons la méthode formelle SCR (Software Cost Reduction) et les méthodes formelles légères.

SCR est une méthode formelle pour la spécification des besoins des systèmes informatiques se basant sur une notation tabulaire assez familière pour l'utilisateur. Plusieurs outils supportant la méthode SCR [3,15,16]. Ces outils fournissent un éditeur de spécification, un simulateur pour exécuter symboliquement la spécification, un vérificateur de consistance qui vérifie que la spécification est bien formée (corriger la syntaxe et les types, vérifier qu'il n'y a pas de cas manquants ni des définitions circulaires ...) et un vérificateur pour analyser les propriétés critiques de la spécification.

Les méthodes formelles légères (lightweight formal methods) sont assez répandues dans la littérature du développement logiciel [9,17]. Dans le contexte de l'ingénierie des besoins, ces méthodes sont qualifiées de légères puisque le coût de leur adaptation est faible par rapport au processus total de l'ingénierie des besoins [12]. Souvent, les méthodes formelles exécutent une analyse partielle pour des spécifications partielles [7]. Elles n'ont pas besoin de traduire tout un document informel en un autre formel ni de maintenir en parallèles les versions formelles et informelles des spécifications [18].

1.2.4 La simulation

Dans le contexte de l'ingénierie des besoins, la simulation est définie comme la visualisation du comportement d'une instance de l'architecture du système. Le langage de modélisation utilisé pour spécifier l'architecture du système doit dépendre d'une sémantique d'exécution bien définie. Basé sur la sémantique, un outil de simulateur peut visualiser le modèle d'architecture par l'interprétation directe. Cependant, les bénéfices des modèles logiciels sont fixés par les outils de simulation correspondants.

En conséquence, plusieurs recherches sont concentrées dans cet axe et beaucoup d'outils sont développés pour valider la spécification des besoins. Parmi les solutions industrielles, nous citons : ObjectGeode [29] qui est un ensemble d'outils commerciaux consacrés à l'analyse, la conception, la vérification et la validation par simulation et test d'applications temps-réel et distribuées. Il supporte une intégration cohérente et complémentaire des approches orientées objet et temps-réel basées sur les langages standards tels qu'UML, SDL (Specification and Design Language) et MSC (Message Sequence Charts). Object Time Développeur [30] est un environnement de modélisation graphique pour la conception orientée objet (avec UML) et la simulation de systèmes temps réel. Il permet aussi la génération de code en C++. IBEPACE [31] est utilisé pour simuler les systèmes développés avec l'orientée objet en se basant sur les réseaux de Petri temporaires et la logique floue. Il permet de modéliser, simuler, analyser et optimiser les processus commerciaux, techniques et logistiques. Mais, il ne fournit pas de méthode de preuves.

1.2.5 Les scénarios

Ce sont des exemples d'exécution du système. Ils permettent d'imiter le fonctionnement du système en décrivant un ensemble de séquences d'interaction.

1.3 La validation des spécifications formelles

1.3.1 Problématique

Bien que le formel présente de nombreux avantages tels que la clarté, la non ambiguïté grâce au raisonnement mathématique et logique, la précision, il est caractérisé par sa complexité et la difficulté de sa compréhension. Pour cette raison, les concepteurs hésitent avant de décider d'utiliser le formel dans leurs spécifications. De plus, ils trouvent des difficultés à valider les besoins avec l'utilisateur qui est généralement non expert du formel.

Il faut donc faire recourt aux techniques de validation des besoins. La révision et le prototypage ne peuvent pas être solution. Aussi, les analyses formelles ne résolvent pas le problème puisque nous demandons plus qu'elles fournissent.

La simulation de scénarios parait une solution puisque elle cache le coté formel et présente l'architecture du système d'une manière simple. Les scénarios permettent de valider les besoins de l'utilisateur par la simulation du fonctionnement du système.

1.3.2 Travaux existants

Nous nous intéressons aux spécifications formelles des architectures logicielles à base de composant. Nous cherchons à vérifier la complétude et la conformité des spécifications.

Tous les simulateurs déjà mentionnés ne supportent pas la simulation d'un modèle à base de composant. En plus, la plupart d'entre eux n'utilisent pas les spécifications formelles comme entrée du processus de simulation.

Par contre, le simulateur proposé par [2] permet de visualiser le comportement d'une application à base de composant en utilisant la notation UML2.0 tout en respectant le style architectural. Via son interface graphique, le simulateur permet aux concepteurs, qui ne maîtrisent pas les techniques de description formelle, d'appréhender eux mêmes le comportement des systèmes qu'ils conçoivent. De plus, il permet de produire plusieurs instances d'une architecture. Si l'architecture conçue est erronée (la spécification Z initiale permet cette configuration), le simulateur offre la possibilité de modifier la spécification en Z (ajout / suppression de contraintes, de composants et de connexions) et de la recharger afin de reprendre la simulation. A la fin du processus de simulation, il y a synthèse d'une spécification correcte.

Ce travail peut aider à la vérification de la complétude et conformité sauf qu'il ne présente pas une démarche à suivre pour valider la complétude et la conformité de la spécification.

1.4 Conclusion

Notre but est de vérifier la complétude et la conformité des spécifications formelles des architectures logicielles à base de composants abstraits. Nous proposons de combiner deux techniques de validation de besoins : la simulation et l'utilisation de scénarios. Nous proposons donc de définir une démarche de validation de spécifications formelles basée sur le test de scénarios en utilisant un simulateur d'architectures logicielles.

Une démarche de validation de spécifications formelles d'architectures logicielles

2.1 Introduction

La production de spécifications formelles nous aide à comprendre les besoins du client et à clarifier les intentions grâce à la notation mathématique permettant une unique interprétation. De plus, l'utilisation d'un outil de développement fournissant un support de preuve nous permet d'avoir une spécification consistante et syntaxiquement valide. Mais, nous ne pouvons pas garantir ni la complétude ni la conformité de la spécification par rapport aux besoins généralement informels, puisque certaines erreurs demeurent inaperçues lors de la vérification. Pour remédier à ce problème, nous proposons une démarche de validation de spécifications formelles des architectures logicielles basée sur le test de configurations architecturales. Dans ce chapitre, nous commençons par présenter le domaine des applications considérées et l'approche de spécification des architectures logicielles, ensuite, nous citons quelques erreurs types des spécifications et enfin nous définissons notre démarche.

2.2 Les architectures logicielles des systèmes informatiques dynamiques à base de composants abstraits

L'architecture logicielle d'une application se définit comme une spécification abstraite en termes de modules logiciels qui la constituent et des interactions entre ces modules [5] et laisse en second plan la programmation des applications.

Un module logiciel ou encore composant logiciel [21] est une entité autonome capable d'interagir avec son environnement à l'aide d'une interface. Deux parties définissent un composant.

Une première partie, dite interne, correspond à son implantation et permet la description du fonctionnement interne du composant. La seconde partie, dite externe, correspond à son interface et décrivant les services fournis et requis par le composant.

L'interface du composant est constituée de points d'interaction, ou ports, qui lui permet de communiquer avec son environnement. L'interconnexion entre les ports des composants peut être réalisée via des connecteurs.

Le connecteur [1] correspond à un élément d'architecture qui modélise de manière explicite les interactions entre un ou plusieurs composants. Il contient les informations concernant les règles d'interactions entre les composants.

Composants et connecteurs sont des types pouvant être instanciés et assemblés à partir de leurs interfaces pour former une configuration.

Chaque architecture ou encore configuration architecturale fait partie d'un ensemble appelé style. Un style architectural caractérise une famille d'architectures logicielles qui sont reliées par des propriétés structurelles et sémantiques partagées [22].

Selon [11,19], un style architectural détermine le vocabulaire des composants et des connecteurs utilisés dans les architectures de ce style ainsi qu'un ensemble de contraintes sur la façon dont ils peuvent être combinés. De plus, un style architectural comporte un ensemble de conditions devant être vérifiées pour toute configuration appartenant à ce style.

Dans notre travail, nous considérons les architectures logicielles dynamiques. La construction d'applications avec une architecture dynamique résulte de l'ajout ou la suppression de composants et de connexions entre eux conformément au style architectural.

Nous utilisons les méthodes formelles pour décrire la dynamique des architectures logicielles. Nous utilisons la notation Z [25] pour spécifier les styles architecturaux ainsi que les opérations de reconfiguration. L'utilisation d'une approche formelle nous a permis de raisonner rigoureusement sur les propriétés architecturales. Nous définissons cette approche dans la section suivante.

2.3 L'approche adoptée pour la spécification d'un style architectural en notation Z

Dans ce travail, nous avons choisi de spécifier les styles architecturaux avec le langage Z tout en adoptant une approche proposée dans [19].

C'est une approche formelle intégrée pour la spécification des architectures dynamiques orientées composants. Il s'agit de l'intégration d'une approche fonctionnelle et d'une approche structurelle basée sur les grammaires de graphes. Cette approche permet la simplification de la spécification,

l'amélioration de la lisibilité et de la compréhension et la prise en charge de la dynamique. En effet, dans [19], l'auteur utilise le langage formel Z pour décrire le style architectural qui doit être préservé durant l'évolution d'une architecture, d'une part. D'autre part, la dynamique est décrite via des règles de transformation de graphes gardées dont le corps décrit les contraintes structurelles et dont les gardes décrivent principalement les contraintes fonctionnelles du système. La sémantique est exprimée avec la notation Z également, obtenant ainsi une approche unifiée qui prend en charge l'aspect statique et l'aspect dynamique.

Selon cette approche, une spécification formelle d'une architecture dynamique d'un système orienté composants est formée par un schéma d'état et des schémas d'opérations Z.

En effet, l'aspect statique d'une architecture logicielle représenté par le style architectural est traduit par un schéma d'état. Ce schéma déclare les types des composants (N_i) ainsi que les relations entre ces types (R_i) et dans sa partie prédicative, il définit les contraintes architecturales à respecter (P_i).

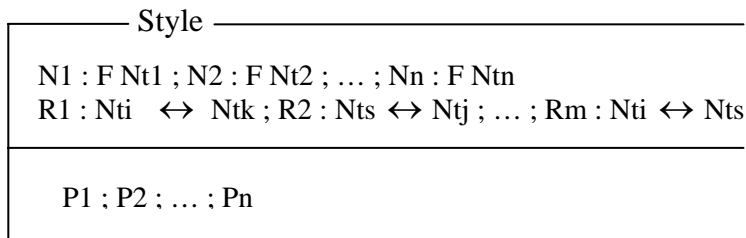


Figure 2.11: Description d'un style

De son côté, l'aspect dynamique d'une architecture logicielle (tel que l'ajout ou la suppression d'un composant, la connexion ou la déconnexion d'un composant) est spécifié par des schémas d'opérations. Ces derniers définissent les relations entre les entrées ($Par ?$), les sorties ($Par !$), et les changements entre un état avant et un état après d'une configuration architecturale.

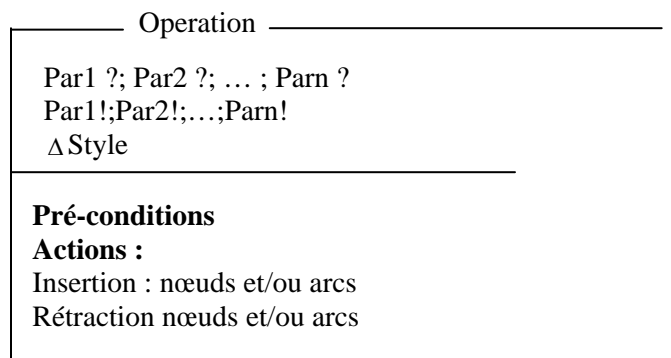


Figure 2.12: Schéma d'opération.

La notation Δ permet d'inclure deux copies du même schéma d'état, *Style* et *Style'*, pour l'état *avant* et l'état *après*, respectivement. Dans la partie prédictive, on précise les pré conditions qui doivent exister pour que l'opération de reconfiguration ait lieu et les actions qui résultent de cette opération (insertion ou suppression de nœuds et/ou arcs).

Pour rédiger les spécifications Z des styles architecturaux, nous avons utilisé l'outil Z/EVES.

Ce dernier permet de vérifier la spécification grâce au support de preuve qu'il fournit. Toutefois, nous constatons que certaines erreurs sont indétectables. Dans la section suivante, nous citons quelques unes.

2.4 Les erreurs types dans une spécification formelle d'un style architectural

Lors du développement d'une spécification formelle, le concepteur peut commettre des erreurs d'inattention. En effet, ce type d'erreurs généralement n'influe pas la logique dans la spécification, c'est pourquoi elles échappent à l'environnement de développement (tel que Z/EVES dans ce travail) lors de la vérification de la consistance de la spécification. Par contre, côté sémantique, ces erreurs peuvent changer complètement le système logiciel à développer. Nous ne pouvons pas déterminer exactement toutes les erreurs, mais par l'expérience et l'étude de plusieurs exemples de spécifications, nous pouvons dire que les erreurs les plus commises sont les :

- erreurs de spécification de domaine d'une relation.
- erreurs de spécification de l'ensemble d'arrivée (image) d'une relation.
- erreurs d'appartenance d'un composant à son ensemble.
- erreurs de spécification de relation : au lieu de mettre une relation $R1$, le concepteur peut écrire $R2$ (exemple *Pull* et *Push*).
- erreurs de cardinalité : ces erreurs accompagnent les contraintes de type :
 - ✓ "le système contient *au plus* (et/ou *au moins*, *exactement*) n composants de type X ".
 - ✓ "un composant de type X est lié *au plus* (respectivement *au moins*, *exactement*) à n composants de type Y ".

L'erreur peut être dans la spécification de l'opérateur de comparaison (au plus, au moins ou exactement) ou bien l'entier n .

- erreurs de spécification d’opérateur logique : par exemple le *OU* sera exprimé par un *ET*, le *SI...ALORS* sera traduit par *SI ET SEULEMENT SI...*
- erreurs d’expression de quantificateur : *pour tout* (respectivement *il existe* ou encore *il existe un seul*) composant de type *X* tel que...
- oubli de spécification de contrainte : le concepteur oublie toute la propriété.

Donc, ces erreurs peuvent changer complètement la sémantique de la spécification d’une part, et d’autre part, il est difficile de les détecter même si le concepteur lit et relit sa spécification vue sa complexité. L’influence de ces erreurs sur les étapes suivantes du cycle de développement du logiciel est très coûteuse. Afin de remédier à ce problème, nous proposons ici une démarche de validation de spécifications formelles d’architectures logicielles.

2.5 Définition d’une démarche de validation de spécifications formelles d’architectures logicielles

La contribution de ce mastère est la définition d’une démarche de test d’architectures logicielles permettant de vérifier la complétude et la conformité d’une spécification formelle rédigée en *Z* par rapport aux besoins exprimés dans le cahier de charges, et ceci spécifiquement aux architectures logicielles des applications à base de composants abstraits.

L’idée est de vérifier une par une toutes les contraintes exprimées en langage naturel par l’utilisateur (qui n’est pas nécessairement un expert du formel).

Vérifier une contrainte signifie vérifier qu’elle est formulée dans la spécification d’une part, et d’autre part, elle est spécifiée conformément aux besoins de l’utilisateur. Ceci ne peut pas se faire par la lecture directe de la spécification formelle mais plutôt par la simulation et le test de certaines configurations architecturales du système spécifié.

En effet, pour une contrainte donnée, correspondent deux ensembles de configurations architecturales : le premier est celui des architectures qui respectent la contrainte, l’autre est celui des architectures ne respectant pas la contrainte : en d’autres termes, elles respectent le ” complément ” de la contrainte. Ces deux ensembles peuvent être finis ou infinis.

Prenons l'exemple de la contrainte suivante :

” le système a au plus 3 composants de type X”.

Parmi les configurations architecturales possibles, nous citons :

Configurations architecturales respectant la contrainte

- Une configuration architecturale avec 3 composants de type X
- Une configuration architecturale avec 2 composants de type X
- Une configuration architecturale avec 1 composant de type X

Configurations architecturales ne respectant pas la contrainte

- Une configuration architecturale avec 4 composants de type X
- Une configuration architecturale avec 6 composants de type X
- Une configuration architecturale avec 7 composants de type X

Nous ne pouvons pas s'intéresser à toutes les configurations possibles : il y a des configurations particulières que nous appelons des cas de test. La particularité de ces cas est dans le fait qu'ils sont à "la frontière" entre les deux ensembles. Ils ont une haute probabilité de découvrir une faute non détectée dans la spécification formelle.

Nous qualifions un cas de test par positif s'il s'agit d'une reconfiguration architecturale respectant la contrainte, et par négatif dans le cas contraire.

Retournons à l'exemple précédent, nous avons donc les cas de test suivants :

- cas de test positif 1 : configuration architecturale avec 3 composants de type X ;
- cas de test positif 2 : configuration architecturale avec 2 composants de type X ;
- cas de test négatif : configuration architecturale avec 4 composants de type X ;

Donc la vérification d'une contrainte se fait par le test des scénarii représentant les cas de test correspondants dans un ordre bien défini.

Pour assurer ces tests, nous faisons recours à un simulateur de test d'architectures logicielles.

Nous parlerons de cet environnement plus en détail dans le chapitre suivant.

En récapitulant, nous disposons d'un ensemble de contraintes informelles, d'une spécification formelle de ces contraintes et d'un simulateur de test d'architectures logicielles.

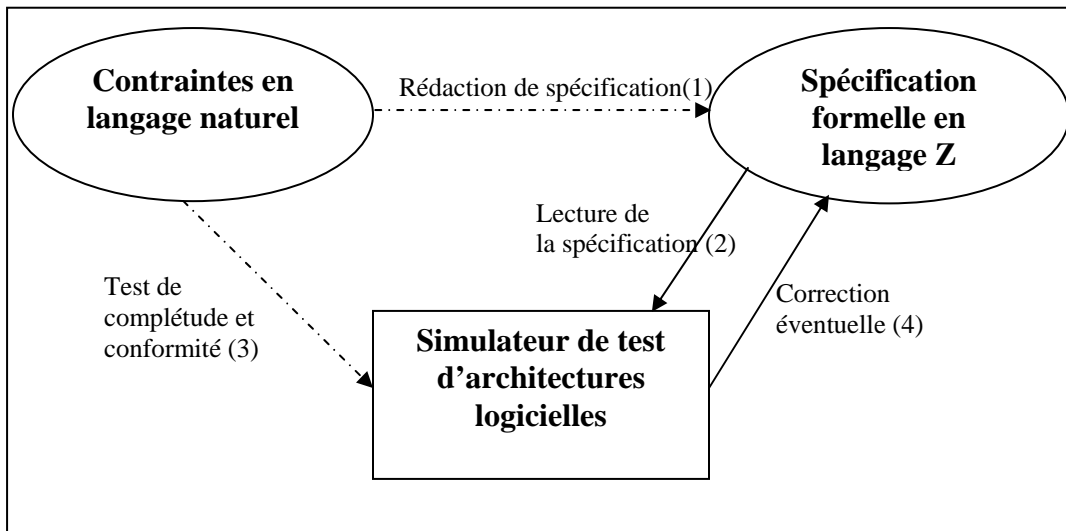


Figure 2.13: *Le processus de vérification de complétude et conformité d'une spécification.*

La première étape est donc le développement de la spécification formelle de l'architecture logicielle répondant aux contraintes informelles de l'utilisateur. Ensuite, le concepteur procède au test de la complétude et la conformité : en suivant les étapes de notre démarche, il testera des configurations architecturales bien définies avec le simulateur. Ce dernier valide les configurations dessinées par rapport au style architectural spécifié dans la première étape de ce processus. Suite aux résultats de test de scénarii restitués par le simulateur, le concepteur peut déduire des erreurs éventuelles dans la spécification et il peut y retourner pour faire les corrections correspondantes.

