Intégration du test dans les composants logiciels

Nicolas Belloir, Jean-Michel Bruel et Franck Barbier

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

RÉSUMÉ. *Dans cet article nous présentons, au travers d'une mise en œuvre concrète, une approche portant sur l'intégration du test dans les composants logiciels. Le but de ces fonctionnalités de test ajoutées aux composants est de développer des scénarios de test sur mesure. Cela concerne en particulier les composants sur étagère (COTS) développés dans des environnements hétérogènes. Cette technologie, développée dans le cadre du projet européen Component+, est illustrée ici au travers d'un système domotique.*

ABSTRACT. *In this paper, we present, via a concrete realisation example, an approach to integrate tests in software components. The objectives of theses test functionnalities is to develop specific test scenarios. This in particular concerns COTS built in heterogeneous environments. This technology, developed in the Component+ European project, is illustrated here via a domotic system.*

MOTS-CLÉS : *Composant, Composabilité, Test Intégré, Certification.*

KEYWORDS: *Component, Componability, Built-in Test, Certification.*

1. Introduction

Dans le domaine du développement d'applications, l'ingénierie du logiciel basée composants (*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 conçus en interne, mais aussi d'utiliser directement des composants sur étagère (*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 incarnant les services fournis, avec les interfaces d'autres composants nécessitant ces services. 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'interopérabilité sémantique (Heiler, 1995), de composabilité (Meijler *et al.*, 1998), de prédiction de comportements des assemblages (Crnkovic *et al.*, 2002), etc.

Nous pensons qu'il est important de prendre en compte ces problématiques scientifiques au plus tôt dans le cycle de développement du logiciel. Cela permet de développer des composants de meilleure qualité et offrant un meilleur potentiel de réutilisation. Pour cela, nous nous proposons d'étudier la 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 la vérification et la validation expérimentale des comportements des composants.

L'expérience montre que, quel que soit le niveau de certification / qualité de service annoncé d'un composant, il est fondamental 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 attendue (résistence aux fautes, etc.), qu'il fournit bien toutes les fonctionnalités, 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 de sorte qu'il est alors possible de les vérifier en phase finale de développement (contract testing), et en phase de déploiement (quality of service testing).

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 quelques perspectives.

2. Test de composant

Le développement d'applications basées composants implique une modification profonde du cycle de développement du logiciel. Cela est notamment dû à la réutilisation de composants logiciels qui diminue de manière significative la phase de développement. En effet, un peu à la manière des composants électroniques, les composants logiciels sont « intégrés » dans une architecture logicielle sans modification du composant. De plus, 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). Ce type de développement a pour effet d'accroître l'importance des phases d'assemblage et de validation par rapport à la phase de développement de nouveau code, ainsi que d'introduire une distinction entre le développement de composants et le développement d'une application utilisant des composants (Meijler *et al.*, 1998). Cette distinction a une impact au niveau des besoins de tests. En effet, le développeur de composant réalise les tests propres au développement du composant, c'est à dire le test unitaire du composant et son test au sein de son environnement de développement. Le client, en revanche, ne teste généralement pas le composant lui-même (il est supposé validé par le 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.

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 permettant 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* (Wang *et al.*, 1998) -, ou bien encore sur l'amélioration du processus de développement des jeux de test (Jézéquel *et al.*, 2001). 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 projet Component+ se propose de développer une technologie permettant de doter les composants logiciels d'un certain nombre de fonctionnalités afin de tester leur comportement dans le nouvel environnement qui leur est fourni. Dans la suite de cette section, nous allons présenter les grands traits de l'approche Component+.

Le modèle de composant retenu est basé sur celui de la méthode KobrA (Atkinson *et al.*, 2001), lui-même basé sur la définition de Szyperski (Szyperski, 1999) : *«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 (OMG, 2000). 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, et la phase de « déploiement » où l'application est en cours d'exécution (Szyperski, 1999). 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). Respectivement, nous avons considéré les deux phases de test correspondantes (que nous appellerons contract testing et run-time testing dans la suite).

Dans la suite de cette section, nous allons uniquement nous consacrer au contract testing[1], basé sur la notion de contrat définie par Meyer (Meyer, 1997) comme l'« engagement » formel entre un composant et ses utilisateurs (on parlera plutôt de clients). Nous souhaitons permettre à un composant, client d'un autre composant (jouant alors le rôle de serveur) d'invoquer des opérations 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.

Un composant « testable » (ou composant BITC) 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.

