Intégration du test dans les composants logiciels

Nicolas Belloir et Jean-Michel Bruel

Université de Pau et des Pays de l'Adour
LIUPPA, B.P. 1155
64013 Pau CEDEX, France
{belloir,bruel}@univ-pau.fr

Résumé

Dans cet article nous présentons, au travers d'une mise en œuvre concrète, notre approche sur l'intégration du test dans les composants logiciels. Le but de ces fonctionnalités de test ajoutées aux composants n'est pas d'en tester à nouveau le fonctionnement, mais de permettre de vérifier à tout moment du cycle de vie qu'il respecte le contrat proposé aux autres éléments de l'application. Cette approche permettra notamment de déployer des composants possédant les outils pour évaluer leur comptabilité avec leur nouvel environnement. Cette technologie, développée dans le cadre du projet européen Component+, est illustrée ici au travers d'un exemple concret.

Mots clés

Composant, Composabilité, Test Intégré, Validation.

1 Introduction

Dans le domaine du développement d'applications, l'ingénierie du logiciel basée composant (*CBSE*) est actuellement la technique la plus recommandée en terme de réutilisation et de réduction des coûts de développement. Non seulement cette approche permet de réutiliser les développements déjà réalisés en interne, mais aussi d'utiliser directement des composants commerciaux (*COTS*).

Le développement d'applications basées sur cette technique repose sur le principe d'assemblage de composants interagissant entre eux. Cet assemblage se fait notamment en mettant en relation les interfaces de composants fournissant un service avec les interfaces d'autres composants nécessitant ce service. L'assemblage de composants n'est cependant pas une tâche aisée et il se pose de nombreux problèmes. Parmi ceux-ci, notons les problèmes d'incompatibilités d'interfaces, de langages, de méthodes de conception, ou encore de comportements.

De nombreux travaux portent sur ces sujets. Cependant, ils sont pour la plupart orientés très fortement sur l'aspect implémentation de ce qu'il est d'usage d'appeler la composition logicielle. Nous pensons pour notre part qu'il est important de prendre en compte cette composition logicielle au plus tôt dans le cycle de développement du logiciel. Cela permettra de développer des composants et des applications basées composant de meilleure qualité et offrant un meilleur potentiel de réutilisation. Pour

cela, dans le cadre d'une thèse qui démarre sur le sujet, nous nous proposons d'étudier cette composition logicielle de manière à fournir des méthodes et des outils aidant le concepteur à orienter le développement d'un logiciel au plus tôt vers les techniques de composition logicielle. Cela permettra de spécifier un ensemble de propriétés et de règles de compositions dès la phase d'analyse. Notre approche en ce sens consiste à modéliser les applications basées composant en utilisant le concept de relation Tout-Partie [BBB01].

Néanmoins, l'expérience montre que dans un contexte industriel les aspects de validation sont bien souvent considérés au second plan par rapport aux aspects de test. Quel que soit le niveau de validation annoncé d'un composant, il est normal de donner la possibilité à ses « acquéreurs » de le tester en situation dans son nouvel environnement. Dans ce contexte, il est nécessaire qu'un composant soit capable de montrer qu'il se comporte bien de la façon qu'il est supposé se comporter, qu'il fournit bien toutes les fonctionnalités qu'il est censé fournir, etc. Dans le but de faciliter et de rendre plus rigoureuse cette étape, un projet européen, *Component+*, se propose de fournir un certain nombre d'outils permettant le test et la validation de composants, aussi bien du point de vue de leur contrat que de leur exécution. L'idée de base est de proposer des composants possédant des capacités de testabilité et de paramétrage tels qu'il est alors possible de les vérifier, y compris en situation d'exploitation (*run-time*). Le LIUPPA[1] est un des contractants principaux dans ce projet, et nous avons fourni une implémentation concrète, sous la forme d'une librairie Java, des principes de bases définis.

Nous abordons dans cet article les principales orientations de notre approche, en illustrant l'utilisation de notre librairie. Ainsi, dans la section 2 nous présentons succinctement le contexte plus général du test de composants, nous détaillons dans la section 3 notre approche, développée au sein de *Component+*, que nous illustrons sur un cas concret dans la section 4. Enfin nous concluons dans la section 5 et présentons un cadre de travail un peu plus général que nous démarrons dans une thèse dans le domaine, et qui permettra de relier l'approche développée ici avec l'approche basée composant en général.