2.5.1 Caractérisation des contraintes en langage naturel

Nous partons de contraintes décrites dans le cahier de charges et donc écrites en langage naturel. Par conséquent, il faut tout d'abord donner les structures types de ces contraintes.

Nous distinguons trois classes principales :

2.5.1.1 Les contraintes simples

Ce sont les contraintes qui expriment les propriétés des composants et des relations. Nous distinguons quatre sous classes :

✓ **Les contraintes de domaine :**

Ce sont les contraintes qui précisent les ensembles de départ et d'arrivée des relations.

✓ **Les contraintes de cardinalité des ensembles :**

Ce sont les contraintes ayant la structure suivante:

Le système contient au plus (et / ou au moins m , ou exactement) n composant(s) de type X (avec $n, m \in \mathbb{IN}^$).*

✓ **Les contraintes de cardinalité des relations :**

Ce sont les contraintes qui précisent le nombre de relations que peut faire un composant de type donné. Nous nous intéressons aux contraintes les plus utilisées et nous distinguons les structures suivantes :

– *un composant de type X est lié au plus (respectivement au moins, exactement) à n composant(s) de type Y par la relation R .*

– *un composant de type Y est l'image d'au plus (respectivement au moins, exactement) n composant(s) de type X par la relation R .*

– *au plus (respectivement au moins, exactement) n composant(s) de type X font une relation R .*

– *au plus (respectivement au moins, exactement) n composant(s) de type Y sont des images par une relation R .*

– *au plus (respectivement au moins, exactement) n couples de composants connectés par la relation R .*

✓ **Les contraintes de types de relations :**

Ces contraintes permettent de préciser les types de relations que peut faire un composant de type donné. Nous distinguons les structures suivantes :

– *un composant de type X ne peut pas appartenir à l'intersection des domaines de deux relations R et $R1$.*

– *un composant de type Y ne peut pas appartenir à l'intersection des images de deux relations R et $R1$.*

2.5.1.2 Les contraintes composées

Les contraintes simples que nous avons définies dans le sous paragraphe précédent peuvent être composées par les opérateurs logiques pour obtenir une des structures suivantes :

✓ *si condition $C1$ alors condition $C2$.*

- ✓ *condition C1 si et seulement si condition C2.*
- ✓ *condition C1 ou condition C2.*
- ✓ *condition C1 et condition C2.*

Où *condition* est une contrainte simple ou encore une contrainte composée.

2.5.1.3 Les contraintes avec quantificateur

Les contraintes simples et composées déjà définies peuvent être précédées par un quantificateur. Nous aurons alors des structures commençant par une des expressions suivantes :

- ✓ *pour tout* composant de type *X* tel que...
- ✓ *il existe* un composant de type *X* tel que...
- ✓ *il existe un seul* composant de type *X* tel que...

Donc, une fois nous avons caractérisé les contraintes types qui peuvent exprimer les besoins attendus par l'utilisateur, nous proposons dans le paragraphe suivant, et selon les types des contraintes, les scénarii permettant de tester la validité d'une contrainte.

2.5.2 Les scénarii proposés pour chaque type de contraintes

Selon le type de la contrainte, nous proposons des scénarii représentant les cas de test correspondants et nous déterminons aussi le résultat qui doit être restitué par le simulateur de test.

Notation: pour mieux alléger l'écriture, nous allons noter " un cas de test négatif " par l'abréviation " CTN " et " un cas de test positif " par l'abréviation " CTP ".

2.5.2.1 Les contraintes simples

✓ **Les contraintes de cardinalité des ensembles**

- Le système a *au plus n* composants de type *X*, avec *n* un entier naturel non nul :

Tableau 2.7: le système a *au plus n* composants de type *X*

<i>Cas de test</i>	<i>Scénario proposé</i>	<i>Résultat attendu</i>
CTP 1	une configuration architecturale avec $(n-1)$ composants de type <i>X</i>	architecture <i>valide</i>
CTP 2	une configuration architecturale avec n composants de type <i>X</i>	architecture <i>valide</i>
CTN	une configuration architecturale avec $(n+1)$ composants de type <i>X</i>	architecture <i>non valide</i>

- Le système a au moins n composants de type X , avec n un entier naturel non nul :

Tableau 2.8: le système a au moins n composants de type X

<i>Cas de test</i>	<i>Scénario proposé</i>	<i>Résultat attendu</i>
CTN	une configuration architecturale avec $(n-1)$ composants de type X	architecture <i>non valide</i>
CTP 1	une configuration architecturale avec n composants de type X	architecture <i>valide</i>
CTP 2	une configuration architecturale avec $(n+1)$ composants de type X	architecture <i>valide</i>

- Le système a *au moins* n et *au plus* m composants de type X , avec n et m deux entiers naturels non nuls et $n \leq m$:

Tableau 2.9: le système a au moins n et au plus m composants de type X

<i>Cas de test</i>	<i>Scénario proposé</i>	<i>Résultat attendu</i>
CTN 1	une configuration architecturale avec $(n-1)$ composants de type X	Architecture <i>non valide</i>
CTP 1	une configuration architecturale avec n composants de type X	architecture <i>valide</i>
CTP 2	une configuration architecturale avec m composants de type X	architecture <i>valide</i>
CTN 2	une configuration architecturale avec $(m+1)$ composants de type X	architecture <i>non valide</i>

- Le système a *exactement* n composants de type X , avec n un entier naturel non nul :

C'est le cas précédent avec $n=m$.

✓ **Les contraintes de cardinalités des relations :**

- x est lié *au plus* à n composants de type Y , avec n un entier naturel non nul :

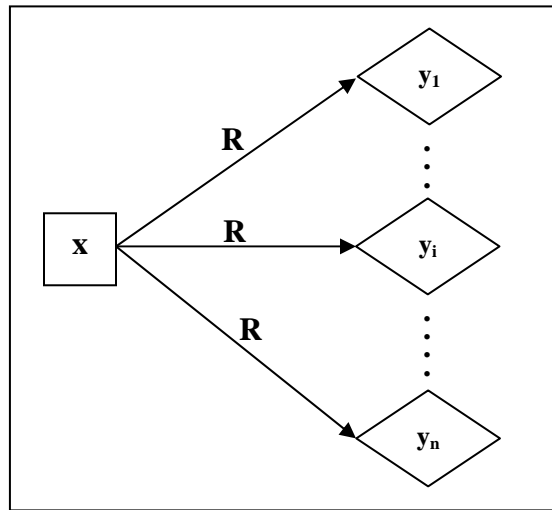


Figure 2.14: x est lié au plus à n y_i

Nous jouons sur le nombre de liaisons que fait un composant x fixé avec les composants y_i . Nous proposons (tableau 2.4) de tester trois configurations architecturales pour vérifier la conformité de cette contrainte.

Tableau 2.10: x est lié au plus à n composants de type Y

<i>Cas de test</i>	<i>Scénario proposé</i>	<i>Résultat attendu</i>
CTP 1	une configuration architecturale avec x lié à $(n-1)$ composants de type Y	architecture <i>valide</i>
CTP 2	une configuration architecturale avec x lié à n composants de type Y	architecture <i>valide</i>
CTN	une configuration architecturale avec x lié à $(n+1)$ composants de type Y	architecture <i>non valide</i>

- x est lié *au moins* à n composants de type Y , avec n un entier naturel non nul :

Tableau 2.11: x est lié au moins à n composants de type Y

<i>Cas de test</i>	<i>Scénario proposé</i>	<i>Résultat attendu</i>
CTN	une configuration architecturale avec x lié à $(n-1)$ composants de type Y	architecture <i>non valide</i>
CTP 1	une configuration architecturale avec x lié à n composants de type Y	architecture <i>valide</i>
CTP 2	une configuration architecturale avec x lié à $(n+1)$ composants de type Y	architecture <i>valide</i>

- x est lié *au moins* à n et *au plus* à m composants de type Y , avec n et m deux entiers naturels non nuls et $n \leq m$:

Tableau 2.12: x est lié au moins à n et au plus à m composants de type Y

<i>Cas de test</i>	<i>Scénario proposé</i>	<i>Résultat attendu</i>
CTN 1	une configuration architecturale avec x lié à $(n-1)$ composants de type Y	architecture <i>non valide</i>
CTP 1	une configuration architecturale avec x lié à n composants de type Y	architecture <i>valide</i>
CTP 2	une configuration architecturale avec x lié à m composants de type Y	architecture <i>valide</i>
CTN 2	Une configuration architecturale avec x lié à $(m+1)$ composants de type Y	architecture <i>non valide</i>

- x est lié *exactement* à n composants de type Y , avec n un entier naturel non nul :

C'est le cas précédent avec $n=m$.

- Un composant de type Y est l'image d'au plus (respectivement au moins, exactement) n composant(s) de type X par la relation R :

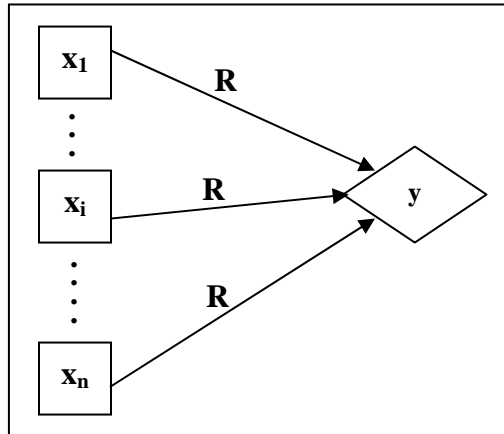


Figure 2.15: y est l'image d'au plus n x_i

Pour cette contrainte, nous suivons le même raisonnement que la contrainte précédente, sauf que, dans ce cas, nous fixons le composant y et nous ajoutons ou supprimons des connexions (x_i, y) de type R .

- Au plus (respectivement au moins, exactement) n composant(s) de type X font une relation R :

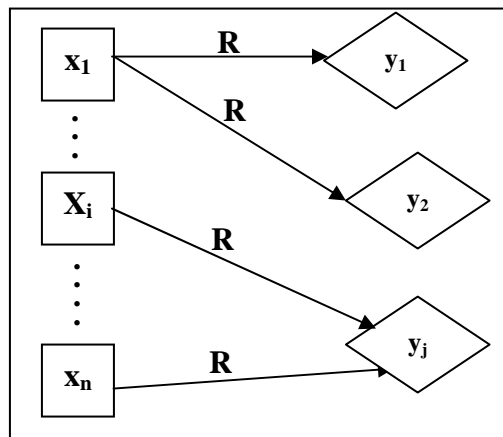


Figure 2.16: au plus n x_i font une relation R

Tout en suivant le même raisonnement, nous comptons dans ce cas les x_i qui font une (ou des) relation(s) R .

- Au plus (respectivement au moins, exactement) n composant(s) de type Y sont des images par une relation R :

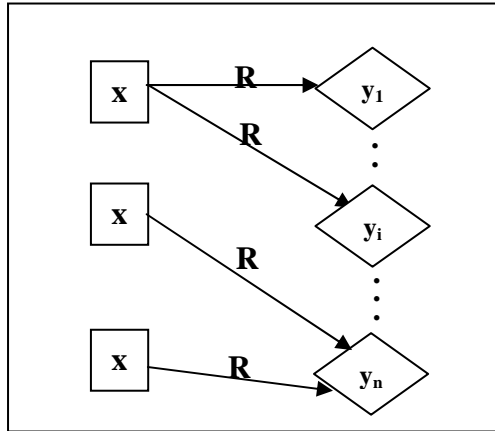


Figure 2.17: au plus n y_i sont images par la relation R

Dans ce cas, nous comptons les y_i qui sont connectés à des x_j .

- Au plus (respectivement au moins, exactement) n couples de composants connectés par la relation R :

Dans ce cas, nous nous intéressons au nombre de connexions faites. Il faut vérifier le nombre de couples de composants reliés par la relation R .

✓ **Les contraintes de types des relations :**

- Un composant de type X ne peut pas appartenir à l'intersection des domaines de deux relations R et R_1 :

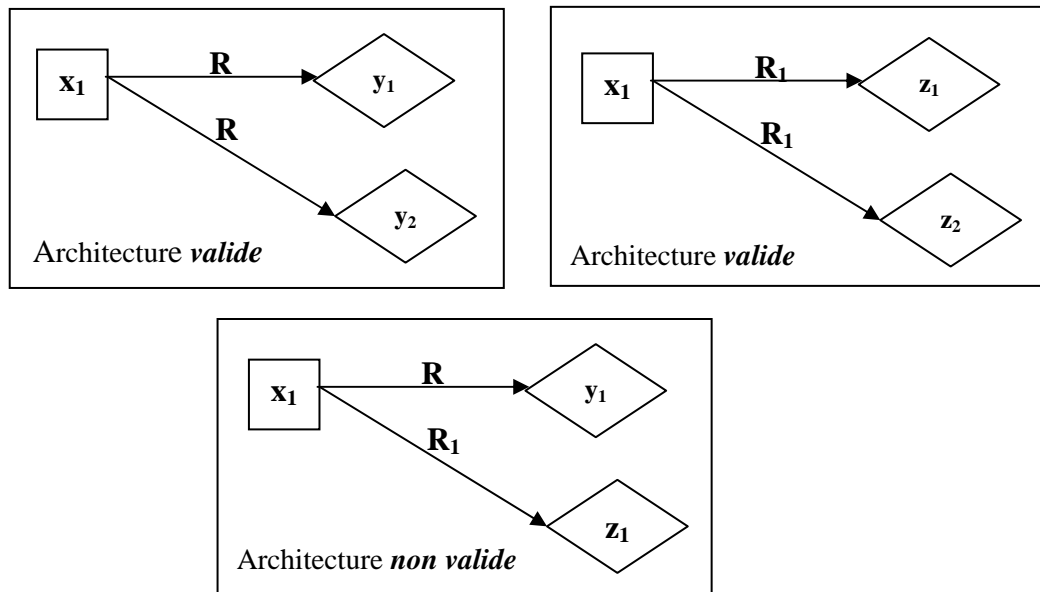


Figure 2.18: x_1 ne peut pas faire une relation R et R_1 en même temps

Nous testons les configurations où un composant de type X fait la relation R uniquement, puis $R1$ uniquement et enfin les deux relations ensemble.

- Un composant de type Y ne peut pas appartenir à l'intersection des images de deux relations R et $R1$:

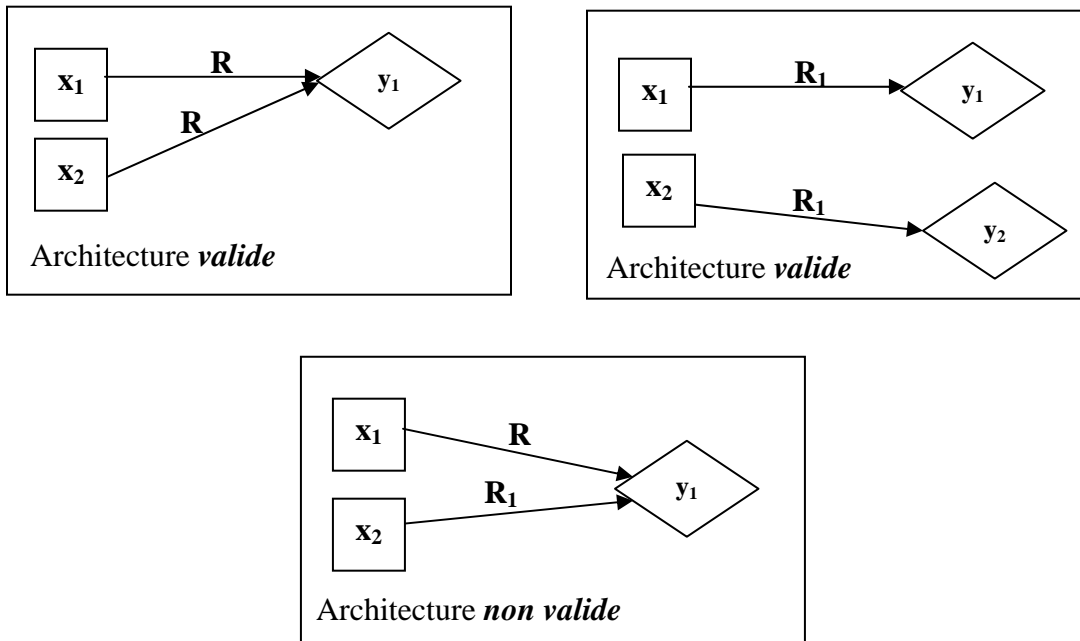


Figure 2.19: y_1 ne peut être image par R et $R1$ en même temps

Remarque :

Pour l'expression élémentaire "x est lié à y par R", nous avons deux cas de test :

- ✓ un CTP : "x est lié à y par R"
- ✓ un CTN : "x n'est pas lié à y par R"

2.5.2.2 Les contraintes composées

- ✓ **Contrainte de type :** "SI condition C1 ALORS condition C2" :

Nous proposons de suivre les étapes suivantes :

- Une première étape où nous testons une configuration architecturale avec un CTP de $C1$ mais un CTN de $C2$.

- Si le résultat est "architecture valide" alors nous sommes sûrs qu'il y a une erreur dans la spécification ou bien la contrainte est non spécifiée.

- Si le résultat est " architecture non valide " alors nous passons à l'étape suivante.

➤ La deuxième étape : nous fixons le CTP pour *C1*, et nous testons les configurations architecturales correspondantes aux CTP de *C2*. Pour chaque configuration, il faut que le simulateur restitue le résultat " architecture valide ". Dans le cas contraire, il faut revoir la spécification pour une erreur éventuelle dans la condition *C2*.

➤ La troisième étape : nous testons une configuration architecturale avec un CTN de *C1* et un CTP de *C2*. Le résultat doit être " architecture valide ". Dans le cas contraire, il faut vérifier que l'opérateur logique qui traduit le si...alors est bien spécifié.

✓ **Contrainte de type : "condition C1 SI ET SEULEMENT SI condition C2" :**

Nous proposons de suivre les étapes suivantes :

➤ la première et la deuxième sont conformes à celles définies pour le type de contrainte précédent.

➤ La troisième étape : nous testons une configuration architecturale avec un CTN de *C1* et un CTP de *C2*.

Le résultat de ce scénario doit être " architecture non valide ". Dans le cas contraire, nous devons vérifier s'il y a erreur de spécification de l'opérateur équivalent traduisant le si et seulement si.

➤ La quatrième étape : nous fixons ce CTP pour *C2*, et nous testons les configurations architecturales correspondant aux CTP de *C1*.

Pour chaque configuration, il faut que le simulateur restitue le résultat " architecture valide ". Dans le cas contraire, il faut revoir la spécification pour une erreur éventuelle dans la condition *C1*.

➤ Pour tester les CTN de *C1* et *C2*, nous devons tester la (ou les) configuration(s) correspondante(s) à un CTN de *C1* et CTN pour *C2*. Le résultat doit être " architecture valide ".

✓ **Contrainte de type : "condition C1 OU condition C2" :**

Pour vérifier cette contrainte, nous sommes amenés à vérifier que *C1*, *C2* et l'opérateur logique *OR* sont bien spécifiés. Pour cela, nous proposons trois étapes :

➤ La première étape : nous testons une configuration architecturale avec un CTN de *C1* et de *C2*. Le résultat doit être " architecture non valide ". Dans le cas d'un résultat différent, nous pouvons penser que la contrainte est non spécifiée ou bien *C1* ou *C2* est erronée ou encore l'opérateur est mal spécifié.