---

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

Nous avons mis en œuvre une implémentation concrète des principes généraux développés dans le projet autour du contract testing - cf. (Atkinson *et al.*, 2002) 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. 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, positionnement du composant dans un certain état, etc.) peuvent être générées automatiquement à partir d'une description du comportement sous la forme d'un automate d'états. En effet, l'automate d'états nous permet de spécifier le comportement du composant de manière formelle. 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. Les tests sont implémentés à l'aide de cas de test (*BICT test case*).

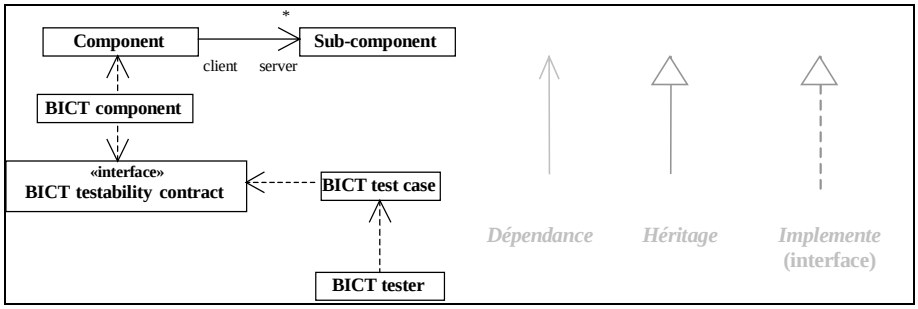

Figure 1- Une mise en œuvre de la technologie BICT

Un point important de notre approche est que, même pour un composant initial dont on ne possède pas le source, on peut la mettre en œuvre. En effet, l'idée principale est bien de séparer le composant lui-même et la partie BIT qui l'accompagne. Ceci permet également, 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).

Un composant testable, 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 fonctionnalités générales à toute interface de test et c'est pourquoi nous avons introduit le contrat de testabilité BITC (*BICT testability contract*) pour représenter les fonctionnalités minimales et souhaitables à tout composant testable. La Figure 2, illustre ce contrat de testabilité. Les 4 opérations de l'interface

BIT_testability_contract retournent des informations propres au contexte dynamique d'exécution du composant, et sont systématiquement utilisés par les jeux de tests. Notons qu'elles ne nécessitent pas que le comportement du composant soit spécifié à l'aide d'états. Les opérations fournies par le contrat *State_Based_BIT_testability_contract* sont, elles, spécifiques aux composants dont le comportement est spécifié par des états. Elles permettent de le manipuler pour le tester du point de vue de son comportement.

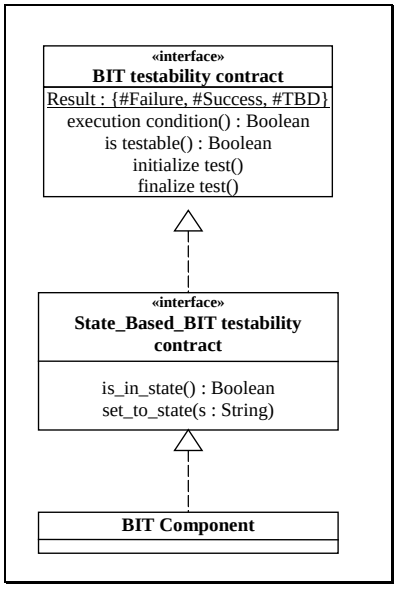

Figure 2– Extraits de l'organisation de la BIT/J

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 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

Afin de vérifier l'applicabilité des notions présentées dans les sections précédentes, nous avons développé une librairie, nommée BIT/J, les mettant en

œuvre. Nous avons ensuite utilisé cette dernière lors de plusieurs études de cas. Dans cette section, nous allons en premier lieu présenter la librairie BIT/J, puis l'une de ces études. Pour cela, nous allons présenter le composant Thermostat Programmable puis la construction du composant BITC correspondant, en insistant sur la démarche employée et sa simplicité.

4.1 La librairie BIT/J

La librairie BIT/J est une mise en œuvre concrète des principes énoncés dans les sections précédentes. Elle utilise les fonctionnalités d'une librairie permettant l'implémentation d'applications fonctionnant suivant le principe des Statecharts (Harel, 1987), développée lors d'un projet précédent. Notamment, elle permet de décrire le comportement des composants à l'aide d'un automate d'états. Elle a été développée en Java et utilise (entre autre) les mécanismes de la réflexion Java (possibilité de retrouver et d'utiliser les opérations de l'interface d'un composant binaire écrit en java, cela dynamiquement). La Figure 3 représente l'architecture de la librairie. L'utilisation de la librairie BIT/J est expliquée sous forme de tutoriel ci-après.