2 Test de composant

Le fait d'intégrer un composant dans une application entraîne en général son utilisation , et donc son test, assez tard dans le cycle de développement. A l'extrême ce composant peut même être « acquis » par l'application au moment de l'exécution (e.g., plates-formes logicielles du type CORBA). Un peu à la manière des composants électroniques, les composants logiciels sont « intégrés » dans une architecture logicielle. Par contre, il est souvent peu utile de tester à nouveau le composant lui-même. Il est en effet la plupart du temps supposé validé par son fournisseur. Il est en revanche indispensable de tester son intégration dans l'architecture développée (communication avec les autres composants, adéquation des interfaces, etc.). De plus, certains composants logiciels intervenant dans des applications critiques (temps réel) nécessitent parfois de pouvoir être vérifiés en situation d'exécution.

[1] Notre laboratoire d'accueil : Laboratoire d'Informatique de l'Université de Pau et des Pays de l'Adour.

Ainsi nous ne nous intéressons pas directement à tester un composant. En effet, contrairement au monde de l'électronique par exemple, la seule condition pour qu'un composant validé (on parle plutôt de certification pour les composants logiciels) cesse soudainement de bien se comporter, vient d'une modification de son environnement. Nous cherchons à doter le composant d'un certain nombre de fonctionnalités qui permettent de tester son comportement dans le nouvel environnement qui lui est fourni.

Les travaux précédents dans le domaine du test intégré au composant (*Built-In Test – BIT*) ont plutôt porté sur le test du composant lui-même (*self-test* [Wang98]), ou bien encore sur l'amélioration du processus de développement des jeux de test [JDT01]. Notre approche vise à doter le composant de capacités permettant de vérifier son comportement dans ses interactions avec les autres composants (via la notion de contrat, ou « *contract testing* »). Nous détaillons dans la section suivante le modèle de composant que nous utilisons et l'approche que nous proposons pour intégrer la testabilité du composant.

3 Présentation de l'approche Component +

Le modèle de composant retenu dans le projet est basé sur celui de la méthode KobrA [Atk01], lui-même basé sur la définition de Szyperski [Szy99] :

> *«A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties. »*

Un composant logiciel est donc vu comme une unité de composition possédant des interfaces contractuelles et un contexte particulier d'exécution. Il peut être déployé indépendamment et composé avec d'autres composants. C'est également la vue prise par la notation UML [OMG00]. L'approche prise dans *Component+* vise à intégrer les modèles et techniques proposés comme une extension de l'approche classique CBSE. Nous avons différencié dans le projet les deux aspects de la vie d'un composant que sont la phase de développement et d'intégration (que nous appellerons *contract testing* dans la suite), et la phase de « déploiement » où l'application est en cours d'exécution (*run-time testing*) [Szy99]. Ainsi nous utilisons, comme le fait par exemple l'approche UML, la notion de *composant* (phase de développement) et d'*instance de composants* (phase de déploiement).

Dans la suite de cette section, nous allons uniquement nous consacrer au contract testing[2], définit par Meyer [Mey97] comme l'« engagement » formel entre un composant et ses utilisateurs (on parlera plutôt de clients). Ainsi, nous souhaitons permettre à un composant, client d'un autre composant (jouant alors le rôle de serveur) d'invoquer des méthodes dédiées permettant de vérifier que ce serveur se comporte bien comme il est censé le faire. Ceci permet une vérification sémantique, complémentaire à la vérification syntaxique classique des interfaces(au niveau des types). L'objectif du contract testing intégré (*BICT –Built-In Contract Testing*) est donc de faciliter la vérification du futur système, améliorant très significativement la réutilisation des composants.

[2] Se reporter au site du projet pour de plus amples détails : `http://www.component-plus.org/`

Un composant « testable » est un composant possédant une partie intégrée, spécifique, permettant à tout composant client de tester le comportement de ce composant. Ces fonctionnalités sont fournies sous la forme d'une interface spécifique de test. Un composant de test (aussi appelé « testeur » par la suite) est un composant spécifique, contenant des jeux de tests individuels permettant de vérifier les propriétés sémantiques du composant qu'il teste.