Nous devons tester toutes les configurations pour couvrir tous les CTN de *C1* et *C2*.

➤ La deuxième étape : nous fixons un CTN de *C1* et nous testons les configurations architecturales correspondantes aux CTP de *C2*. Pour chaque scénario, nous devons avoir le résultat "architecture valide". Sinon, il faut vérifier la spécification de *C2* et encore celle de l'opérateur logique.

➤ La troisième étape : c'est l'inverse de l'étape précédente : nous testons les scénarii ayant un CTN de *C2* et couvrant tous les CTP de *C1*. Pour chaque scénario, nous devons avoir le résultat "architecture valide".

✓ **Contrainte de type : "condition C1 ET condition C2" :**

Pour vérifier la complétude de cette contrainte, il faut suivre les étapes suivantes :

➤ La première étape : nous testons une configuration architecturale avec un CTP de *C1* mais un CTN de *C2*. Le résultat doit être "architecture non valide".

Nous devons tester autant de configurations que de CTN de *C2*.

➤ La deuxième étape : c'est l'inverse de la première étape : nous proposons de tester une configuration architecturale avec un CTN de *C1* et un CTP de *C2*. Le résultat doit être "architecture non valide".

Nous devons tester autant de configurations que de CTN de *C1*.

➤ La troisième étape : nous testons les configurations architecturales couvrant tous les CTP de *C1* et *C2*. Tous les résultats doivent être "architecture valide".

2.5.2.3 Les contraintes avec quantificateur

Pour les trois contraintes suivantes, nous testons uniquement le quantificateur et cela suppose que la contrainte est vérifiée auparavant.

✓ **Contrainte commençant par "pour tout" :**

Une seule configuration architecturale permet de vérifier que le quantificateur est bien spécifié. Nous proposons de tester un scénario avec deux composants : un respecte la contrainte et l'autre la fausse : le résultat doit être "architecture non valide".

En effet, si le quantificateur spécifié est "il existe" ou "il existe un seul" alors le résultat devrait être "architecture valide".

✓ **Contrainte commençant par "il existe" :**

Dans ce cas, nous devons tester deux configurations architecturales :

- Une configuration telle qu'aucun composant ne respecte pas la contrainte : le résultat doit être " architecture non valide ". Ainsi, nous garantissons l'existence.
- Une deuxième configuration avec deux composants respectant la contrainte : le résultat doit être " architecture valide ". Nous sommes sûre maintenant que nous n'avons pas fait " il existe un seul ".

✓ **Contrainte commençant par " il existe un seul " :**

Nous testons les deux configurations architecturales du cas précédent. Les deux résultats doivent être " architecture non valide ".

2.5.3 Les étapes de la démarche

Dans ce paragraphe, nous précisons la manière dont le concepteur (ou encore un simple utilisateur) doit procéder pour tester les scénarii.

Tout d'abord, nous allons profiter du fait que le simulateur n'accepte une connexion que si les types des composants connectés sont respectivement des ensembles de départ et d'arrivée de la relation, ainsi nous vérifions les contraintes de domaine.

Ensuite, nous constatons que commencer par le test d'une configuration architecturale idéale (respectant toutes les contraintes) permet de détecter des erreurs très apparentes. Notons que en cas d'erreur (la configuration ne respecte pas le style architectural), le simulateur indique le numéro de la contrainte non respectée. Nous désignons par numéro de la contrainte son ordre de spécification dans la partie déclarative du schéma de style.

Passons maintenant à la vérification de chaque contrainte : le processus de vérification des contraintes est décrit par la figure 2.10.

Notation :

- RA : résultat attendu
- RR : résultat restitué
- Ci : contrainte numéro i dans la spécification formelle
- k : numéro de scénario à tester
- j : numéro de la contrainte non respectée lors du test de la configuration architecturale

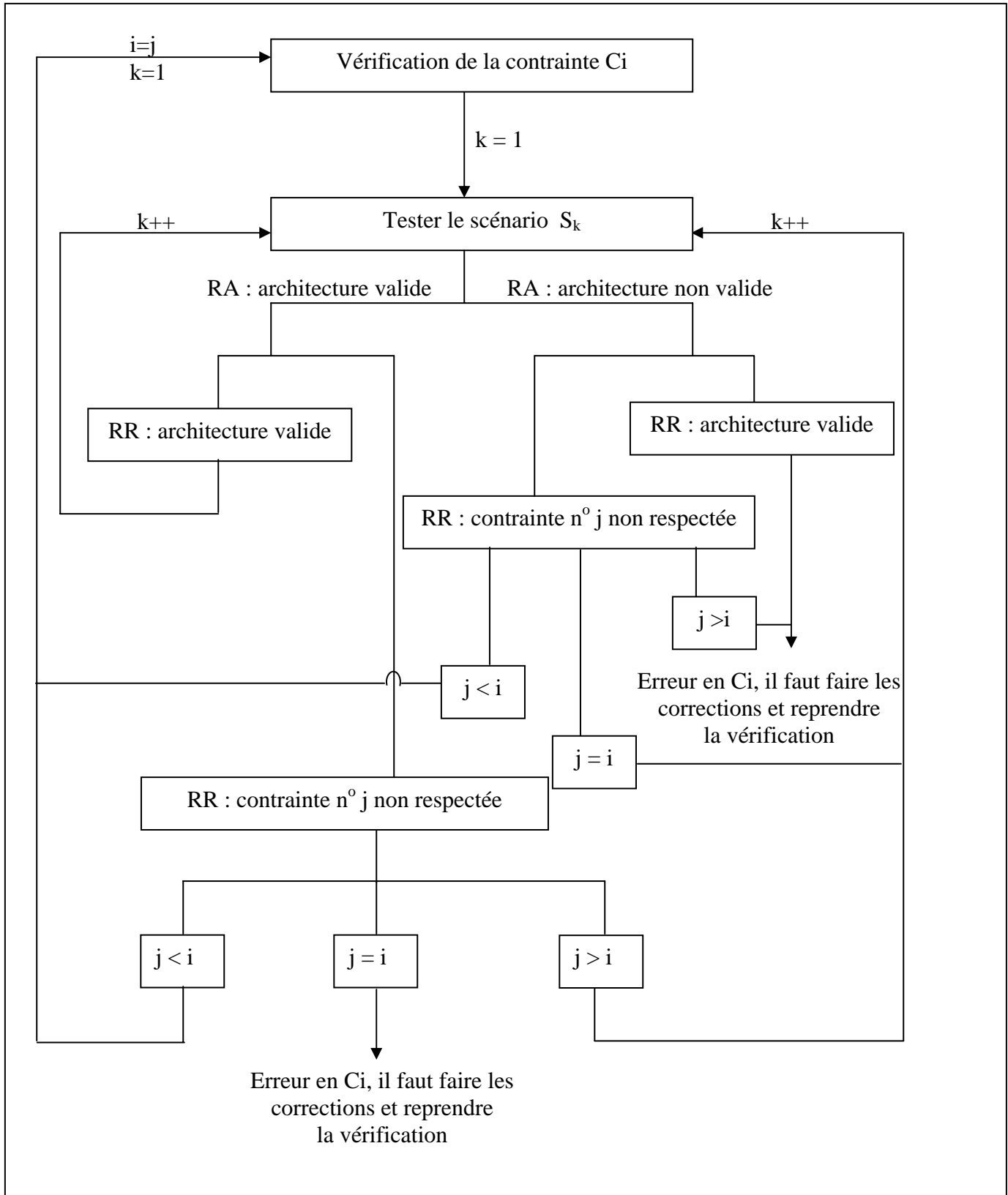


Figure 2.10. *Le processus de vérification des contraintes*

Supposons que nous allons tester la contrainte C_i , nous devons donc maintenir toutes les contraintes C_j avec $i \neq j$ respectées.

Si C_i est une contrainte de cardinalité alors nous n'avons qu'à tester les scénarii correspondants.

Sinon, C_i est de la forme *pour tout* (respectivement *il existe*, *il existe un seul*) composant de type X tel que la condition C . Nous devons tout d'abord vérifier la condition : nous allons procéder de manière récursive jusqu'à avoir les expressions élémentaires dont nous avons déterminé les cas de test. Ensuite, nous passons au test du quantificateur.

Nous signalons que lorsque nous testons les scénarii, nous devons trouver à chaque fois le résultat correspondant pour confirmer que la contrainte est bien spécifiée. Dans le cas où le résultat restitué par le simulateur diffère de celui prévu, nous sommes sûrs qu'il y a erreur et nous n'avons pas besoin de terminer le test des autres scénarii.

Il faut noter que lorsque le résultat prévu pour un scénario est "architecture non valide", il faut que le simulateur restitue le message "la contrainte numéro i est non respectée" (nous rappelons que nous sommes en train de tester la contrainte C_i).

Prenons l'exemple de ces deux contraintes

C_1 : au plus 4 composants de type X font des relations R

C_2 : un composant de type X est lié au plus à 3 composants de type Y par la relation R

Supposons que les contraintes réellement spécifiées sont :

C_1 : au plus 4 composants de type X font des relations R

C_2 : un composant de type X est lié au plus à 1 composant de type Y par la relation R

Nous commençons par tester C_1 , et la première configuration est la suivante :

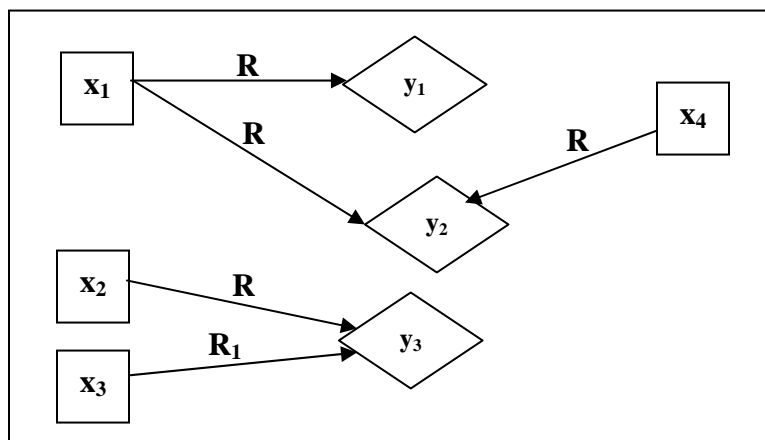


Figure 2.20: au plus 4 x_i font une relation R

Cette reconfiguration doit être valide par rapport au style architectural. Mais nous obtenons le message " la contrainte numéro 2 est non respectée ".

Pour éviter de telle situation, nous conseillons de suivre un ordre pour tester les contraintes non nécessairement celui de la spécification formelle.

En effet, nous commençons avec les contraintes concernant le nombre de liaisons pour un composant, ensuite les contraintes concernant le nombre de composants faisant des connexions, puis les contraintes concernant les types de relations pour un composant. Suivre cet ordre facilite la vérification et permet de gagner en nombre de tests.

Remarquons qu'au niveau du quantificateur, nous nous sommes intéressés qu'à un seul composant (c.à.d. pour tout x de type X).

Mais pour des contraintes telles que :

" Pour tout x, y de type X ; si x est lié à y par R alors y est lié à x par R "

Elles sont prises en compte puisque lorsque nous vérifions la contrainte pour x elle le sera automatiquement pour y car x et y sont symétriques dans la contrainte.

Aussi, dans le cas de :

" Pour tout x, y de type X ; pour tout z de type Y ; si x est lié à z par R et y est lié à z par R alors $x = y$ "

Nous sommes dans le cas de la contrainte " pour tout y de type Y ; y est l'image d'au plus un composant de type X ".

2.6 Conclusion

Dans ce chapitre, nous avons décrit la démarche que nous proposons pour tester la complétude et la conformité d'une spécification formelle d'une architecture logicielle par rapport aux besoins du client. Notre démarche se base sur la validation de certaines configurations architecturales par rapport au style architectural spécifié en Z et suivant une approche bien définie. Nous utilisons un simulateur de test d'architectures logicielles couplé avec l'outil $Z/EVES$. Ainsi, la partie formelle est cachée lors de la vérification de la complétude et la conformité, ce qui rend notre démarche exploitable même par un utilisateur non expert en formel.

Simulateur de test d'architectures logicielles

3.1 Introduction

La simulation des spécifications formelles joue un rôle important dans la validation des besoins. En effet, c'est une solution qui couvre le côté négatif majeur des spécifications formelles à savoir la difficulté de compréhension. Ainsi, l'utilisateur du simulateur vérifie ses besoins d'une manière familière tout en profitant des avantages des spécifications formelles.

Dans ce chapitre, nous présentons dans la première section le simulateur, ensuite, nous donnons sa conception et nous décrivons son implémentation dans les sections 3 et 4. Dans la dernière section, nous donnons l'intérêt de transformer le simulateur en un plug-in Eclipse.

3.2 Présentation du simulateur

Nous présentons notre simulateur (figure 3.1) comme une application générique de test des architectures logicielles par rapport à leur spécification formelle selon l'approche présentée dans le chapitre précédent.

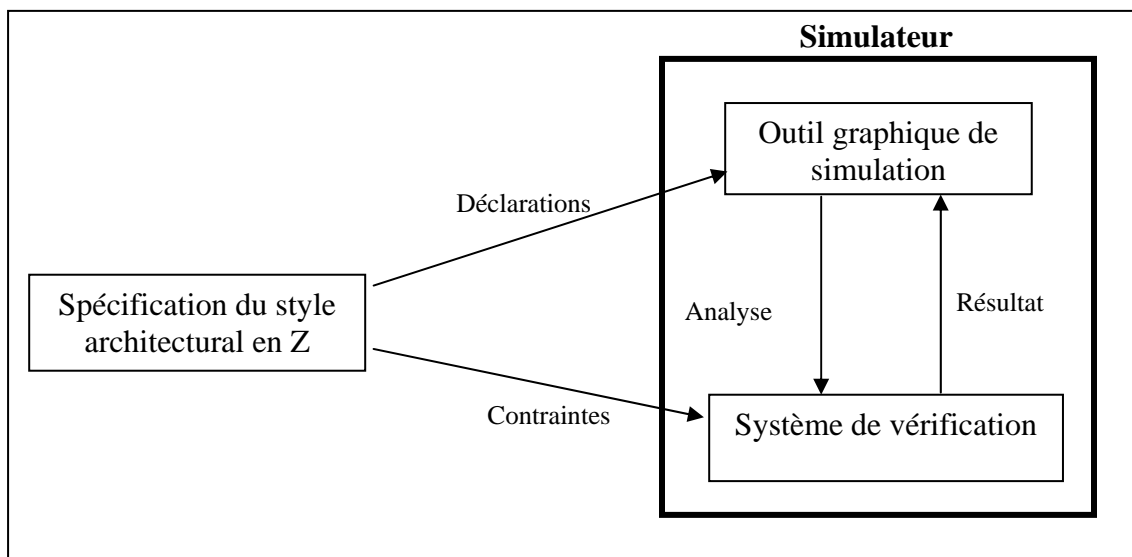


Figure 3.1. Architecture de l'application.

Il s'agit d'un outil graphique de simulation permettant de dessiner une architecture en termes de composants et connexions en cohérence avec les conventions de la notation UML 2.0. Cet outil permet, après la saisie de l'architecture, de lancer un processus de vérification basé sur la description formelle du style architectural.

Le système de vérification analyse l'architecture dessinée, teste si elle respecte toutes les contraintes spécifiées et restitue à l'outil graphique un résultat indiquant si la configuration saisie est conforme au style architectural spécifié.

Nous illustrons dans la figure 3.2 les différents scénarii possibles de fonctionnement de notre simulateur. Son exploitation passe par les étapes suivantes :

- L'utilisateur démarre le simulateur.
- Il choisit la spécification formelle des architectures logicielles à tester.
- Le simulateur analyse la spécification, extrait de la partie déclarative du schéma de style les différents composants et leurs types ainsi que les différentes connexions et leurs ensembles de départ et d'arrivée.

De même, il extrait les contraintes spécifiées dans la partie déclarative du schéma de style.

- Le simulateur prépare la boîte d'outils : il présente la liste des composants et celle des connexions.
- L'utilisateur saisit la configuration architecturale désirée : afin de représenter graphiquement, selon la notation d'UML 2.0, un composant, il appuie sur le bouton *composant* dans la boîte d'outils, clique dans la zone de dessin et choisit le type du composant désiré. Les mêmes étapes sont répétées pour représenter une connexion.

Notons que nous avons conçu le simulateur d'une manière donnant toute la liberté à l'utilisateur lors de la saisie de la configuration architecturale. Toutefois, nous signalons que le simulateur n'ajoute une connexion que si les types des composants sont ceux des ensembles de départ et d'arrivée de la connexion.

- L'utilisateur teste la configuration saisie.
- Le simulateur analyse la configuration saisie, teste si elle respecte toutes les contraintes et restitue le résultat de ce test à l'utilisateur.
- Suite au résultat de test et selon la démarche que nous avons définie dans le chapitre précédent, l'utilisateur teste d'autres configurations ou bien fait des corrections dans la spécification

formelle. Pour prendre en compte les nouvelles modifications, l'utilisateur doit choisir la commande *mettre-à-jour* du menu *Spécification*.

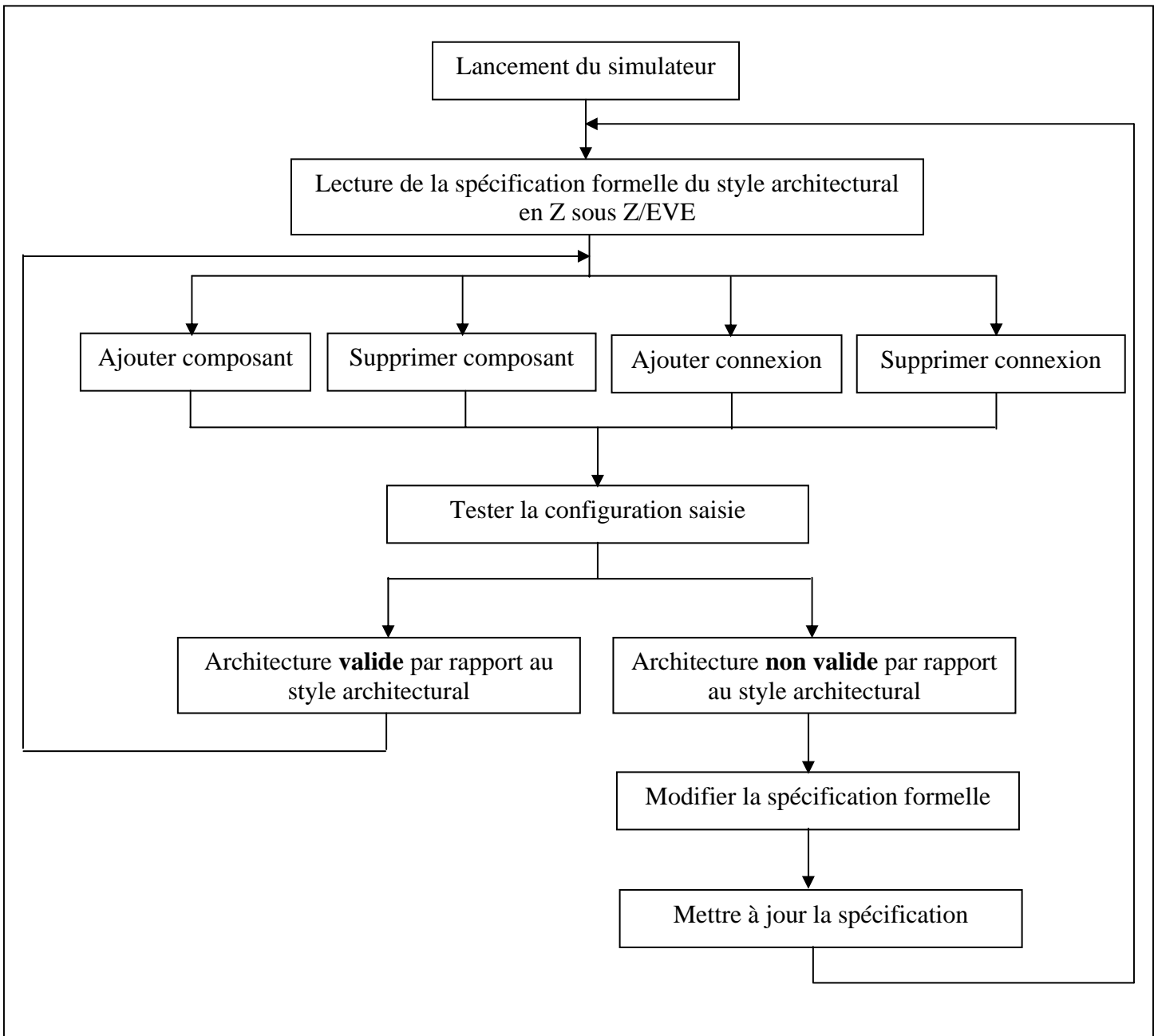


Figure 3.2. *Le scénario du fonctionnement du simulateur*

3.3 Conception du simulateur

Selon l'architecture de notre simulateur présentée ci-dessus et abstraction faite de l'implémentation, le problème majeur à résoudre consiste à : étant donné un graphe G qui représente une configuration d'une architecture et une description formelle S du style architectural, la question à laquelle il faut répondre par oui ou non est : " G est-il une instance de S ?".