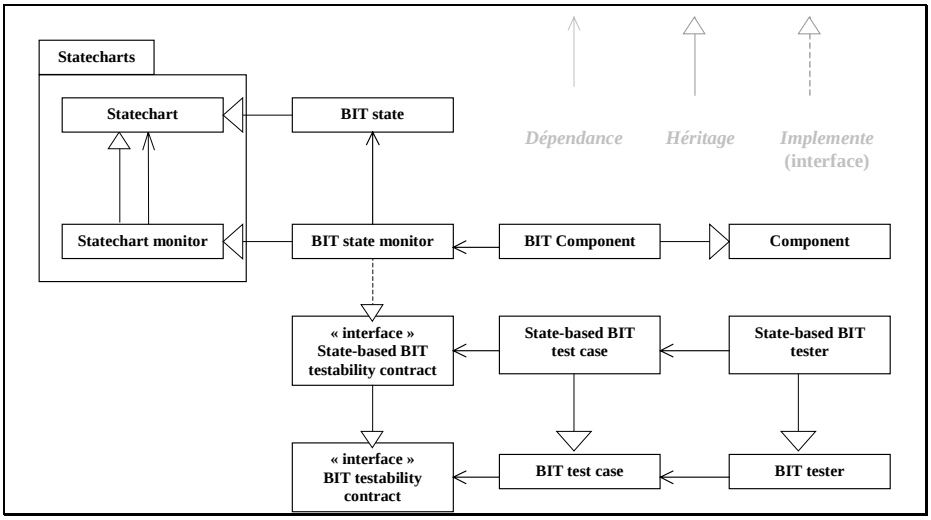

Figure 3 - Architecture de la librairie BIT/J

4.2 Le composant Thermostat Programmable

Le thermostat programmable est un composant industriel développé lors d'un projet précédent. Nous l'avons utilisé pour évaluer le bon fonctionnement de la librairie BIT/J. Il s'agit d'un composant logiciel Java s'intégrant dans une application domotique (*Home Automation System*) assurant la régulation de la température dans une pièce. Pour cela, il gère trois sous-systèmes (chauffage, climatisation et ventilation) en fonction de la température demandée par l'utilisateur, à l'aide d'une IHM, et de celle mesurée par un capteur. Ces trois éléments peuvent être vus comme des sous-composants et sont encapsulés dans le composant Thermostat Programmable. Ce dernier est conçu pour être composé avec une IHM.

Nous avons la spécification complète de l'application et du composant thermostat en UML (Class Diagrams et Statechart Diagrams). Cet article n'ayant pas pour objet de la décrire complètement, nous ne donnons ici que la spécification de l'interface fournie par le composant (voir Figure 4). Nous invitons le lecteur intéressé à consulter le reste de la documentation sur le site du projet.

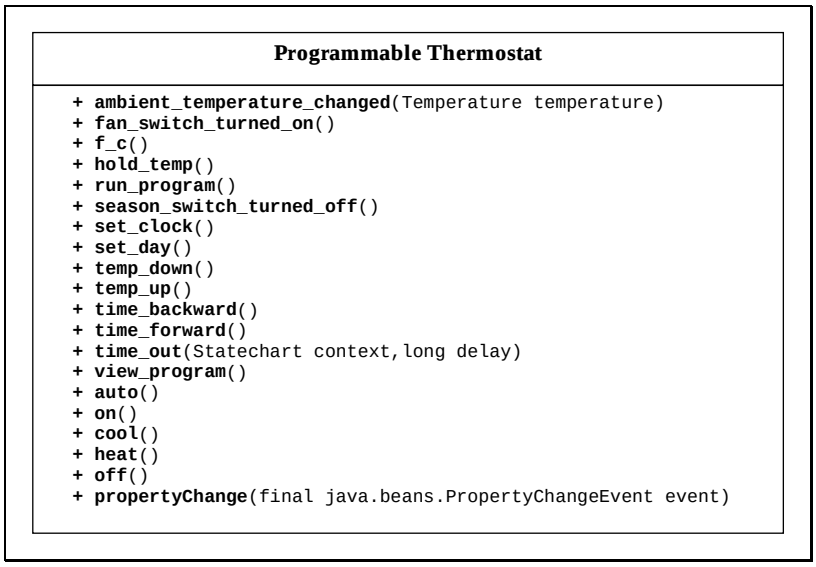

Figure 4 - Interface du Thermostat Programmable

Cette étude de cas est intéressante sur plusieurs points. D'une part, le composant a un comportement complexe et est donc particulièrement illustratif de la facilité de mise en œuvre de cette technologie. D'autre part, il a été implémenté lui-même en utilisant la librairie *Statechart*. Cela permet une manipulation aisée de ce composant par son interface de test. En effet, il est alors très simple de définir des cas de tests associés au comportement du composant.