Nous avons mis en œuvre dans notre équipe une implémentation concrète des principes généraux développés dans le projet autour du contract testing (cf. [AGB01] pour plus de détails). La mise en œuvre logicielle du BICT est propre à chaque composant et étend l'interface du composant en autorisant l'exploration de son comportement. Néanmoins les fonctionnalités de cette interface de test sont similaires d'un composant à l'autre. Aussi proposons-nous une architecture logicielle résumée dans la Figure 1.

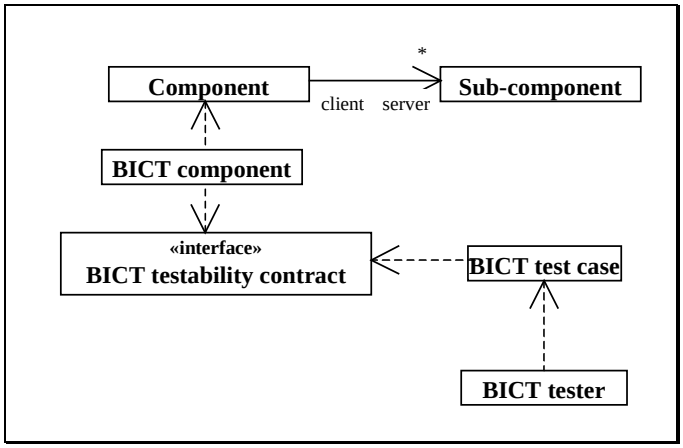

Fig. 1. Une mise en œuvre de la technologie BICT

Dans ce diagramme UML, les flèches en pointillé représentent un lien de dépendance (héritage par exemple si l'on se place dans un contexte orienté objet). Les opérations proposées par l'interface (interrogation et initialisation d'un état, etc.) peuvent être générées automatiquement à partir d'une description du comportement sous la forme d'un automate d'états. Nous nous sommes servis pour cela d'une bibliothèque d'outils pour diagrammes Statecharts, développée dans le cadre d'un projet précédent. Un testeur minimal est également proposé, qui peut être étendu en fonction des besoins du concepteur.

L'exemple d'utilisation de la technologie BICT que nous allons utiliser pour l'illustrer va montrer que même basé sur un composant initial dont on ne possède pas le source, on peut mettre en œuvre notre approche. L'idée principale est bien de séparer le composant lui-même et la partie BIT qui l'accompagne. Ceci permet, en phase de déploiement de n'implémenter éventuellement que le composant lui-même si les contraintes (de temps, de mémoire, etc.) le réclament (ce qui est le cas de la plupart des applications critiques). Nous avons implémenté cette approche au travers d'une librairie Java (BIT/J) pour en montrer la faisabilité et la puissance.

Un composant testable (ou composant BICT), offre des fonctionnalités de test au travers d'une interface spécifique de test. Cette interface, propre à chaque composant, hérite

d'un certain nombre de fonctionalités générales à toute interface de test et c'est pourquoi nous avons introduit une classe *BICT testability contract* (cf. Fig. 2) pour représenter les fonctionnalités minimales et souhaitables à tout composant testable. Dans la Figure 2, les 4 opérations retournent des informations propres au contexte dynamique d'exécution du composant, et sont systématiquement utilisés par les jeux de tests.

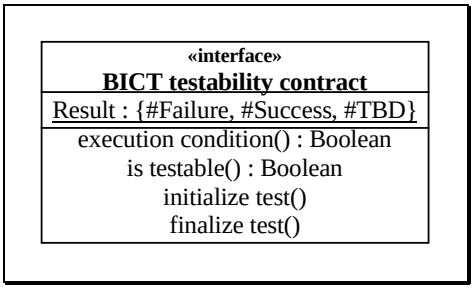

Fig. 2. Un exemple d'interface de test minimale

Le BICT testeur (en bas de la Figure 1) est le point d'entrée pour l'activité de test du composant. Il permet de développer des stratégies de test en fonction du contexte dynamique d'exécution du composant. Nous illustrons dans la section suivante le développement de tous ces éléments BICT dans le cadre d'une petite étude de cas. Cet exemple utilise une version avancée du diagramme présenté en Figure 1, où les états du composant sont utilisés et déterminés à partir d'une modélisation UML du comportement dynamique du composant.