En fait, G est un ensemble d'objets graphiques alors que S est écrit en langage Z . Pour résoudre ce problème, nous proposons de représenter un graphe G textuellement par un fichier XML dont la structure est donnée par la figure suivante.

```
- <Style>
- <composants>
  <typecomp1>nomcomp1</typecomp1>
  <typecomp1>nomcomp2</typecomp1>
  <typecomp2>nomcomp3</typecomp2>
  ...
  <typecomp_n>nomcomp m</typecomp_n>
</composants>
- <connexions>
- <typeconn1>
  <comp1>nomcomp1</comp1>
  <comp2>nomcomp3</comp2>
</typeconn1>
- <typeconn1>
  <comp1>nomcomp2</comp1>
  <comp2>nomcomp1</comp2>
</typeconn1>
  ...
- <typeconn_j>
  <comp1>nomcomp_n</comp1>
  <comp2>nomcomp_m</comp2>
</typeconn_j>
</connexions>
</Style>
```

Figure 3.4. Structure d'un fichier XML représentant une architecture.

En effet, une architecture est un ensemble de composants et de connexions typés. Un composant est défini par un type et un nom alors qu'une connexion a un type, un composant de départ et un composant d'arrivée. Un tel fichier XML nous permet d'identifier les ensembles des composants

et les ensembles des connexions ainsi que leurs types. Ces deux ensembles finis constituent un environnement qui va servir pour évaluer les formules spécifiant le style architectural.

Reste maintenant à interpréter les formules logiques du langage Z. Dans ce travail, nous considérons la partie du langage Z qui concerne la logique de prédicat du premier ordre et la théorie des ensembles sans tenir compte des relations et des fonctions.

Par formule du langage Z, on désigne un mot qui peut être généré par la grammaire décrite dans la suite :

$$\begin{aligned} \langle \text{Formule} \rangle ::= & \forall \langle \text{variable} \rangle : \langle \text{Ensemble} \rangle \bullet \langle \text{Formule} \rangle \mid \\ & \exists \langle \text{variable} \rangle : \langle \text{Ensemble} \rangle \bullet \langle \text{Formule} \rangle \mid \\ & \exists_1 \langle \text{variable} \rangle : \langle \text{Ensemble} \rangle \bullet \langle \text{Formule} \rangle \mid \langle \text{exp_logique} \rangle \end{aligned}$$

$$\begin{aligned} \langle \text{exp_logique} \rangle ::= & (\langle \text{exp_logique} \rangle) \mid \langle \text{exp_logique} \rangle \langle \text{op_log} \rangle \langle \text{exp_logique} \rangle \mid \\ & \langle \text{comp} \rangle \langle \text{op_app} \rangle \langle \text{Ensemble} \rangle \mid \langle \text{paire} \rangle \langle \text{op_app} \rangle \langle \text{Relation} \rangle \mid \\ & \langle \text{Ensemble} \rangle \langle \text{op_ens} \rangle \langle \text{Ensemble} \rangle \mid \langle \text{exp_arith} \rangle \langle \text{op_comp} \rangle \langle \text{exp_arith} \rangle \end{aligned}$$

$$\langle \text{op_log} \rangle ::= \neg \mid \wedge \mid \vee \mid \Rightarrow \mid \Leftrightarrow$$

$$\langle \text{op_comp} \rangle ::= < \mid > \mid \leq \mid \geq \mid =$$

$$\langle \text{op_app} \rangle ::= \in \mid \notin$$

$$\langle \text{op_ens} \rangle ::= \cup \mid \cap \mid \subset \mid \not\subset \mid \subseteq \mid = \mid \neq$$

$$\begin{aligned} \langle \text{exp_arith} \rangle ::= & (\langle \text{exp_arith} \rangle) \mid \langle \text{exp_arith} \rangle \langle \text{op_arith} \rangle \langle \text{exp_arith} \rangle \mid \\ & \# \langle \text{Ensemble} \rangle \mid \# \langle \text{Relation} \rangle \mid \langle \text{constante_num} \rangle \end{aligned}$$

$$\langle \text{op_arith} \rangle ::= * \mid / \mid + \mid -$$

$$\langle \text{constante_num} \rangle ::= \{ \langle \text{digit} \rangle \}^+ \mid - \{ \langle \text{digit} \rangle \}^+$$

$$\langle \text{digit} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$\langle \text{Ensemble} \rangle ::= \mathbf{ran} (\langle \text{Relation} \rangle) \mid \mathbf{dom} (\langle \text{Relation} \rangle) \mid \mathbf{image_rel} (\langle \text{Ensemble} \rangle) \mid$$

$$\{ \langle \text{comp} \rangle \}^+ \mid \emptyset$$

$$\begin{aligned} \langle \text{Relation} \rangle ::= & \mathbf{soustraction_dom} (\langle \text{Ensemble} \rangle, \langle \text{Relation} \rangle) \mid \\ & \mathbf{restriction_dom} (\langle \text{Ensemble} \rangle, \langle \text{Relation} \rangle) \mid \\ & \mathbf{soustraction_image} (\langle \text{Ensemble} \rangle, \langle \text{Relation} \rangle) \mid \\ & \mathbf{restriction_image} (\langle \text{Ensemble} \rangle, \langle \text{Relation} \rangle) \mid \{ \langle \text{paire} \rangle \}^+ \end{aligned}$$

Avec :

– $\langle \text{variable} \rangle$: une variable qui prend ses valeurs dans un ensemble de composants.

- $\langle comp \rangle$: représente un composant
- $\langle paire \rangle$: représente deux composants connectés par une relation.

Pour la définition de dom , ran , $image_rel$, $soustraction_dom$, $restriction_dom$, $soustraction_image$ et $restriction_image$, nous utilisons la notation Z .

En effet, nous considérons deux $\langle Ensemble \rangle X, Y$ et une $\langle Relation \rangle R$ tel que $R : X \leftrightarrow Y$.

Nous avons :

$$\checkmark \text{ dom } \mathbf{R} = \{x : \mathbf{X}, y : \mathbf{Y} \mid x \leftrightarrow y \in \mathbf{R} \bullet x \}$$

$$\checkmark \text{ ran } \mathbf{R} = \{x : \mathbf{X}, y : \mathbf{Y} \mid x \leftrightarrow y \in \mathbf{R} \bullet y \}$$

Nous considérons deux ensembles \mathbf{A} et \mathbf{B} , nous définirons

- ✓ la restriction de domaine $restriction_dom$ par

$$\mathbf{A} < \mathbf{R} = \{x : \mathbf{X}, y : \mathbf{Y} \mid x \leftrightarrow y \in \mathbf{R} \wedge x \in \mathbf{A} \bullet x \alpha y \}$$

- ✓ la restriction d'image $restriction_image$ par

$$\mathbf{R} > \mathbf{B} = \{x : \mathbf{X}, y : \mathbf{Y} \mid x \leftrightarrow y \in \mathbf{R} \wedge y \in \mathbf{B} \bullet x \alpha y \}$$

- ✓ La soustraction de domaine $soustraction_dom$ par

$$\mathbf{A} \triangleleft \mathbf{R} = \{x : \mathbf{X}, y : \mathbf{Y} \mid x \leftrightarrow y \in \mathbf{R} \wedge x \notin \mathbf{A} \bullet x \alpha y \}$$

- ✓ La soustraction d'image $soustraction_image$ par

$$\mathbf{R} \triangleright \mathbf{B} = \{x : \mathbf{X}, y : \mathbf{Y} \mid x \leftrightarrow y \in \mathbf{R} \wedge y \notin \mathbf{B} \bullet x \alpha y \}$$

- ✓ L'image relationnelle $image_rel$

$$\mathbf{R}(|\mathbf{A}|) = \mathbf{ran}(\mathbf{A} < \mathbf{R}) = \{x : \mathbf{X}, y : \mathbf{Y} \mid x \leftrightarrow y \in \mathbf{R} \wedge x \in \mathbf{A} \bullet y \}$$

Soit F une formule du langage Z et soit σ un environnement d'exécution. Ainsi, le problème se réduit à évaluer la formule F dans cet environnement. Cela revient donc à affecter aux variables de la formule les valeurs des ensembles et, ensuite, itérer l'évaluation de l'expression logique ($\langle exp_logique \rangle$) dans le cas de l'opérateur universel \forall pour tous les éléments de l'ensemble ou de trouver un élément qui satisfait la proposition dans le cas d'un opérateur existentiel \exists .

Pour l'évaluation de l'expression logique, il faut respecter l'ordre de priorité de ses opérateurs.

Ces derniers sont présentés dans un ordre de priorité décroissant :

- les parenthèses
- restriction de domaine, restriction d'image, soustraction de domaine et soustraction d'image
- le domaine et l'image d'une relation

- l'intersection et l'union de deux ensembles
- l'inclusion de deux ensembles
- l'opérateur \neq
- la cardinalité d'un ensemble: $\#$
- les opérateurs arithmétiques: $*$, $/$
- les opérateurs arithmétiques: $+$, $-$
- les opérateurs de comparaison: $<$, $>$, \leq , \geq , $=$
- l'appartenance d'un élément à un ensemble: \in , \notin
- la négation (non): \neg
- la conjonction (et): \wedge
- la disjonction (ou inclusif) : \vee
- l'implication (si, alors) : \Rightarrow
- l'équivalence (si et seulement si): \Leftrightarrow

Afin de respecter les priorités des opérateurs lors de l'évaluation d'une expression, nous proposons d'utiliser une méthode basée sur deux piles. Une pile *P1* dans laquelle nous stockons les valeurs et une deuxième *P2* pour stocker les opérateurs.

3.3.1 Diagrammes de classes préliminaires

Dans les diagrammes suivants, nous représentons des classes d'analyse (sans attributs ni méthodes). Nous avons choisi de représenter deux sous diagrammes pour assurer la lisibilité et la clarté de la figure. Le premier diagramme montre les classes qui constituent le processus de vérification, alors que le second présente les classes nécessaires au développement de l'interface graphique.

La conception du processus de vérification est basée sur une architecture qui décompose le problème de l'évaluation en modules (classes) où chacun est concerné par l'évaluation d'un type particulier de formules. S'ajoute à ces modules un module de lecture et prétraitement des formules Z ainsi que l'analyse et l'extraction des composants et des connexions à partir de la représentation XML.

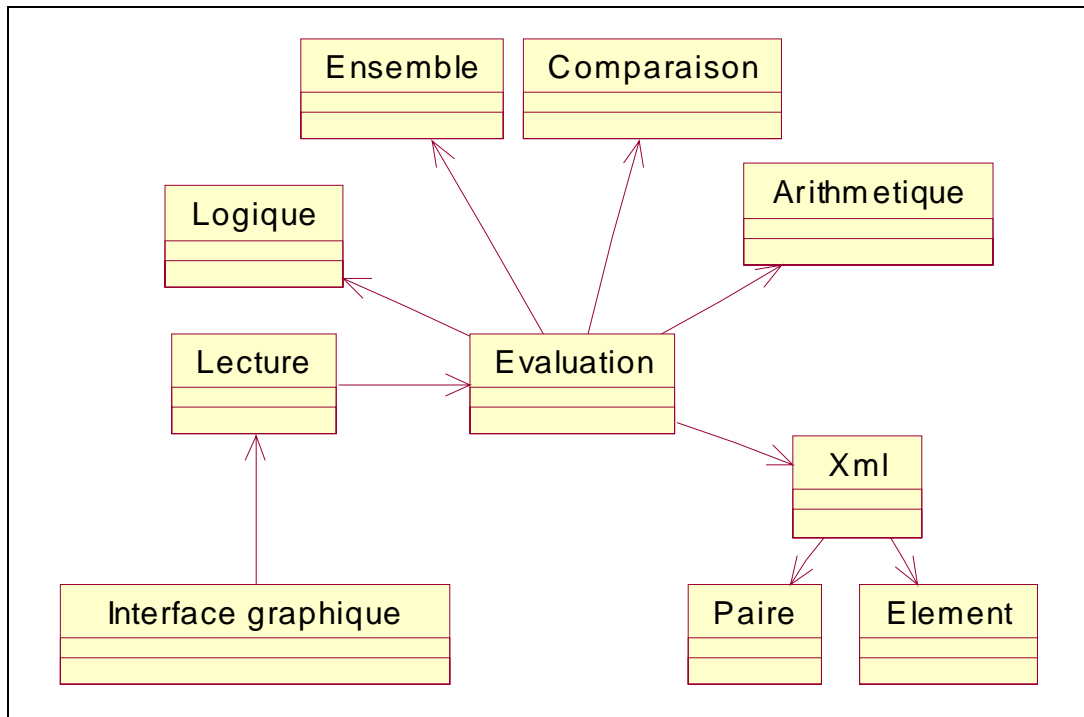


Figure 3.5. *Diagramme de classes préliminaires (première partie).*

Afin de concrétiser la partie graphique, nous avons conçu trois modules à savoir :

- un module de création de composants et d’édition de leurs types.
- Un module permettant d’ajouter des relations de différents types entre les composants.
- Un module qui permet de gérer les fonctionnalités assurées par la boîte d’outils et les commandes des différents menus.

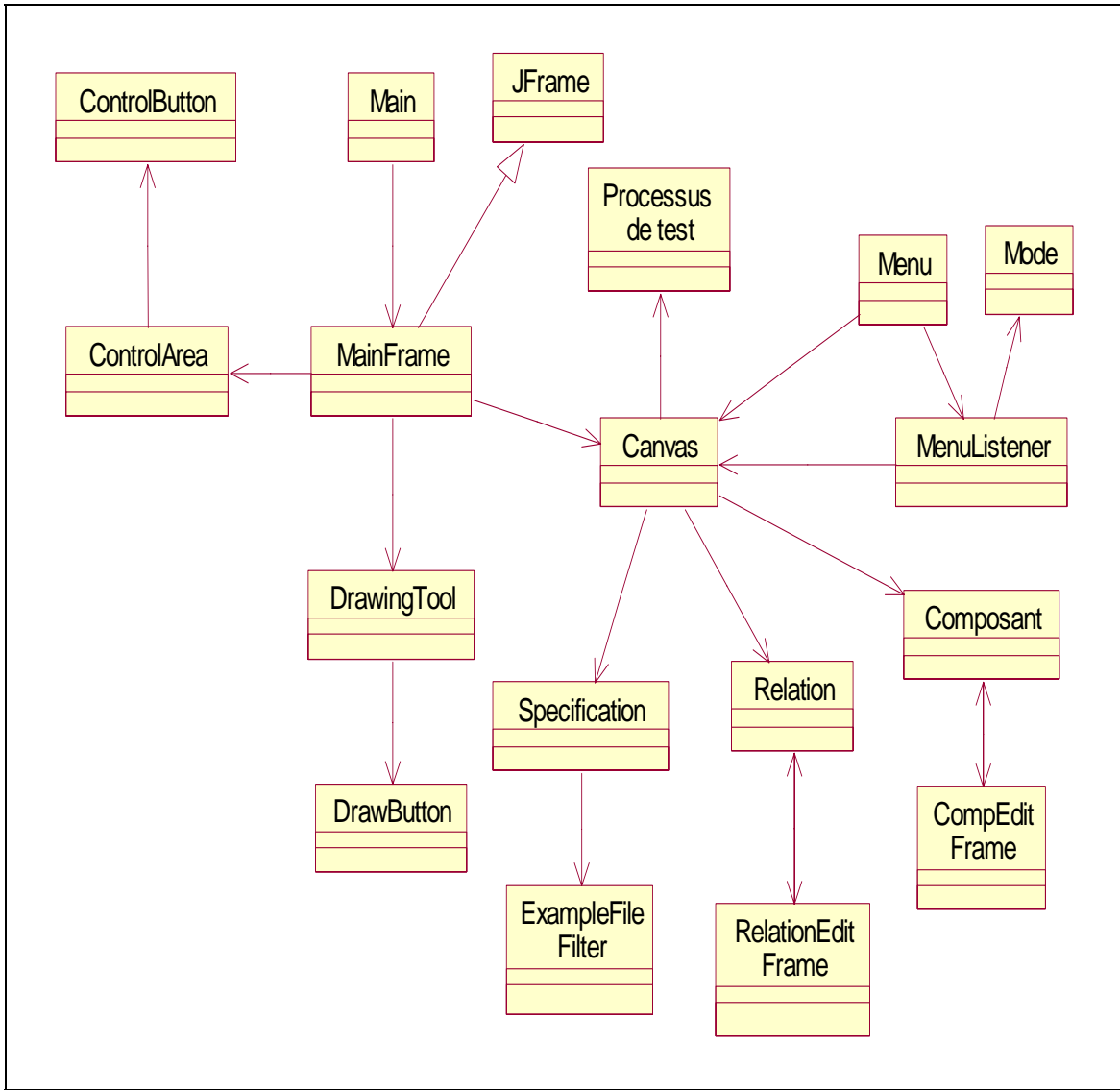


Figure 3.6. Diagramme de classes préliminaires (deuxième partie).

Une description détaillée des principales fonctionnalités offertes par ces classes est donnée dans le tableau suivant.