4-3 Implémentation BIT du Thermostat Programmable

4.3.1 Vue générale

Nous invitons le lecteur à s'appuyer sur le schéma de la Figure 3 afin de mieux comprendre la mise ne œuvre de la technologie BIT à l'aide de notre librairie. La réalisation d'un composant BITC et son évaluation se déroule en quatre étapes. La première peut être différente en fonction de la nature du composant, selon qu'il ait été construit « from scratch », c'est à dire conçu depuis le début comme un composant BITC, ou qu'il ait été fabriqué à partir d'un composant existant. Elle consiste à lier le composant BITC au composant initial. La seconde étape consiste à spécifier le comportement du composant dans la partie BIT du composant BITC. La spécification du comportement est basée sur la technique des *Statecharts*. La troisième étape consiste à implémenter l'*interface générique de test* pour définir des tests basiques, ou pour fournir au testeur toutes les opérations nécessaires pour manipuler le composant. Enfin, lors de la quatrième étape, le client du composant BITC définit des cas de test spécifiques (cas où le client souhaite développer des tests particuliers non prévus dans le composant BITC) et les exécute ou exécute les cas de test déjà définis dans le composant BITC.

4.3.2 Etape 1 – Lier le composant à sa version BITC

Il existe plusieurs manière de lier le composant à sa version BITC. Premièrement, cette dernière peut hériter du composant. Cette méthode est intéressante car elle permet de manipuler le composant lui-même (partie *public* et *protected* du composant). En effet, il est alors possible, par exemple, de donner accès à des attributs et des opérations *protected*, afin de positionner le composant dans un état particulier avant l'exécution d'un test spécifique. Cela permet également d'accéder à la description de son comportement dans le cas où le composant a été conçu initialement pour prendre en compte la technologie BIT. Deuxièmement, le composant peut être inclut comme un attribut particulier dans sa version BITC. L'avantage principal de cette technique est le respect du principe d'encapsulation puisque les attributs et opérations *protected* ne sont alors pas accessibles. Dans notre exemple, nous avons choisi la première approche.

4.3.3 Etape 2 - Description du comportement à l'aide d'un automate d'états

Pour décrire le comportement du composant, il faut le spécifier dans un automate d'états propre au composant BITC. Pour ce faire, le développeur spécifie les différents états, puis l'automate et la manière dont les états sont reliés entre eux, et enfin définit les points d'entrée dans l'automate.

Notre librairie BIT/J est construite à partir de la librairie Statecharts. Nous avons donc implémenté la notion d'état en tant que *BIT_state*, qui est une spécialisation de la notion de *Statechart*. De même, l'automate d'états est implémenté en tant que *BIT_state_monitor*, qui est une spécialisation de la notion de *Statechart_monitor*.

Dans note étude de cas, le comportement du composant thermostat programmable est déjà implémenté dans le composant lui-même en utilisant les fonctionnalités de la librairie Statecharts. On peut donc surcharger l'automate d'états du composant existant avec les fonctionnalités de la librairie *BIT/J* dans le composant BITC. Pour ce faire, il faut spécifier un nouvel automate dans le composant BITC (ce qui aurait été de toute manière le cas si le composant n'avait pas été développé en utilisant une méthode orientée *Statecharts*).

Un des intérêts que présente l'implémentation des composants en utilisant la librairie Statecharts consiste dans le fait que l'automate du composant BITC réutilise alors les attributs qui décrivent le modèle d'état du composant. Cela est possible car ces attributs sont de type *Statecharts*. On peut alors en manipulant le composant BITC directement manipuler le composant initial sans aucun intermédiaire. Il est intéressant de noter que l'utilisation de la technologie BIT est vraiment aisée et puissante lorsqu'elle est appliquée à un composant qui a été conçu en utilisant des aspects comportementaux implémentés à travers des « états ».

La librairie *Statechart* implémente l'automate d'états comme un arbre binaire. La racine de l'arbre, comme les éléments non nommés ne sont pas des états physiques. Chaque état est actif ou non actif. Elle permet de relire chaque état à ses états adjacents à l'aide méthode implémentant le *et* et le *ou exclusif* logiques.