4 Exemple concret de mise en œuvre

4.1 L'exemple du composant Stack

Afin de tester l'applicabilité des notions présentées dans les sections précédentes, nous avons développé une librairie permettant leur mise en œuvre. Nous avons ensuite implémenté un composant testable à l'aide de cette librairie en s'appuyant sur un composant existant. Ce composant est le composant `Stack` qui implémente les fonctionnalités d'une pile. Ce composant peut être vu comme une application basée état, s'appuyant sur trois états :

- `Empty` : La pile ne contient aucun élément.
- `Loaded` : La pile contient un nombre d'éléments inférieur à sa capacité.
- `Full` : La pile contient un nombre d'éléments égal à sa capacité.

Le composant `Stack` possède trois interfaces fournissant des services :

- `push` : Ajoute un élément.
- `pop` : Retire un élément.
- `top` : Accède au dernier élément ajouté à la pile.

Un moyen intuitif pour exprimer le comportement de ce composant consiste à utiliser un automate d'état comme celui illustré en Figure 4. Ce dernier est un automate simplifié pour les besoin de l'exemple.

4.2 Aperçu de notre approche

Comme nous l'avons vu dans les sections précédentes, notre approche est fondée sur les Statecharts et la notion de contract testing. Nous avons réutilisé une librairie nommée Statecharts et nous l'avons étendue en créant une nouvelle librairie nommée `BIT/J`. Ces librairies sont implémentées sur le principe de la réflexion java. Cette technologie permet de retrouver et d'utiliser les méthodes d'un composant binaire écrit en java.

Le principe de notre approche consiste, dans un premier temps, à spécifier le comportement du composant à l'aide de Statecharts dans un contrat de testabilité basé état puis à utiliser les principes de la réflexion java afin de manipuler les méthodes du composant et de tester leur conformité aux besoins spécifiés. La Figure 3 montre le principe de construction de la librairie BIT/J.

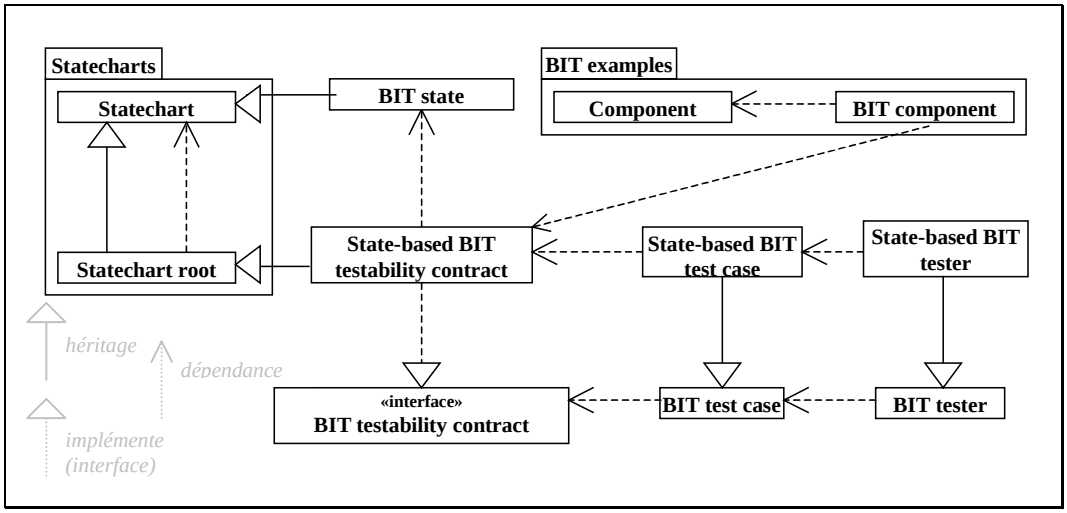

Figure 3 . Principe de la librairie BIT/J

4.3 Un implémentation pas à pas

4.3.1 Du composant au composant testable

Il existe plusieurs manières permettant de lier le composant au composant testable. Premièrement, le composant testable peut être hérité du composant. Cette approche est intéressante car elle permet une manipulation aisée du composant lui-même. En effet, il est ainsi possible d'accéder aux champs et aux méthodes protégées (`protected`) dans le but de positionner le composant dans un état spécifique avant l'exécution du test. Secondement, le composant peut être incorporé en un champ spécifique du composant testable. L'avantage de cette technique est principalement le respect du principe d'encapsulation puisque les champs et les méthodes protégées ne sont pas utilisables. Dans notre exemple, nous avons choisi le contexte de la première approche.