Tableau 3.1. *Tableau explicatif des diagrammes de classes*

Arithmétique	Cette classe implémente les différentes opérations arithmétiques.
Ensemble	Les méthodes de cette classe décrivent les opérations relatives à un ensemble (cardinalité, appartenance d'un élément, inclusion...).
Logique	Cette classe s'intéresse aux opérations logiques.
Comparaison	Cette classe implémente les opérations de comparaison.
Paire	Cette classe représente la structure d'une connexion.
Element	Un Elément est soit un type de composant soit un type de connexion.
XML	C'est la structure qui représente l'architecture dessinée par l'utilisateur.
Evaluation	C'est une classe dont les méthodes testent le respect d'une contrainte par une représentation textuelle.
Lecture	Cette classe est responsable de lire les contraintes décrites dans la spécification formelle, les analyser et ensuite les évaluer.
Main	Cette classe contient la méthode principale main.
MainFrame	Cette classe implémente l'interface principale et place tous les éléments à leurs endroits.
Canvas	Cette classe est responsable du dessin graphique.
ButtonListener	Cette classe est responsable d'ajout de composants et de connexions.
Composant	Cette classe décrit un composant.
CompEditFrame	La fenêtre d'édition des propriétés d'un composant.
ControlArea	Le panneau contenant tous les éléments de l'interface principale.
DrawButton	La classe qui implémente les différents boutons.
DrawingTool	La classe qui dessine tous les boutons dans le panneau ControlArea à l'aide de la classe DrawButton.
ExempleFileFilter	Elle est utilisée au moment de l'ouverture ou l'enregistrement d'un fichier pour filtrer les extensions.

Specification	Cette classe analyse le fichier Z/EVES contenant la spécification formelle pour en extraire les composants et les connexions.
Menu	Cette classe regroupe l'ensemble des menus de l'outil.
MenuListener	Cette classe implémente les commandes des menus.
ControlButton	Cette classe prépare le bouton avant de le placer dans un panneau.
Relation	Cette classe décrit une connexion.
RelationEditFrame	La fenêtre d'édition des propriétés d'une connexion.

Les relations qui existent entre les différentes classes représentent une instantiation. Les quatre premières classes décrites dans le tableau précédent ainsi que les classes *Evaluation* et *Lecture* contiennent seulement des méthodes statiques (eng. *static*).

Avant de représenter le modèle statique en détail en termes de diagrammes de classes, nous présentons, dans la section suivante, les diagrammes de séquences des principaux scénarii qui peuvent avoir lieu lors de l'exécution de notre application.

3.3.2 Diagrammes de séquences

Les diagrammes qui suivent décrivent les principales fonctionnalités de l'outil de test des architectures. Nous allons détailler les scénarios ajouter / supprimer un composant (respectivement une connexion) et tester une architecture logicielle.

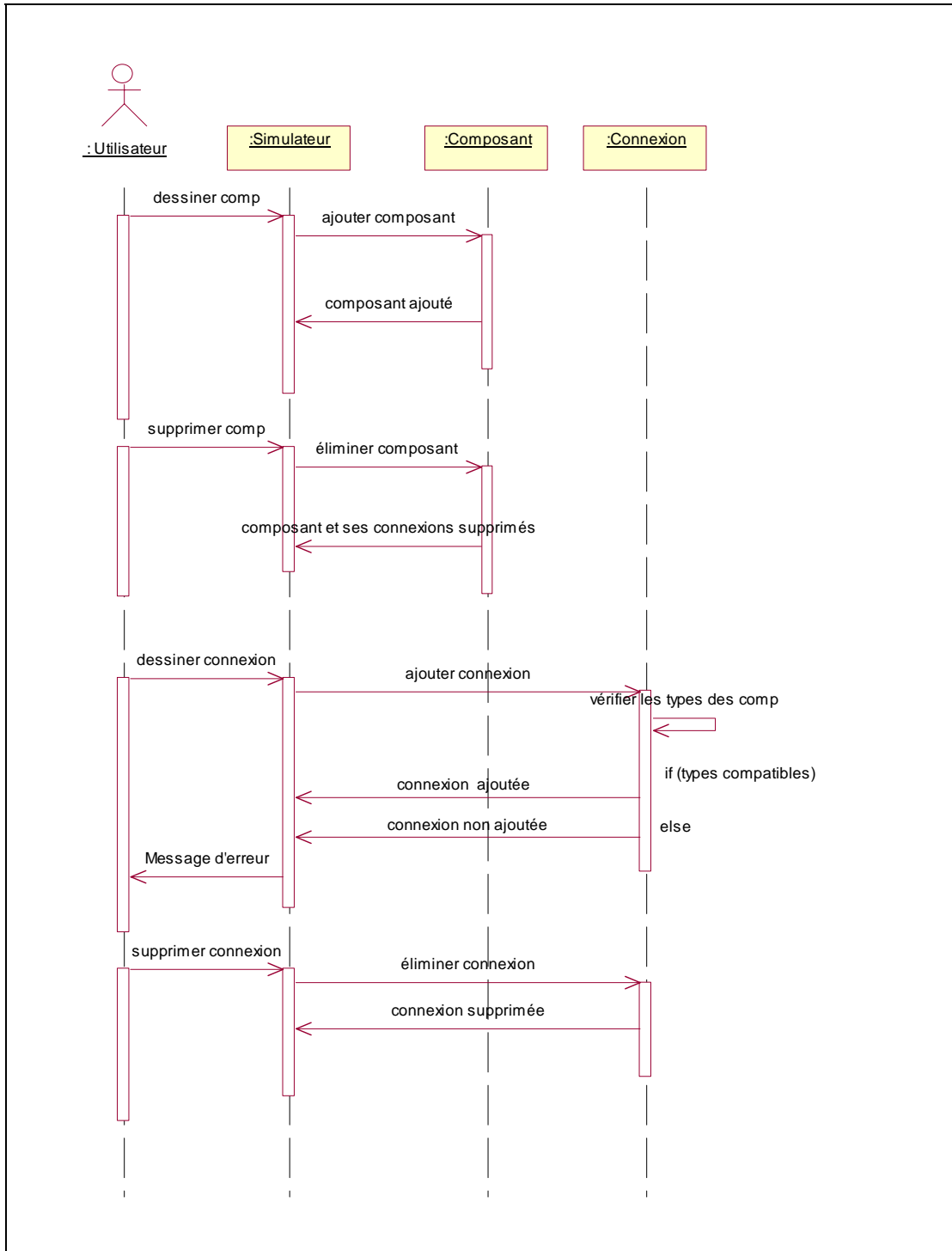


Figure 3.7. Diagramme de séquence : Ajouter (supprimer) un composant (respectivement une connexion).

L'utilisateur ajoute les composants désirés mais lors de la suppression d'un composant donné, le simulateur élimine automatiquement les connexions qui partent de ce composant ou qui lui arrivent.

Lors de l'ajout d'une connexion, le simulateur vérifie si les types des composants connectés sont ceux des ensembles de départ et d'arrivée de la connexion. En cas de non comptabilité, le simulateur restitue un message d'erreur.

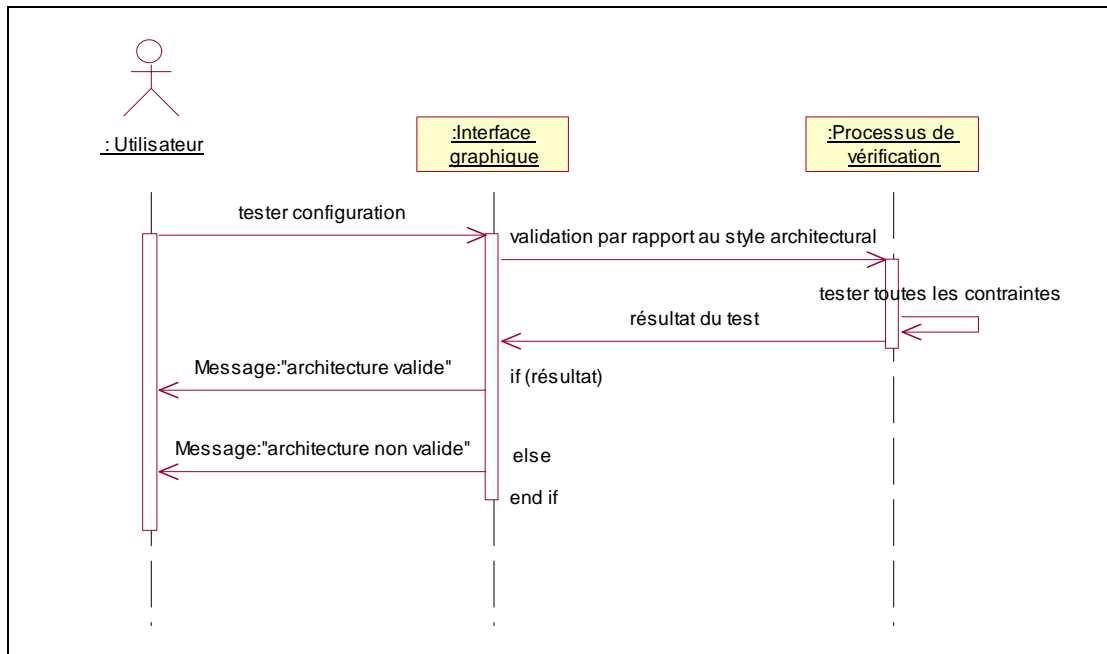


Figure 3.8. Diagramme de séquence : Tester une architecture logicielle.

Le test d'architecture est le but de ce travail et il nécessite l'appel du processus de vérification qui est responsable de tester toutes les contraintes relatives à la spécification formelle.

Pour réaliser cet outil avec toutes les commandes décrites dans les figures 3.7, 3.8, nous avons retenu les différentes classes décrites dans le diagramme du paragraphe suivant.

3.3.3 Diagrammes de classes de conception

L'architecture présentée par le diagramme de la figure 3.9 est inspirée de la structure des formules du langage Z considérées. En effet, pour faciliter et mieux structurer la procédure d'évaluation, nous avons réparti cette dernière en plusieurs classes :

- la classe *Lecture* pour extraire les contraintes architecturales du style spécifié. Chaque contrainte sera évaluée à part et le résultat de test sera la conjonction des résultats des évaluations des contraintes.
- la classe *Evaluation* responsable de l'évaluation d'une formule. Elle a deux méthodes : une pour faire les itérations nécessaires suivant la nature du quantificateur et l'autre pour évaluer l'expression logique.

– la classe *Arithmétique* regroupe les opérations arithmétiques (addition, soustraction, multiplication, division).

– la classe *Logique* fournit les méthodes relatives aux opérateurs logiques : not, and, or, implique, équivalent.

– la classe *Ensemble* implémente des méthodes relatives aux opérations qui s’appliquent à un ensemble (de composants ou de connexions) telles que :

- tester l’appartenance d’un composant à un ensemble de composants,
- donner la cardinalité d’un ensemble,
- donner l’ensemble des composants constituant le domaine d’une connexion,
- donner l’ensemble des composants constituant l’image d’une connexion,
- donner l’ensemble des paires de composants obtenus par soustraction de domaine (respectivement d’image),
- donner l’ensemble des paires de composants obtenus par restriction de domaine (respectivement d’image),
- donner l’ensemble des composants obtenus par image relationnelle,
- donner l’union de deux ensembles,
- donner l’intersection de deux ensembles,
- compare un ensemble avec le vide,
- tester l’inclusion de deux ensembles,

– la classe *Comparaison* pour les opérateurs de comparaison.

– la classe *XML* exploite le fichier *XML* relatif à l’architecture dessinée. Cette classe a deux attributs : deux vecteurs un pour les ensembles des composants et un pour les types de connexions. Les éléments de ces vecteurs sont de type *Element*.

– la classe *Element* représente un ensemble. Un ensemble a un nom et un vecteur d’éléments. Ces derniers peuvent être des composants ou des connexions représentées par des paires.

– la classe *Paire* désigne une connexion.

REMARQUE : dans le diagramme de la figure 3.9 nous nous contentons de représenter les classes qui forment le processus de test vue leur importance dans ce travail.

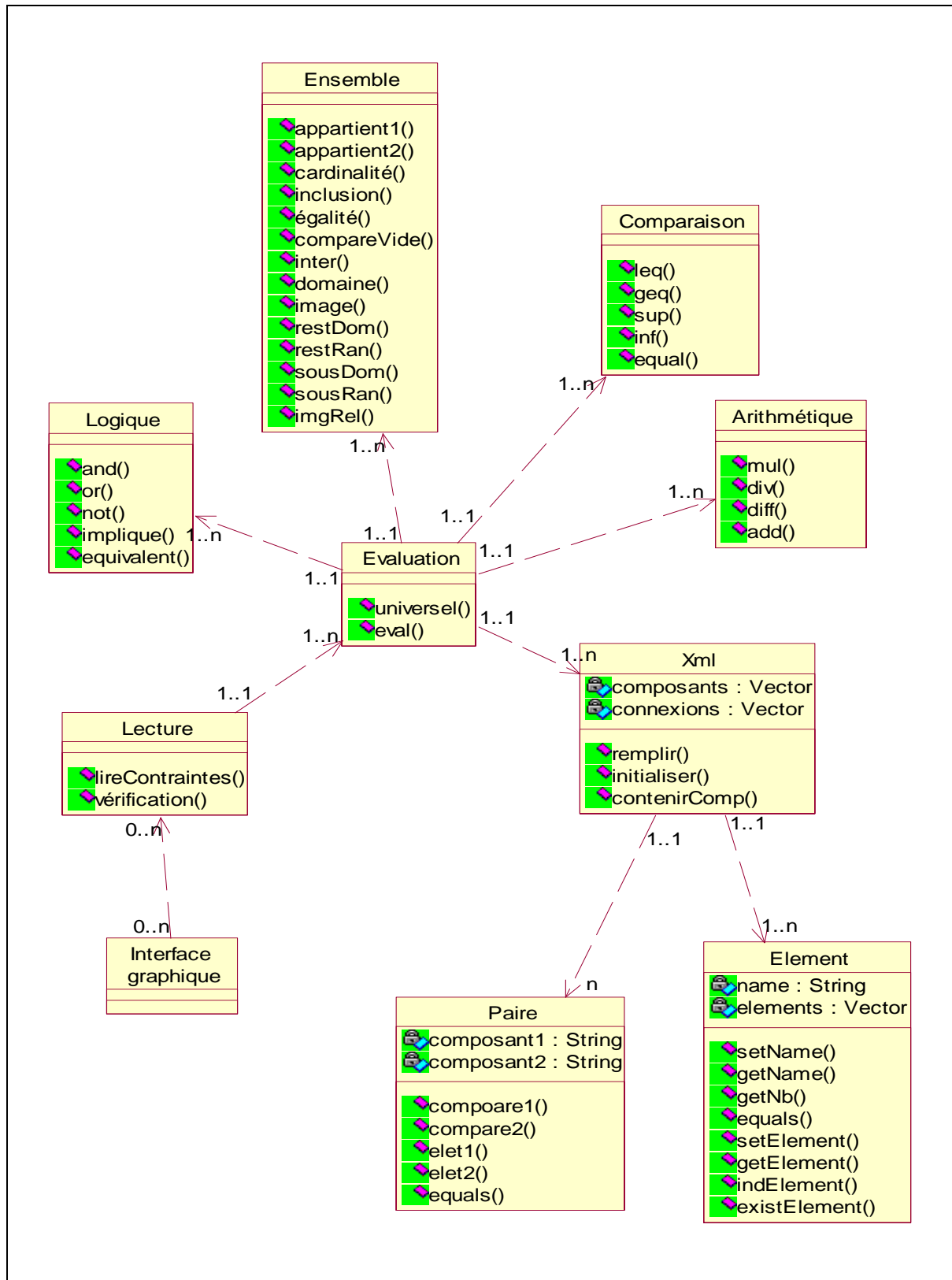


Figure 3.9. Diagramme de classes de conception.

3.4 Implémentation du simulateur

Le simulateur que nous souhaitons développer doit être générique. En d'autres termes, il est valable pour toute spécification formelle décrivant un style architectural selon la méthode adoptée. Cette propriété est la base de tous les choix d'implémentation que nous avons faite pour mettre en œuvre cet outil.

3.4.1 Langages et environnements

➤ **Le langage XML**

Avant que le processus de vérification teste la conformité de l'architecture dessinée par l'outil graphique par rapport au style spécifié, il y a une étape assez importante soit l'étape d'analyse de l'architecture. Cette étape est la base du test et elle doit traduire un ensemble d'objets graphiques en des données exploitables par le système de test. Comme solution, nous avons utilisé un fichier XML dans lequel nous stockons toutes les informations nécessaires à partir du graphe de l'architecture.

XML (*Extensible Markup Language*) est une technologie précieuse pour représenter et échanger les données. C'est un langage de balisages, ce qui aidera plus tard dans le processus de vérification pour le parcours des données.

➤ **L'outil Z/EVES**

Les styles architecturaux utilisés sont spécifiés en langage Z avec l'outil Z/EVES. Avec Z/EVES, l'utilisateur peut écrire, éditer et analyser des spécifications en Z. L'interface de Z/EVES (figure 3.10) supporte l'analyse par accroissement des spécifications et gère la synchronisation des analyses avec les modifications des spécifications.

Z/EVES utilise les techniques récentes de méthodes formelles, en intégrant une notation de spécification éminente avec une capacité élevée d'automatisation de déduction. Le système résultant supporte l'analyse des spécifications en Z de plusieurs manières. En d'autres termes, il permet :

- la vérification automatique de la syntaxe et des types,
- la vérification possible des domaines d'application des fonctions,
- la mise en évidence des pré-conditions à une opération,
- la démonstration de théorème sur la spécification,
- l'élaboration de tests sur la dynamique (enchaînement d'opérations de même niveau).

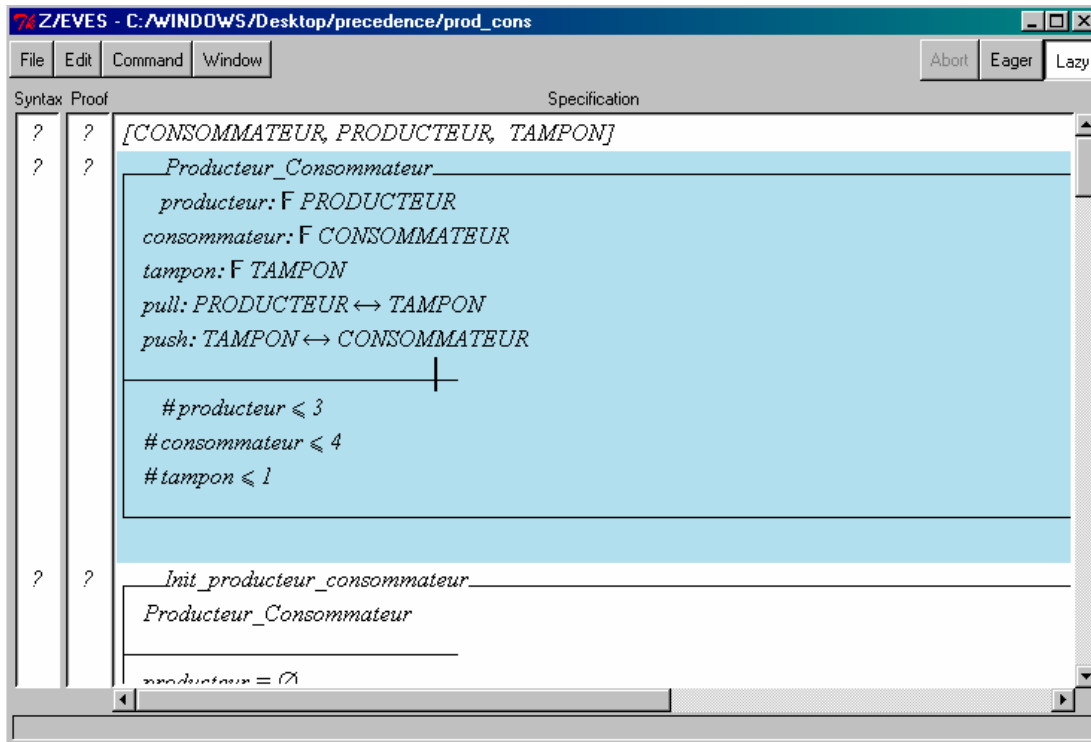


Figure 3.10. L'interface de Z/EVES.