Nous expliquons maintenant comment implémenter un automate dans le thermostat programmable BITC. Premièrement, en général, il est nécessaire de déclarer et de construire de nouvelles instances de BIT_state. Dans le cas de cette étude, nous réutilisons les champs de type *Statechart* qui décrivent l'automate du composant initial. Ci-dessous, nous avons la déclaration de quatre états (*Run, Hold,Control* et *Current_date_and_time_displaying*) de type *Statechart* dans le composant initial ainsi que celle de l'automate d'états.

```
// declaration of the statecharts in the existing component
protected Statechart   _Run;
protected Statechart   _Hold;
protected Statechart   _Control;
protected Statechart   _Current_date_and_time_displaying;
...
protected Statechart_monitor   _Programmable_thermostat;
```

Dans notre librairie, la classe *BIT_state* qui représente la notion d'état, hérite de la classe *Statechart*. Ainsi, nous pouvons écrire dans le composant BITC :

```
// declaration of the statemachine in the BITC
// set of simple BIT_states

_Run = new BIT_state("Run");
_Hold = new BIT_state("Hold");
_Control = new BIT_state("Control");
. . .
```

L'exemple précédent illustre l'instanciation des états simples de types *BIT_state*. Il est également possible de construire des états complexes sur le même schéma, eux-

mêmes construits à partir d'états simples ou complexes. Afin de les relier entre eux, les opérations appelées *and* et *xor* sont utilisées.

```
// complex BIT_state
_Operate=((_Run).xor(_Hold)).and(…)).name("Operate");
```

Il est possible d'associer une opération du composant initial ou du BITC avec une entrée (ou une sortie ou un passage interne) dans un état. Celle-ci est alors exécutée automatiquement. Pour le spécifier, nous utilisons les opérations appelées *entryAction*, *internalAction* et *exitAction* de la classe *BIT_state*. Par exemple, nous pouvons spécifier qu'une opération appelée *Op1* est activée lorsque l'automate arrive dans l'état appelé *S1*. A chaque fois que l'automate arrivera dans l'état *S1*, *Op1* sera exécutée.

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

// complex BIT_state
_Operate = (((_Run) …).name("Operate")).internalAction("time-
out", this, "set_time",null);
...
_Setup = (( (...)).name("Setup")).internalAction(...);
```

Une fois déclarés tous les états, l'automate d'états du composant BITC doit être déclaré et instancié. Il est spécifié avec les opérations *and* et *xor* (voir ci-dessus).

```
// State machine
_Programmable_thermostat = new BIT_state_monitor
(((_Operate).xor( _Setup)).and( _Control),
"Programmable_thermostat");
```

Pour finir, l'état initial de l'automate doit être spécifié à l'aide de l'opération *inputState()*. Dans notre exemple, les états initiaux sont *Hold* et *Current_date_and_time_displaying*.

```
_Hold.inputState();
_Current_date_and_time_displaying.inputState();
```

4.3.4. Etape 3 – Définition de l'interface de test

Nous avons, dans l'étape 2, spécifié le comportement du composant initial dans le composant BITC. Nous allons maintenant définir les opérations de test de l'interface de test.

Il y a plusieurs manières de configurer et d'implémenter des opérations de test BITC. Premièrement, la librairie BIT/J fournit un ensemble d'opérations génériques qui peuvent être surchargées par les développeurs pour implémenter des nouveaux cas de test. Ces opérations sont définies dans deux classes de la librairie. La première est la classe *BIT_testability_contract*. Elle a été créée afin de permettre de développer des contrats de testabilités BIT *classiques* permettant la mise en œuvre des pré, post conditions et invariants. Nous disons classiques en opposition aux contrats de testabilité BIT basés états (*State_Based_BIT_testability_contract*) qui permettent au développeur de définir des tests portant sur le comportement du

composant. Les interfaces des différentes classes sont décrites dans la Figure 2. Elles permettent de définir des pré et post-conditions pour un test ou un besoin spécifique dans un test particulier. Les opérations *is_in_state( )* et *set_to_state( )* permettent de manipuler l'automate afin de tester l'état d'un composant ou de le positionner dans un état particulier avant un test par exemple.

Deuxièmement, en plus des interfaces génériques de la technologie BIT, le développeur peut définir et implémenter d'autres opérations de test fournies avec le composant BITC dans le but de rendre plus efficace le composant dans son environnement de déploiement. Par exemple, ces opérations peuvent tester les transitions les plus communes sur l'automate. Comme autre exemple, on peut présenter les opérations qui préparent le composant avant l'exécution d'un test (positionnement dans un état spécifique lorsque cela est trop complexe pour l'opération *set_to_state(), …*).