4.3.2 Description de l'automate d'état

Bien que le composant `Stack` puisse être vu comme un composant basé état, il n'implémente pas réellement le code Statechart. Il est donc nécessaire d'implémenter l'automate d'état dans le composant testable. Afin de simplifier notre exemple, nous avons décidé de travailler avec une pile pouvant stocker trois entités. Nous adoptons

donc un nouvel automate d'état composé de quatre états : `Empty, OneElement, TwoElements` et `Full`. La Figure 4 illustre l'automate d'état.

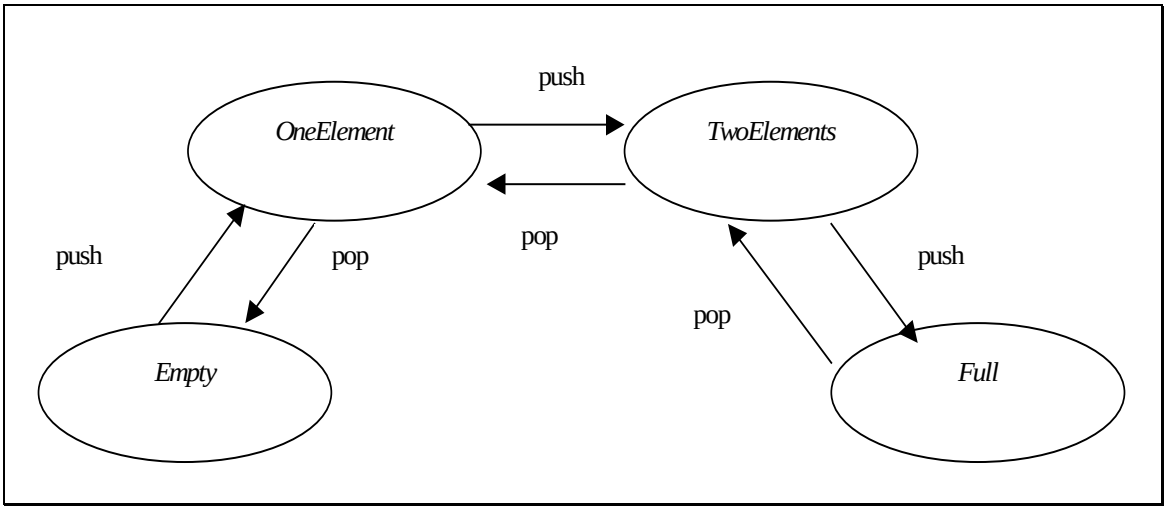

Figure 4 . Un automate d'état pour une pile

La librairie `Statecharts` implémente l'automate d'état par un arbre binaire. La Figure 4 montre le Statechart pour notre automate d'état de la pile et la Figure 5 montre l'arbre binaire correspondant. La racine de l'arbre ainsi que les éléments non nommés ne sont pas des états réels. Chaque état peut être actif ou non actif.