Les styles architecturaux écrits et vérifiés avec Z/EVES seront stockés dans des fichiers dont l'extension est .zev. Comme un fichier XML, un fichier .zev est structuré en balises.

Cette caractéristique est très utile lors de l'extraction des composants et des connexions ainsi que lors de l'extraction des contraintes architecturales. En effet, le tableau suivant présente les différentes balises qui délimitent les parties utilisées lors de l'implémentation de cet outil.

Tableau 3.2. Certaines balises d'un fichier .zev et leur signification

<schema-box>	Cette balise indique le début du schéma.
<decl-part/>	À partir de cette balise commence la partie déclarative du schéma.
<ax-part/>	Cette balise indique la fin de la partie déclarative et le début de celle prédictive.
</schema-box>	Elle indique la fin du schéma.

3.4.2 Détails d'implémentation

La partie consistante du simulateur est le code assurant la vérification des contraintes spécifiées. En effet, lorsque l'utilisateur presse la commande *Test* du menu *Architecture*, un fichier XML est généré selon la structure présentée dans la section de conception.

Par la suite, nous considérons les contraintes architecturales extraites de la spécification ; nous manipulons donc un ensemble de formules $\{ F_1, F_2, \dots, F_n \}$.

Où F_i est une formule définie par la grammaire présentée précédemment dans ce chapitre.

Le résultat du test est la conjonction de l'évaluation (test) de toutes les formules. Il sera donné par l'algorithme suivant :

$$\text{Résultat} = \text{Test}(\text{Spécification}) = \bigwedge_{i=1}^n \text{Eval}(F_i)$$

Nous distinguons quatre cas :

- i. Premier cas : F_i est de la forme : $\forall x : \langle \text{ensemble} \rangle \bullet P_i$

L'évaluation d'une telle formule se fait selon l'algorithme suivant (écrit en langage algorithmique simplifié).

```
Eval (Fi) ::= {  
    boolean res = true ;  
    while ( (not_empty (<ensemble>)) and res)  
    {  
        Let e ∈ <ensemble>  
        res := res and Eval (Pi (e)) ;  
        <ensemble>:=<ensemble>\{e};  
    }  
    return res;  
}
```

- ii. Deuxième cas : F_i est de la forme : $\exists x : \langle \text{ensemble} \rangle \bullet P_i$

Dans ce cas, F_i sera évaluée selon l'algorithme suivant:

```
Eval (Fi) ::= {  
    boolean res = false ;  
    while ( (not_empty (<ensemble>)) and (non res))  
    {
```

```

Let e ∈ <ensemble>
res := Eval (Pi (e) );
<ensemble>:=<ensemble>\{e} ;
}
return res;
}

```

iii. Troisième cas F_i est de la forme : $\exists_1 x : \langle \text{ensemble} \rangle \bullet P_i$

L'évaluation de F_i est donnée par l'algorithme suivant :

```

Eval (Fi) ::= {
    boolean res = true ;
    int compteur = 0;
    while ( not_empty (<ensemble>))
    {
        Let e ∈ <ensemble>
        if ( Eval (Pi (e) ) ) compteur ++ ;
        <ensemble>:=<ensemble>\{e} ;
    }
    res= (compteur == 1);
    return res ;
}

```

iv. Le dernier cas : F_i s'écrit sous la forme d'une expression logique $\langle \text{Exp_Log} \rangle$

L'évaluation de F_i est donnée par celle de l'expression logique :

```

Eval (Fi) ::= Evaluation (<exp_Log>);

```

La procédure *Evaluation* est basée sur l'utilisation de deux piles $P1$ et $P2$: une pour les valeurs et l'autre pour les opérateurs. Il s'agit de lire l'expression logique et exploiter les deux piles de la manière suivante :

- une valeur est toujours stockée dans la pile $P1$;
- en cas d'un opérateur op :
 - si la pile $P2$ est vide, nous stockons op et avançons dans la lecture de l'expression ;

- si la pile $P2$ est non vide, il faut alors tester la priorité des opérateurs. Si l'opérateur à la tête de $P2$ est plus prioritaire, alors il sera exécuté avec les deux opérandes se trouvant à la tête de $P1$ en cas d'un opérateur binaire ou avec la tête de $P1$ s'il est unaire. Dans le cas contraire (op est plus prioritaire), nous stockons op dans $P2$ et avançons dans la lecture en empilant les opérateurs et les valeurs jusqu' à la présence d'un opérateur $op1$ moins prioritaire que op . A ce moment, nous exécutons les opérateurs empilés dans $P2$ jusqu'à op et nous reprenons cette procédure à partir de $op1$. Suivant l'opérateur, il y aura appel à une méthode qui prend en paramètres les opérandes en tête de la pile $P1$.

Prenons l'exemple de l'expression

$\langle \text{exp-logique} \rangle = \# (\langle \text{Relation} \rangle (\langle \text{ensemble} \rangle)) < n$

Nous avons alors :

$\text{Evaluation} (\langle \text{exp-logique} \rangle) ::=$

$\text{Comparaison.inf} (\text{Ensemble.card} (\text{Ensemble.imgRel} (\langle \text{ensemble} \rangle, \langle \text{Relation} \rangle)), n) ;$

Avec :

$\text{Ensemble.imgRel} (\langle \text{Ensemble} \rangle, \langle \text{Relation} \rangle) ::=$

```
{
    Element E= new Element() ;
    E=Ensemble.image(Ensemble.restDom(<ensemble>,<Relation>)) ;
    return E ;
}
```

$\text{Ensemble.image} (\langle \text{Relation} \rangle) ::=$

```
{
    Element E=new Element() ;
    while ( not_empty (<Relation>))
    {
        Let p ∈ <Relation> ;
        // p est une paire de composants (elet1, elet2)
        E=E U {p.elet2};
        <Relation>=<Relation>\{p} ;
    }
}
```

```
    return E ;  
}
```

Ensemble.restDom(*<Ensemble>*,*<Relation>*) ::=

```
{  
    Element E=new Element() ;  
    while ( not_empty (<Relation>))  
    {  
        Let p ∈ <Relation> ;  
        // p est une paire de composants (elet1, elet2)  
        if (Ensemble.appartenance (p.elet1,<Ensemble>)  
        then E=E U {p};  
        <Relation>=<Relation>\{p} ;  
    }  
    return E;  
}
```

Ensemble.appartenance(x, *<Ensemble>*) ::=

```
{  
    boolean found = false ;  
    while ( non found and not_empty (<Ensemble>))  
    {  
        Let e ∈ <Ensemble> ;  
        found = (e == x) ;  
        <Ensemble>=<Ensemble>\{e} ;  
    }  
    return found ;  
}
```

Ensemble.card (*<Ensemble>*) ::=

```
{  
    int compteur = 0 ;
```

```

while ( not_empty (<Ensemble>))
{
    Let e ∈ <Ensemble> ;
    compteur ++ ;
    <Ensemble>=<Ensemble>\{e} ;
}
return compteur ;
}

```

Comparaison.inf (n_1, n_2) ::=

```

{
    return  $n_1 < n_2$ ;
}

```

Avec n_1, n_2 sont deux entiers.

3.4.3 Présentation de l'interface du simulateur

La figure 3.11 présente l'interface du simulateur. Elle contient des menus, une boîte d'outils et une zone de dessin.

Nous proposons quatre menus :

– Le menu *fichier* contient cinq commandes :

- ✓ *Nouveau* : permet de commencer une nouvelle configuration.
- ✓ *Ouvrir* : permet de récupérer un ancien graphe d'architecture.
- ✓ *Enregistrer* et *Enregistrer sous* : permettent de sauvegarder le graphe d'architecture dessiné.
- ✓ *Quitter* : permet de fermer l'environnement.

– Le menu *Spécification* contient deux commandes :

- ✓ *Lire* : lire une spécification
- ✓ *Mettre-à-jour* : après modification de la spécification, cette commande permet d'extraire les composants les connexions et les contraintes de nouveau sans relancer le simulateur.

– Le menu *Architecture* contient la commande *tester* qui permet de lancer le processus de vérification.

– Le menu *A propos* lance une fenêtre de dialogue à partir de la commande *Env de test*.

La boîte d'outils contient cinq boutons. Ces boutons sont respectivement *un composant*, *une connexion*, *supprimer*, *déplacer* et *éditer propriétés*.

Au chargement de l'interface, les boutons *composant* et *connexion* sont *désactivés*. Il faut choisir de lire une spécification pour les activer (ce qui correspond à l'extraction des types des composants et des connexions). Lorsque l'utilisateur dessine un composant (respectivement une connexion) une fenêtre se lance pour choisir les propriétés du composant (respectivement de la connexion).

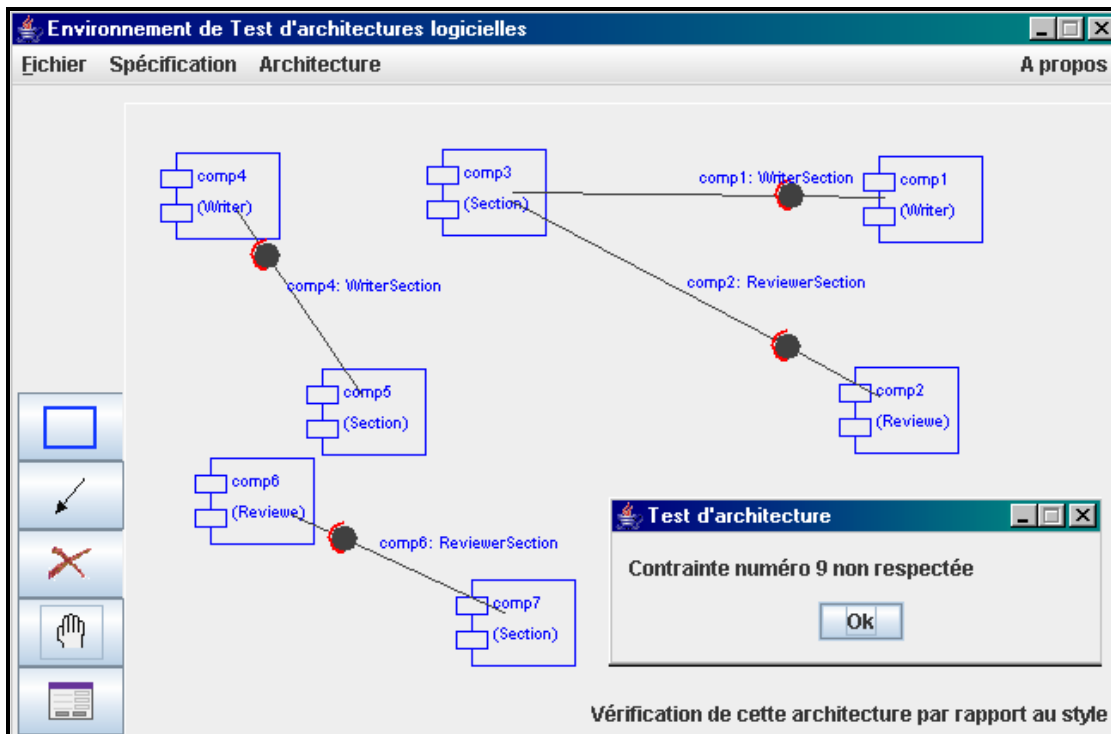


Figure 3.11. *Simulateur de test d'architectures logicielles*

3.5 Plug-in pour l'IDE Eclipse

IBM, Borland, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft et Webgain lancent, en novembre 2001, la première version de la plateforme Eclipse, comme un projet code libre [27].

Eclipse n'est pas une simple plateforme pour développer des applications en Java, ceci est une de ses nombreuses facettes. Les créateurs d'Eclipse ont voulu faire d'elle une plateforme

multifonctionnelle. De nos jours, l'utilité d'une telle plateforme est immense puisqu'elle nous permet de travailler avec une multitude de fichiers différents sans changer d'outils à chaque fois. Les plug-ins sont la base d'Eclipse. L'architecture de plug-in d'Eclipse est comparable à un jeu de construction Lego, elle est aussi transformable que ce dernier. Ainsi nous pouvons ajouter tout simplement une fonctionnalité à la plateforme en posant une brique, ou plus précisément un plug-in.

Si un utilisateur veut ajouter son propre outil à la plateforme, il lui suffit de créer un plug-in qui étend une des fonctionnalités d'Eclipse. Car, presque tous les plug-ins qui forment la plateforme offrent des points d'extension où le plug-in de l'utilisateur peut venir se greffer.

Un plug-in est bien évidemment codé en Java. Celui-ci se présente sous la forme d'une archive, Java Archive (JAR), qui contient des fichiers Java, des images, d'autres archives...

Dans ce contexte et pour contribuer au travail d'équipe au sein de notre unité de recherche ReDCAD, nous avons transformé ce simulateur en un plug-in pour étendre tout un travail [14] concernant les architectures logicielles dynamiques.

3.6 Conclusion

Nous avons conçu et implémenté un simulateur générique, basé sur une technique formelle utilisant la notation Z et couplé avec l'outil Z/EVES. Toutes fois nous rappelons que le simulateur est conçu pour les spécifications formelles des architectures logicielles suivant l'approche que nous avons définie. Pour cela, lors de l'implémentation, nous avons structuré les classes d'une manière que l'extension du simulateur soit facile.

Etude de cas "l'édition coopérative"

4.1 Introduction

Nous avons présenté dans les deux chapitres précédents la démarche de validation de spécifications et le simulateur de test d'architectures logicielles, nous allons, dans la suite, illustrer ce travail par une étude de cas intitulée " l'édition coopérative ". Nous considérons la spécification formelle qui a été élaborée par Slim KALLEL au sein de l'unité de recherche ReDCAD.

4.2 L'édition coopérative

4.2.1 Le cahier de charges

Le système d'édition coopérative " *Collaborative Authoring System* " est une application ayant trois types de composants : *Writer*, *Reviewer* et *Section*. Deux types de relations lient ces composants : la relation *WriterSection* entre un composant de type *Writer* et un composant de type *Section* et la relation *ReviewerSection* pour connecter un *Reviewer* et une *Section*.

Le système d'édition coopérative doit respecter les contraintes suivantes :

- Le système doit contenir au maximum 4 *reviewers*, 5 *writers* et 4 *sections*.
- Au plus deux connexions de type *WriterSection*.
- Au plus trois connexions de type *WriterSection* et *ReviewerSection*.
- Tout *writer* rédige au plus une *section*.
- Tout *reviewer* révisé au plus une *section*.
- Toute *section* est écrite par un seul *writer*.
- Toute *section* est révisée par un seul *reviewer*.
- Toute *section* ne peut pas être rédigée et révisée en même temps

4.2.2 Le schéma de style

Nous avons introduit quelques erreurs dans la spécification initiale et nous avons obtenu comme résultat le schéma de style suivant :

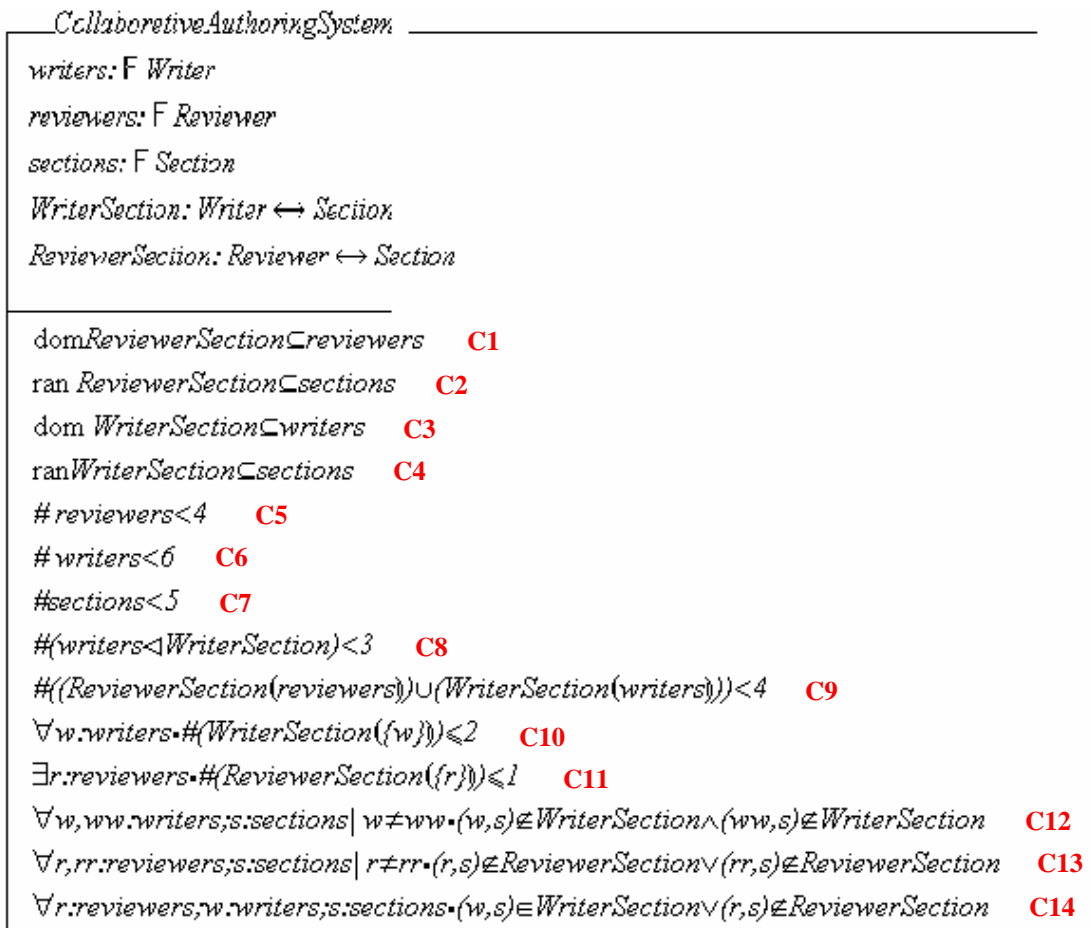


Figure 4.1 : *Le schéma de style de l'application*

Nous avons modifié la spécification de sorte que :

- ✓ le système a **au plus 3 reviewers** (la contrainte C5) ;
- ✓ tout *writer* rédige **au plus deux sections** (la contrainte C10) ;
- ✓ **il existe** un *reviewer* qui révisé au plus une *section* (la contrainte C11) ;
- ✓ toute *section* est rédigée par **tous** les *writers* (la contrainte C12) ;
- ✓ toute *section* peut être **rédigée et révisée à la fois** (la contrainte C14) ;

4.2.3 La validation de la spécification

Nous allons présenter dans la suite la vérification de quelques contraintes de la spécification tout en suivant les scénarios proposés par la démarche.

❖ **Contrainte numéro 5** : le système doit contenir au maximum 4 *reviewers*

➤ **Scénario 1** : une configuration architecturale avec 3 *reviewers*. Le résultat doit être ” *architecture valide* ”.

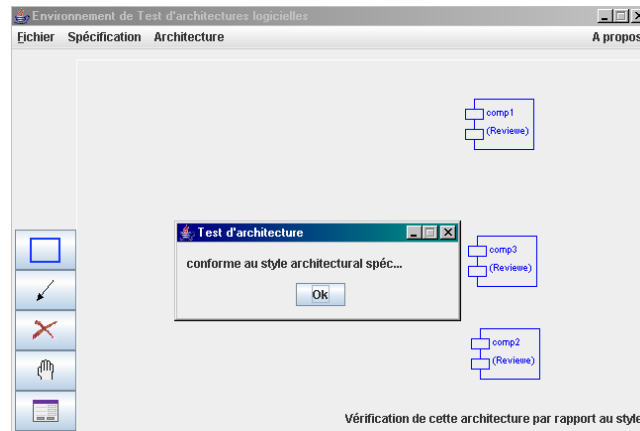


Figure 4.2 : *Contrainte numéro 5, scénario 1*

Le résultat restitué est conforme à celui attendu, nous pouvons donc tester le scénario suivant.

➤ **Scénario 2** : une configuration architecturale avec 4 *reviewers*. Le résultat doit être ” *architecture valide* ”.

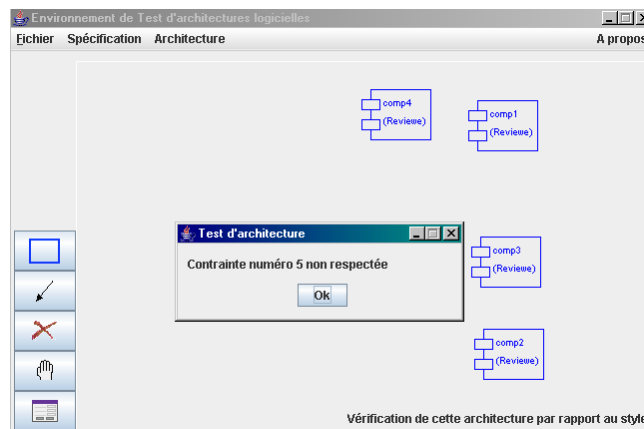


Figure 4.3 : *Contrainte numéro 5, scénario 2*

L'architecture est non conforme au style architectural, nous concluons qu'il y a *une erreur dans la spécification de la contrainte numéro 5*.

❖ **Contrainte numéro 8** : Au plus deux connexions de type *WriterSection*

➤ **Scénario 1** : une configuration architecturale avec deux connexions de type *WriterSection*. Le résultat doit être " *architecture valide* ".

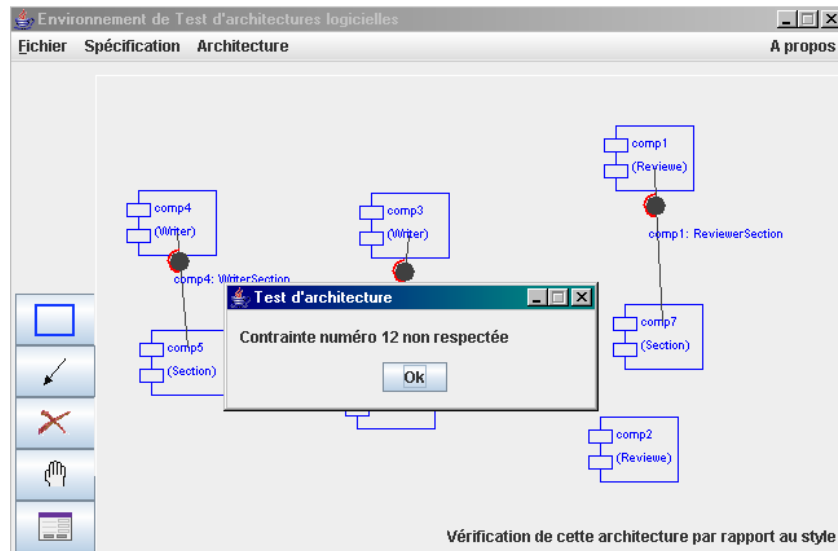


Figure 4.4 : *Contrainte numéro 8, scénario 1*

Le simulateur indique que la contrainte numéro 12 est non respectée. Nous sommes en train de vérifier la contrainte numéro 8, donc ceci implique que la configuration respecte la contrainte numéro 8 spécifiée. Nous pouvons tester le scénario suivant.

➤ **Scénario 2** : une configuration architecturale avec trois connexions de type *WriterSection*. Le résultat doit être " *architecture non valide* ".

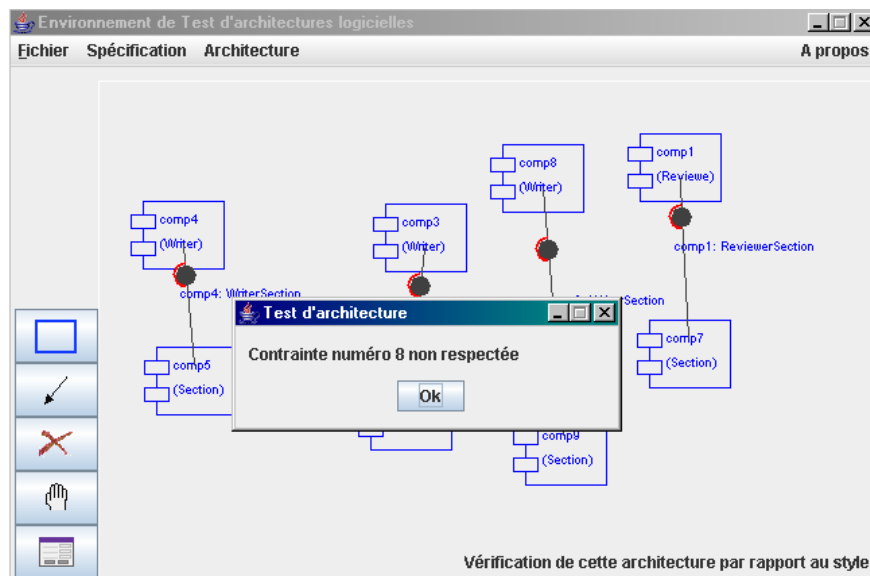


Figure 4.5 : *Contrainte numéro 8, scénario 2*

Le résultat restitué est conforme à celui attendu, nous pouvons donc tester le scénario suivant.

- **Scénario 3** : une configuration architecturale avec une connexion de type *WriterSection*. Le résultat doit être ” *architecture valide* ”.

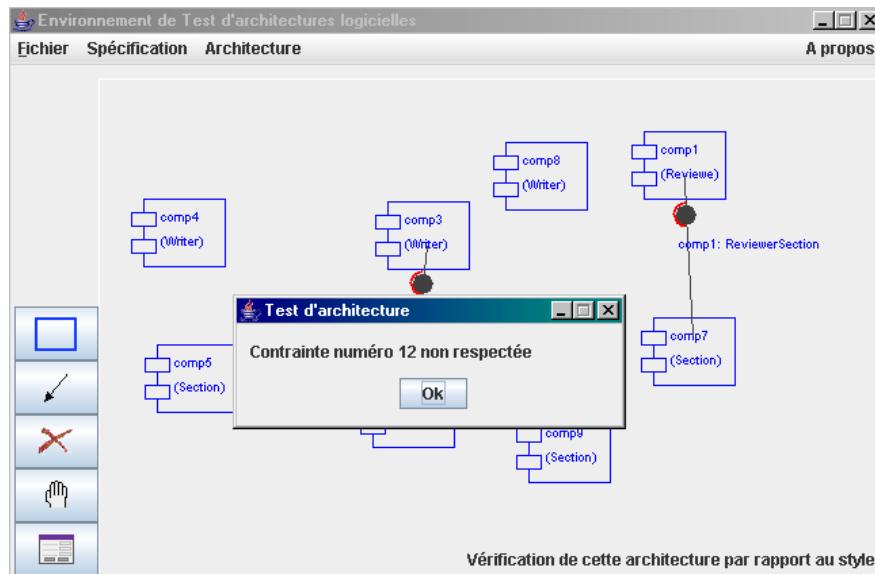


Figure 4.6 : *Contrainte numéro 8, scénario 3*

Le résultat restitué implique que la configuration respecte la contrainte numéro 8, donc nous avons eu les résultats attendus pour les trois scénarios. Nous sommes certains que **la contrainte numéro 8 est bien spécifiée**.

- ❖ **Contrainte numéro 10** : Un *writer* rédige au plus une *section*.

- **Scénario 1** : une configuration architecturale où un *writer* est lié à une *section*. Le résultat doit être ” *architecture valide* ”.

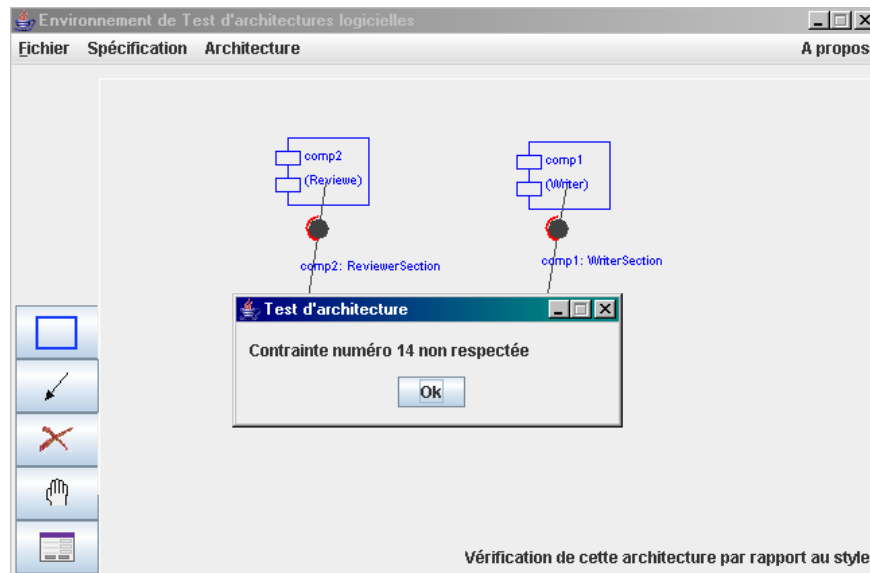


Figure 4.7 : *Contrainte numéro 10, scénario 1*

Le résultat restitué implique que la configuration respecte la contrainte numéro 10, nous pouvons donc tester le scénario suivant.

- **Scénario 2 :** une configuration architecturale où un *writer* est lié à deux *section*. Le résultat doit être " *architecture non valide* ".

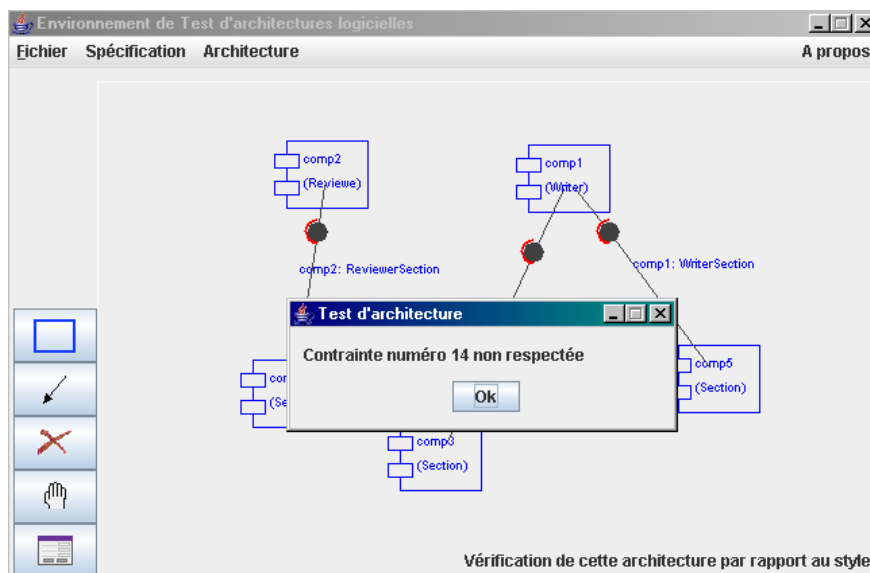


Figure 4.8 : *Contrainte numéro 10, scénario 2*

Le résultat restitué implique que la configuration respecte la contrainte numéro 10, ce qui contredit le résultat attendu. Il y a donc ***une erreur au niveau de la contrainte numéro 10.***

❖ **Contrainte numéro 11** : Un *reviewer* révisé au plus une *section*.

➤ **Scénario 1** : une configuration architecturale où un *reviewer* est lié à une *section*.

Le résultat doit être ” *architecture valide* ”.

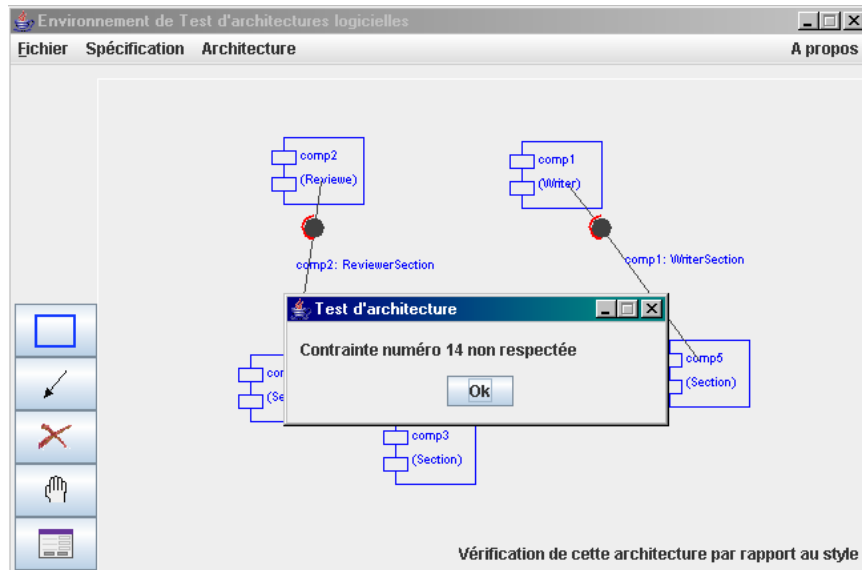


Figure 4.9 : *Contrainte numéro 11, scénario 1*

Le résultat restitué implique que la configuration respecte la contrainte numéro 11, nous pouvons tester le scénario suivant.

➤ **Scénario 2** : une configuration architecturale où un *reviewer* est lié à deux *section*. Le résultat doit être ” *architecture non valide* ”.

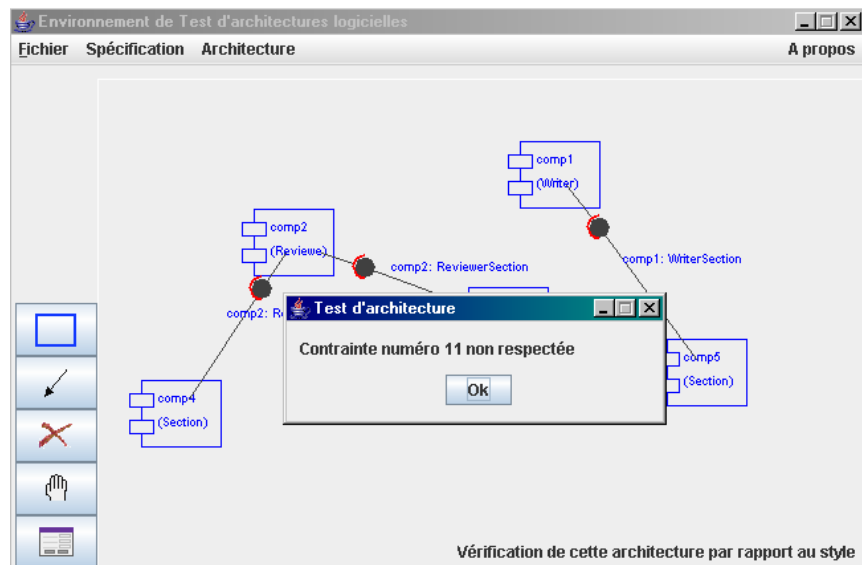


Figure 4.10 : *Contrainte numéro 11, scénario 2*

La configuration est non valide, nous pouvons passer au scénario suivant.

- **Scénario 3** : une configuration architecturale où un *reviewer* n'est pas lié à une *section*. Le résultat doit être " *architecture valide* ".

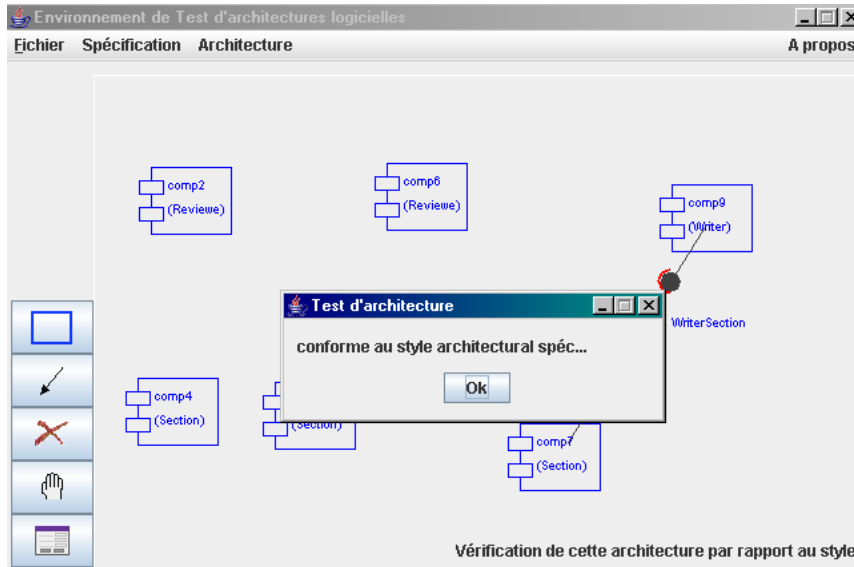


Figure 4.11 : Contrainte numéro 11, scénario 3

Le résultat restitué est conforme à celui attendu, nous pouvons donc passer à la vérification du quantificateur.

- **Scénario 4** : une configuration architecturale où un *reviewer* est lié à une *section* et un autre lié à deux *section*. Le résultat doit être " *architecture non valide* ".

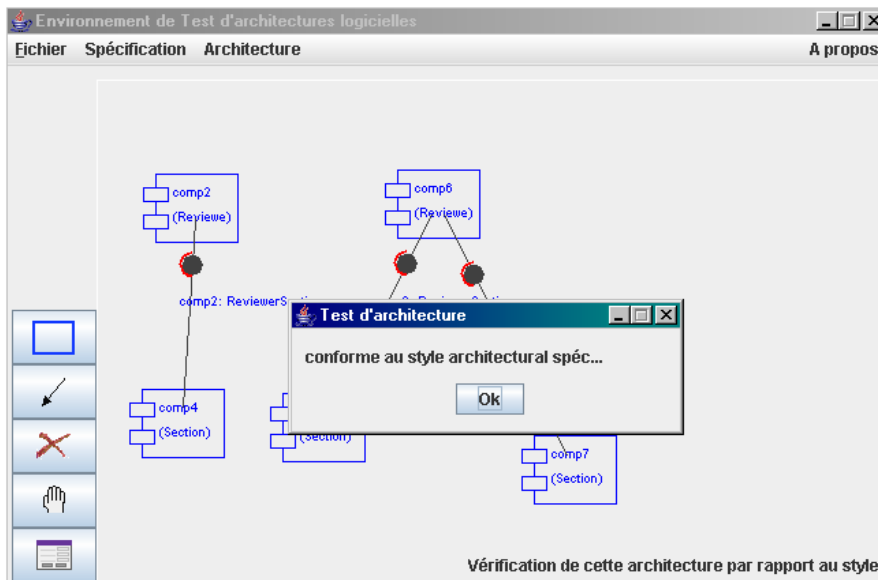


Figure 4.12 : Contrainte numéro 11, scénario 4

Le résultat restitué est différent de celui prévu, *nous concluons que le quantificateur est erroné.*

❖ **Contrainte numéro 12** : Une *section* est écrite par un seul *writer*.