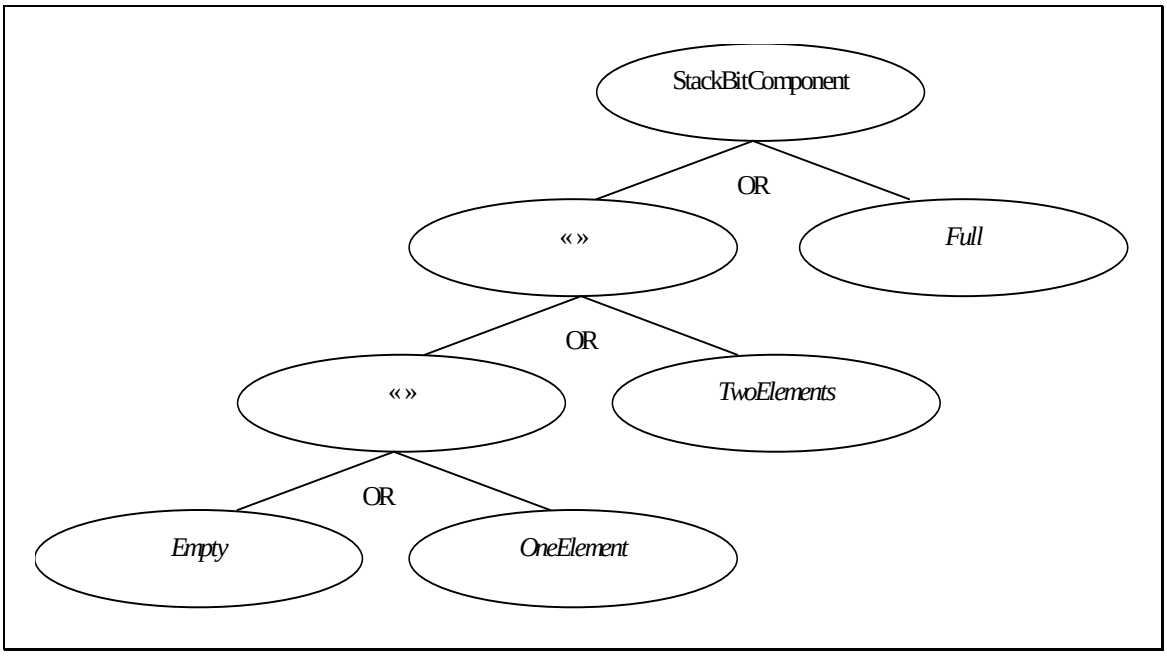

Figure 5. Implémentation de l'automate d'état BIT

Nous allons expliquer maintenant comment implémenter un automate d'état dans un composant testable. Premièrement, il est nécessaire de déclarer et de construire de nouvelles instances de `BIT_state`. `BIT_state` est une classe de la librairie `BIT/J`. Chaque instance de la classe représente un état.

// declaration of BIT states
 	
protected BIT_state _Empty;
protected BIT_state _OneElement;
protected BIT_state _TwoElements;
protected BIT_state _Full;
	// construct of BIT states
	
	_Empty = new BIT_state("Empty");
	_OneElement = new BIT_state("OneElement");
	_TwoElements = new BIT_state("TwoElements");
	_Full = new BIT_state("Full");

Il est possible d'associer une méthode d'un composant testable avec l'entrée ou la sortie du composant dans un état. Cette méthode sera alors automatiquement exécutée lorsque le composant entrera ou sortira dans cet état. Pour spécifier cette association, il faut utiliser les méthodes nommées `entryAction` et `exitAction` de l'objet `BIT_state`. Par exemple, il est possible de spécifier qu'une méthode m soit activée lorsque l'automate d'état arrive dans un état nommé s.

```
_s = (new BIT_state("s")).entryAction(m)
```

Deuxièmement, le contrat de test doit être déclaré et instancié avec l'automate d'état du composant testable. L'automate d'état est spécifié à l'aide des méthodes `and` et `xor` des entités `BIT_state`.

```
protected State_based_BIT_testability_contract _stackBitComponent;
…
…
_stackBitComponent = new State_based_BIT_testability_contract
((((_Empty.xor( _OneElement)).xor( _TwoElements)).xor( _Full)),
"stackBitComponent");
```

Troisièmement, l'état initial de l'automate d'état doit être spécifié. Pour cet exemple, l'état initial est l'état `Empty`.

```
_Empty.inputState();
```

4.3.3 Implémentation des méthodes de test

Les méthodes de test sont des méthodes des composants testable qui exécutent un cas de test spécifique. Chaque méthode de test exécute une méthode du composant dans un état donné du composant. Les méthodes `fires` de la librairie `Statecharts` décrivent un mouvement d'état.
Il est possible de seulement spécifier une transition d'état avec la première méthode `fires`. La spécification est suffisante si l'une des deux entités `BIT_state` est associée avec une méthode d'entrée ou de sortie décrite en amont.
La seconde méthode `fires` fournit la possibilité d'associer une pré-condition (guard)
La troisième spécifie une transition d'état, une pré-condition, l'objet composant traité, le nom de la méthode du composant à exécuter et ses arguments. Si l'exécution de la méthode ne change pas l'état courant de l'automate, les deux premiers arguments de la méthode `fires` doivent être égaux.

```
public void fires(BIT_state from, BIT_state to)
public void fires(BIT_state from, BIT_state to, boolean guard)
public void fires(BIT_state from, BIT_state to, boolean guard, Object
object, String action, Object[] args)
```