Nous allons voir cette contrainte de la façon suivante : deux *writer* différents $w, w1$, w est lié à la *section* ou (exclusive) $w1$ est lié à la *section*.

➤ **Scénario 1** : une configuration architecturale où une section est rédigée par deux *writer*. Le résultat doit être " *architecture non valide* ".

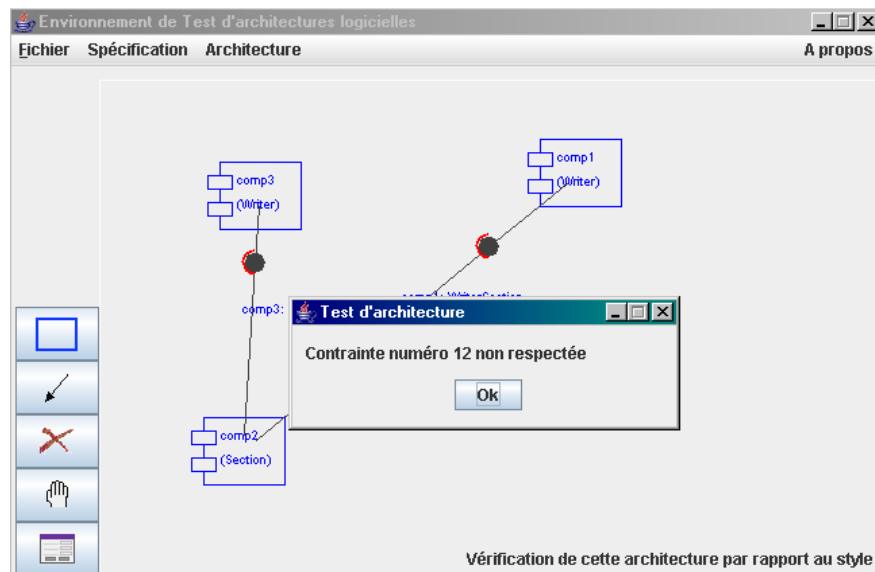


Figure 4.13 : *Contrainte numéro 12, scénario 1*

Le résultat restitué est conforme à celui attendu, nous pouvons donc tester le scénario suivant.

➤ **Scénario 2** : une configuration architecturale où nous maintenons une connexion *WriterSection* et nous supprimons la deuxième. Le résultat doit être " *architecture valide* ".

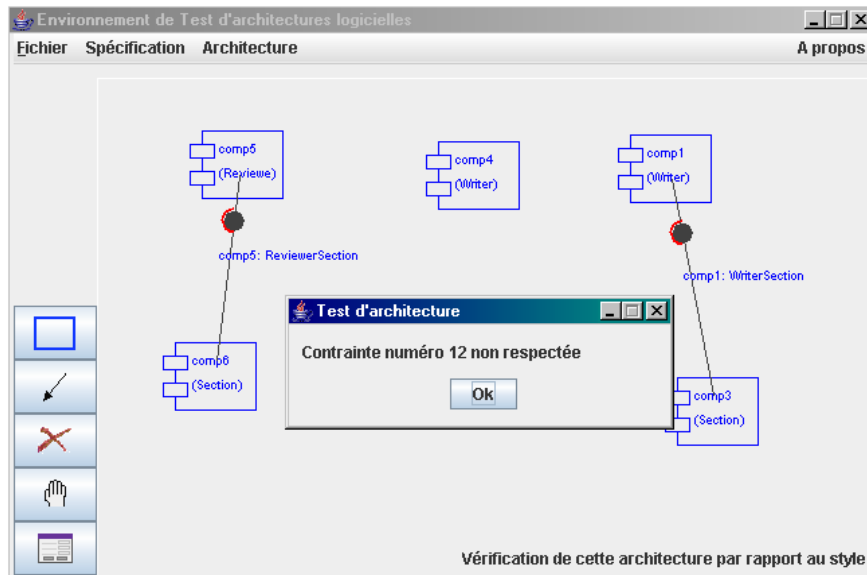


Figure 4.14 : Contrainte numéro 12, scénario 2

La configuration architecturale est non valide, il y a donc *une erreur au niveau de l'opérateur logique*.

- ❖ **Contrainte numéro 14** : Une *section* ne peut pas être rédigée et révisée en même temps.
 - **Scénario 1** : une configuration architecturale où une *section* est rédigée et révisée en même temps. Le résultat doit être " *architecture non valide* ".

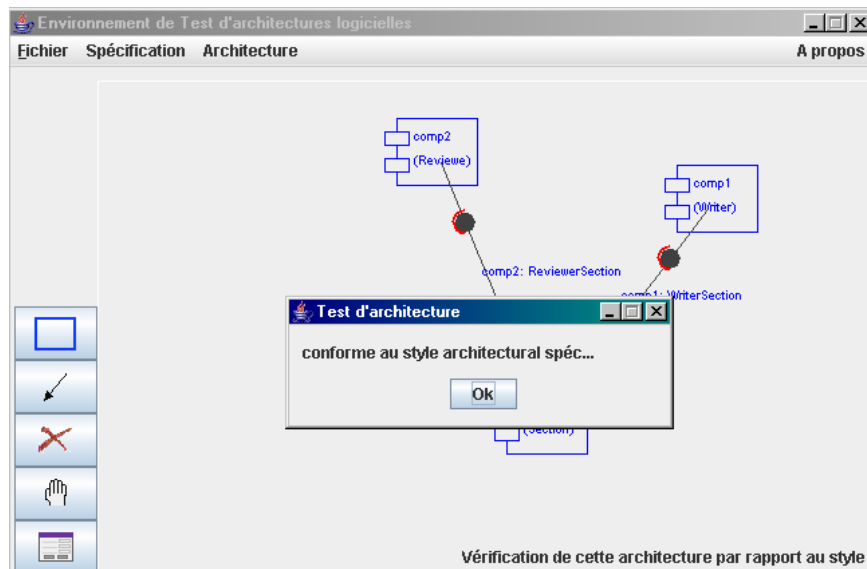


Figure 4.15 : Contrainte numéro 14, scénario 1

Le résultat est architecture valide, il y a donc *une erreur dans la contrainte numéro 14*.

4.2.4 La spécification après correction

Une fois, nous avons terminé les tests des scénarios et les corrections des erreurs détectées, nous obtenons la spécification du schéma de style suivant.

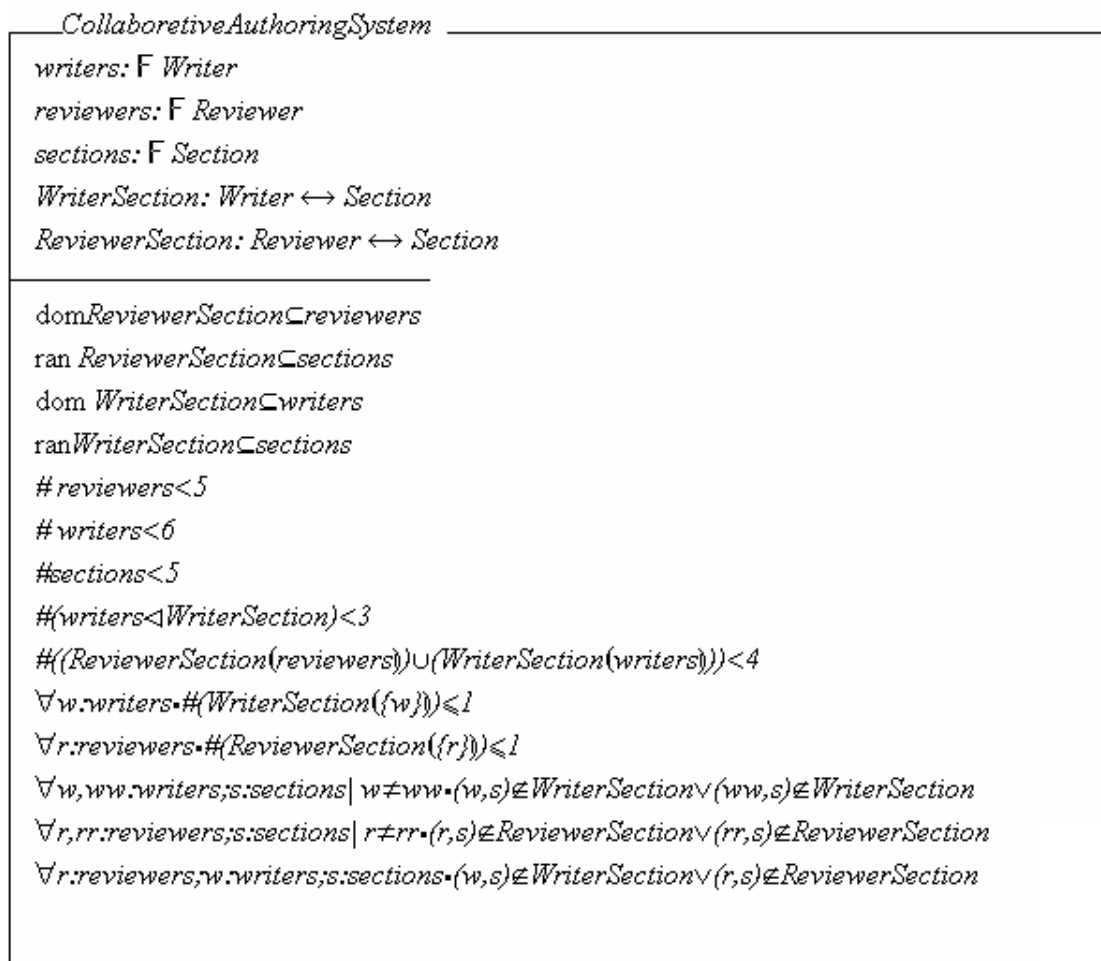


Figure 4.16 : la spécification du schéma de style rectifiée

Pour s'assurer de plus, la configuration de la figure 4.17 doit être valide et son test a prouvé cette supposition.

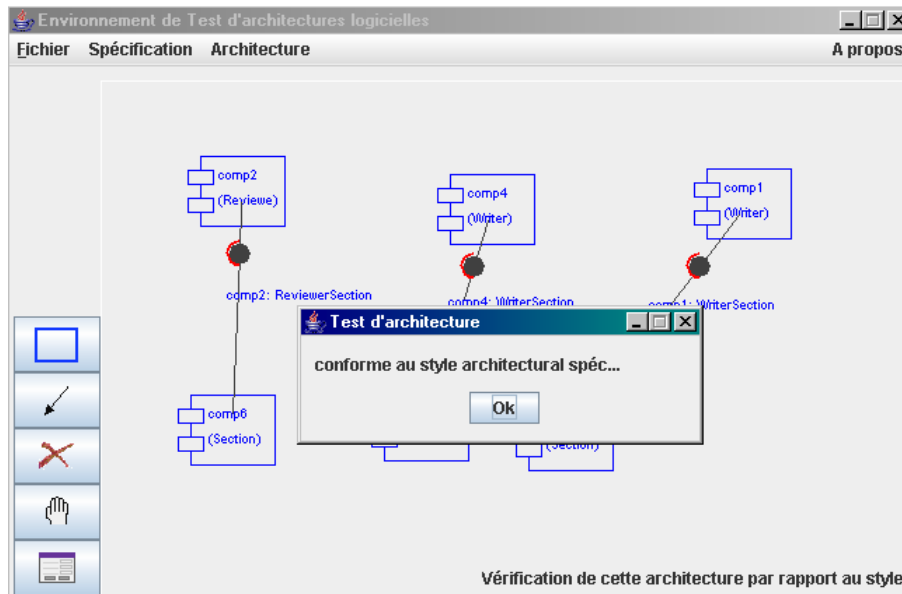


Figure 4.17: *une configuration architecturale type*

4.3 Conclusion

Dans ce chapitre, nous avons essayé de donner une idée sur la manière d'exploitation de la démarche proposée. Nous ne pouvons pas montrer le test de toutes les contraintes vue le grand nombre des figures nécessaires.

Conclusion générale

Dans ce travail de mastère, nous avons essayé de résoudre le problème de la complétude et conformité des spécifications formelles des architectures logicielles. Nous avons défini une démarche se basant sur la validation de scénarios par rapport au style architectural spécifié.

En effet et suivant les types de contraintes exprimées en langage naturel, nous avons proposé des configurations architecturales à tester tout en déterminant les résultats des tests qui doivent être restitués. Pour mettre en œuvre notre démarche, nous avons implémenté un simulateur de test d'architectures logicielles. C'est un outil générique, se basant sur une technique formelle et couplé avec l'outil Z/EVES.

Nous sommes convaincus que la validation des besoins avec l'utilisateur final du système est une étape très importante. Pour cela, nous avons essayé que notre démarche (ainsi que le simulateur) soit simple de sorte qu'elle puisse être adoptée par un concepteur ou encore par un utilisateur non expert du formel.

Nous remarquons que nous avons essayé de prouver la complétude et la conformité des spécifications formelles, mais nous ne pouvons pas garantir que le concepteur n'a pas ajouté des contraintes autres que celles exprimées dans le cahier de charges.

Ce travail peut être étendu de façon que la démarche couvre toutes les contraintes architecturales en particulier celles qui concernent des séquences de composants. Nous pouvons aussi penser à la génération automatique des cas de test et à guider davantage l'utilisateur dans la phase de validation.

Bibliographie

- [1] [O.Barais](#), [L.Duchien](#), [OPAD: Outils pour Architectures Dynamiques](#), *Journées Composants Adaptables, JC 2002*, pages 112-118, octobre 2002.
- [2] R. Ben Halima, M. Jmaiel, K. Drira, Graphical simulation of the dynamic evolution of the software architectures specified in Z, *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages: 45 – 48, 2005.
- [3] R. Bharadwaj, C. Heitmeyer. Verifying SCR requirements specifications using state exploration. *In Proc., First ACM SIGPLAN Workshop on Automatic Analysis of Software*, 1997.
- [4] B.W. Boehm, Verifying and validating software requirements and design specifications, *IEEE Software*, vol.1, No1, January 1984.
- [5] T. Coupaye, R. Lenglet, M. Beauvois, P. Dechamboux, Composants et composition dans l'architecture des systèmes répartis, *FT RD*, octobre 2001.
- [6] E. Durand, A. Deplanche, and Y. Trinquet. Langages de configuration. *Technical Report IRCYN-LangArchi, IRCYN Equipe temps réel*, Mars 1999.
- [7] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, D. Hamilton, Experiences Using Lightweight Formal Methods for Requirements Modeling, *IEEE Transactions on Software Engineering*, pages: 4-14, January 1998.
- [8] ME, Fagan, Design and code inspection to reduce errors man program development, *IBM Systems Journals*, page 182-211, 1976.

- [9] M. S. Feather, Rapid Application of Lightweight Formal Methods for Consistency Analyses, *IEEE Transactions on Software Engineering*, pages 949-959, November 1998.
- [10] D.P. Freedman, G.M. Weinberg, Handbook of Walkthroughs, Inspections and cal reviews, *Boston, Toronto: Little, Brown and Company*.
- [11] D. Garlan and M. Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering* vol. 2, Singapore, 1993.
- [12] V. Gervasi and B. Nuseibeh. Lightweight validation of natural language requirements: a case study. *In Proc. of the 4th IEEE International Conference on Requirements Engineering. IEEE Computer Society Press*, June 2000.
- [13] K. Mc Graw, K. Harbison, User Centered Requirements, The Scenario-Based Engineering Process, *Lawrence Erlbaum Associates Publishers*, 1997.
- [14] [M. Hadj Kacem](#), [A. Hadj Kacem](#), Using UML2.0 and GG for Describing the Dynamic of Software Architectures, *Proceedings of the Third International Conference on Information Technology and Applications*, pages: 46 - 51, 2005.
- [15] C. Heitmeyer, A. Bull, C. Gasarch, B. Labaw. SCR: A tool set for specifying and analyzing requirements. *In Proc.10th Annual Conf. on Computer Assurance(COMPASS '95)*, pages 109–122, Gaithersburg, MD, June 1995.
- [16] C.L.Heitmeyer, R.D. Jeffords, B.G.Labaw, Automated consistency checking of requirements specifications. *ACM Trans. Software Eng. and Methodology*, July 1996.
- [17] D. Jackson, J. Wing, Lightweight Formal Methods, *IEEE Computer*, pages 21-22, April 1996.

- [18] R. A. Kemmerer, Integrating Formal Methods into the Development Process, *IEEE Software*, pages: 37-50, September 1990.
- [19] I. Loulou, A. Hadj Kacem, M. Jmaiel et K. Drira. Towards a unified graph based framework for dynamic component-based architectures description in Z. In *The IEEE/ACS International Conference on Pervasive Services (ICPS'04)*, American University of Beirut (AUB), Lebanon, pages 227–234, July 2004.
- [20] M. Lubars, C. Potts, C. Richer, A review of the state of the practice in requirements modeling. *Proc. IEEE Symp. Requirements Engineering, San Diego 1993*.
- [21] R. Marvie, M. Pellegrini, Modèles de composants, un état de l'art , *RSTIL'objet, vol. Coopération et systèmes à objets*, , page 61 à 89, 2002.
- [22] R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan, Architectural styles, design patterns, and objects. *IEEE Software*, 14(1):43–52, Janvier 1997.
- [24] B. Nuseibeh, S. Easterbrook, Requirements engineering: a roadmap, *{ICSE} - Future of {SE} Track*, pages 35-46, 2000.
- [25] J. Ryser, S. Berner, M. Glinz, On the State of the Art in Requirements-based Validation and Test of Software, *Universitt Zrich, Institut fr Informatik, Zrich, Berichte des Instituts fr Informatik 98.12*, Nov 1998.
- [23] I. Sommerville, P. Sawyer, Requirements Engineering: A Good Practise Guide. *John Wiley & Sons*, 1997.
- [26] M. Spivey, The Z Notation, *Prentice Hall International*, 1 992.
- [27] J. Vuichard, Etude de la plate forme Eclipse, pages 7,8,23, Septembre 2006.

[28] P. Zave, Classification of Research Efforts in Requirements Engineering. *ACM Computing Surveys*, 29(4), pages 315-321, 1997.

[29] «ObjectGeode»: www.telelogic.com/products/additional/objectgeode.

[30] «Objectime»: www.objectime.com/.

[31] «Ibepace»: www.ibepace.com/.

UNE DEMARCHE DE VALIDATION DE SPECIFICATIONS FORMELLES DES ARCHITECTURES LOGICIELLES

Lamia YANGUI BOUAZIZ

Résumé : Dans ce mémoire, nous nous intéressons aux architectures logicielles dynamiques à base de composant et spécifiées selon une approche formelle adoptant la notation Z. Pour aider le concepteur à vérifier qu'il a spécifié toutes les contraintes informelles de l'utilisateur et conformément à ses attentes, nous proposons une démarche pour la vérification de la complétude et conformité de spécifications formelles d'architectures logicielles. Cette démarche se base sur le test de scénarios représentant des configurations architecturales par rapport au style spécifié. Pour valider les configurations, nous avons implémenté un simulateur de test d'architectures logicielles. C'est un outil générique, basé sur une technique formelle et couplé avec l'outil Z/EVES.

Notre démarche est simple de sorte qu'elle peut être adoptée par un concepteur ou un utilisateur non expert du formel.

Mots clés : ingénierie des besoins, validation des spécifications, simulation des architectures logicielles.

Abstract : In this work, we focus on formal specifications of software architectures using the Z notation. In order to check the completeness and conformity of specifications, we define a process which tries to verify if all user requirements are specified. The process is based on the validation of scenarios representing architectural configurations regarding the specified style and using a simulator.

Key-words : requirement engineering, requirement validation, software architecture simulation