Voici un exemple de test sur l'exécution de la méthode `push`. La pile est dans l'état `Empty` et nous voulons tester le fait qu'elle se trouve effectivement dans l'état `OneElement` après l'exécution de la méthode `push`. L'objet `_tab` est un tableau contenant les arguments de la méthode `push`.

```
_stackBitComponent.fires(_Empty,_OneElement,true,this,"push",_tab);
```

La méthode `used_up` exécute les actions spécifiées.
Les méthodes de test peuvent capturer les exceptions et les faire remonter sous la forme de nouvelles exceptions ré-encapsulées qui pourront être interprétées par les cas de test.

Les méthodes de test peuvent être exécutées indépendamment les unes des autres ou bien dans un ordre déterminé afin de faire un cas de test complet.

4.3.4 Implémentation du cas de test basé état

Dans l'objectif d'exécuter un test, il est nécessaire de créer un ensemble de cas de test basés états. Les cas de test exécutent une méthode et interceptent le résultat correspondant de l'exécution. Cette interprétation est basée, premièrement, sur le résultat direct de l'exécution de la méthode de test et, secondement, sur l'état final de l'automate d'état du composant testable. Une méthode de test peut avoir un résultat correct mais peut se terminer dans un état inopportun.
Le constructeur de la classe de cas de test basé état a besoin d'un contrat de testabilité BIT, du nom de la méthode, d'un ensemble d'entrées, d'un objet `comparable` afin de tester le résultat de la méthode de test, et le nom de l'état final.

```
public State_based_BIT_test_case(BIT_testability_contract bc,String
     test,Object[] inputs,Comparable expected_result,String
expected_state)
```

Un cas de test basé état fournit une méthode nommée test afin d'exécuter la méthode de test spécifiée, et une méthode nommée interprétation afin de tester le résultat.

Dans l'exemple suivant, la méthode nommée `func_to_be_tested` va être exécutée et l'automate d'état devra se terminer dans l'état `OneElement`.

```
BIT_stack    _bs = new BIT_stack();
State_based_BIT_test_case _sbbtc2 = new State_based_BIT_test_case
(_bs,
    "func_to_be_tested", null, null, "OneElement");
_sbbtc2.test();
System.err.println("Interpretation: " + _sbbtc2.interpretation());
```

Pour tester une exception (si la méthode testée doit retourner une exception par exemple), il est nécessaire de créer une classe qui implémente l'interface Java `comparable`. Dans notre exemple, nous testons les exceptions `FullStackException` et `EmptyStackException` avec une classe nommée `BIT_stack_exception_result`. Nous avons premièrement créé une nouvelle classe nommée `BIT_stack_exception_result` qui implémente l'interface `comparable`. Lorsqu'une exception est capturée, les mécanismes BIT retournent une

nouvelle exception avec le message spécifique de l'exception initiale. Alors, la méthode de comparaison peut effectuer une comparaison entre l'exception attendue et l'exception capturée. L'exemple suivant illustre l'utilisation d'un objet `comparable`.

```
BIT_stack    _bs = new BIT_stack();
BIT_stack_exception_result _bser1 = new BIT_stack_exception_result
("EmptyStackException");
State_based_BIT_test_case _sbbtc1 = new State_based_BIT_test_case
(_bs,
    "test_1", null, _bser1, "Empty");
_sbbtc1.test();
System.err.println("Interpretation: " + _sbbtc1.interpretation());
```

Le résultat attendu est d'intercepter l'exception `EmptyStackException` et de rester dans l'état `Empty`.
Le code complet de cet exemple ainsi que toute la documentation relative à notre approche sera prochainement disponible sur le site du projet Component+.

5 Conclusion et travaux futurs

Une des principales difficultés de l'approche composant provient du caractère réutilisable et générique des composants qui, s'ils fournissent une panoplie de fonctionnalités (interface) bien définies syntaxiquement, fournissent rarement une expression formelle du comportement qu'ils s'engagent à avoir vis-à-vis des clients qui les utilisent. Dans ce cadre, nous avons travaillé à définir un environnement où la testabilité du composant est fournie avec le composant, au travers d'une interface spécifique de test. Nous avons présenté dans cet article la problématique ainsi que l'approche globale dans laquelle s'inscrivait ces travaux et nous avons montré la faisabilité de notre approche au travers d'une mise en œuvre concrète des principes proposés. Nous sommes en ce moment en train d'appliquer cette technologie à un composant industriel afin de s'assurer de sa faisabilité sur un cas complexe. Ce composant est un thermostat programmable dont le fonctionnement repose sur de nombreux états parallèles ou concurrents, et possédant de nombreuses contraintes temporelles.

Dans le cadre du projet Component+ et avec nos partenaires industriels, nous allons évaluer les apports de cette technologie à l'aide de projets pilotes. Ces projets reprennent notamment le développement d'applications basées composant déjà réalisés en utilisant cette fois la technologie BIT. La comparaison du développement d'applications basées composant suivant un schéma classique ou suivant la technologie BIT nous permettra d'en évaluer les apports et les limites.

En parallèle à ces travaux, nous travaillons à différents niveaux dans le domaine de l'ingénierie logicielle orienté composant. Dans le cadre d'une thèse qui démarre sur le sujet, nous cherchons notamment à définir des méthodes permettant de traiter la composition logicielle dès la phase d'analyse et non plus seulement lors des phases de conception logicielle. En effet, nous pensons que la prise en compte de la composition au plus tôt dans le cycle de vie du logiciel permettra d'améliorer sensiblement la qualité et la facilité d'assemblage des composants logiciels [MN98]. Dans ce sens, nous travaillons sur une étude approfondie de la relation *Tout-Partie* et notamment à sa meilleure modélisation dans UML [BBB01]. Cette relation est en effet, comme nous

l'avons montré, mal traitée notamment dans UML 1.4. Nos travaux font par ailleurs l'objet d'une réponse pour la définition d'UML 2.0., corrigeant quelques lacunes d'UML 1.4.

Afin de proposer un cadre de travail cohérent, nous envisageons par la suite d'intégrer nos différents travaux dans un environnement permettant une prise en charge du développement d'applications basées composant de la phase d'analyse jusqu'à la phase de validation, en passant évidement par les phases de conception et de d'implémentation. Dans ce sens, nous suivons avec attention les travaux menés sur les *frameworks* pour l'environnement intégré et sur les modèles de composants pour la conception et l'implémentation de l'architecture logicielle.

6 Bibliographie

[Atk01] Atkinson, C. and others. Component-Based Product Line Engineering with UML. Addison-Wesley 2001.

[AGB01] Atkinson, C., Groß, H.-G., Barbier, F., Built-In Contract Testing in Component-Based Software Development, submitted to ICSE'2002.

[BBB01] Belloir, N., Bruel, J.-M., Barbier, F., Formalisation de la relation Tout-Partie : application à l'assemblage des composants logiciels. Actes des Journées Composants du 25 et 26 octobre, Besançon 2001.

[JDT01] Jézéquel, J.-M. , Deveaux, D. , Le Traon, Y. , Reliable Objects : Lightweight Testing for OO Languages, IEEE Software, July/August 2001.

[Mey97] Meyer, B. Object-Oriented Software Construction. Prentice Hall, 1997.

[MN98] T. Meijler, O. Nierstasz, "Beyond Objects: Components", Proceedings of Cooperative Information Systems, Trends and Directions, Academic Press, pp. 81-110, 1998.

[OMG00] OMG Unified Modeling Language Specification, Object Management Group, 2000.

[Szy99] Szyperski, C. Component Software: Beyond Object-Orientded Programming. Addison-Wesley, 1999.

[Wang98] Wang, Y., King, G., Patel D., Court, I., Staples, G., Ross, M. and Patel, S., On Built-in Test and Reuse in Object-Oriented Programming, ACM Software Engineering Notes, Vol. 23, No.4, pp.60-64. 1998.