```
// In the BITC Component
public boolean Configuration_1 () throws . . . {
     // Put the BCIT in a specific state before the test
     set_to_state("Hold");

     //execution of the "set_clock()" method
     set_clock( );
     . . .
}
```

4.3.5 Etape 4 - Instantiation et exécution des cas de test BIT

Le tester doit en premier lieu instancier le composant BITC. Ensuite, il peut jouer les jeux de test. Pour cela, il instancie les cas de tests (*BIT_test_case* et *State_based_BIT_test_case*) en spécifiant l'opération de test qu'il veut exécuter. Puis, il exécute l'opération réellement à l'aide de l'opération *test()* et interprète le résultat à l'aide de l'opération *interpretation()*.

Les cas de test peuvent avoir plusieurs formes. Premièrement, le client du composant BITC peut tester directement une opération fonctionnelle du composant dans un certain état. L'exemple suivant montre la réalisation d'un test sur l'opération *set_clock( )* du thermostat programmable. Le résultat attendu est le positionnement du composant dans l'état *Setup*.

```
// bc is a BIT component
BIT_programmable_thermostat bc = new BIT_programmable_thermostat
(temperature);

// Put the BCIT in a specific state before the test
bc.set_to_state("Hold");

// Definition on the test case: in this case we test the
// execution of the "set_clock()" method.
// We want to be in the state "Setup" at the end of the
// execution of this method.
State_based_BIT_test_case sbbtc1 = new State_based_BIT_test_case
(bc, "set_clock", null, null, "Setup");

// Execution of the test case
sbbtc1.test();

// Interpretation of the result of the test case: true or false
System.err.println("\tInterpretation: " +
sbbtc1.interpretation());
```

Deuxièmement, le client du composant BITC peut exécuter une opération de test définie dans le composant BITC. On peut par exemple, si on reprend l'exemple précédent, encapsuler l'appel à la méthode *set_clock( )* dans une opération du composant BITC (comme montré dans l'exemple de code de la section précédente). L'avantage de cette approche est le respect du principe d'encapsulation, mais également le fait qu'on puisse systématiser cette approche en vue de problématique de génération automatique d'opérations de test, par exemple.

```
// In the Tester

// bc is a BIT component
BIT_programmable_thermostat bc = new BIT_programmable_thermostat
(temperature);

// Execution of the test case: We want to be in the state
// "Setup" at the end of the execution of this method.
State_based_BIT_test_case sbbtc1 = new State_based_BIT_test_case
(bc, "Configuration_1", null, null, "Setup");

// Execution of the test case
sbbtc1.test();

// Interpretation of the result of the test case: true or false
System.err.println("\tInterpretation:" +
sbbtc1.interpretation());
```

4.5.6. Bénéfices

Dans cette section, nous allons illustrer un exemple d'erreur que la technologie BIT peut détecter. Le thermostat a été conçu pour automatiquement se positionner dans l'état *Hold* à partir de l'état *Setup* lorsque aucune action n'a été effectuée pendant un délai spécifique. Nous avons simulé une erreur dans notre composant, afin de tester la détection d'une erreur dans ce type de transition. Le test consiste à positionner le composant BITC dans l'état *Setup*. De là, nous avons implémenté une attente du tester de 20 secondes. Puis nous avons testé le fait que le composant ait bien transité dans l'état *Hold*.

```
/* Instantiation of the BIT component */
public boolean Configuration_2 () throws . . . {

     /* BCIT is putting in the Setup State */
     set_to_state("Setup");

     /* Starting of a timer for 20 s */
     try{
         Thread.sleep(20000);
     }
     catch(Exception e){…}

     /* Test if the BCIT is in the Hold State */
     if (is_in_state("Hold"))
         return (true);
     else
         return(false);
}
```

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'inscrivaient ces travaux et nous avons montré la faisabilité de notre approche au travers d'une mise en œuvre concrète des principes proposés sur un composant industriel. 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 composants déjà réalisées en utilisant cette fois la technologie BIT. La comparaison du développement d'applications basées composants suivant un schéma classique ou suivant la technologie BIT nous permettra d'en évaluer les apports et les limites. Nous finalisons également, d'une part, l'ajout d'outils permettant d'automatiser une partie de la phase de développement du composant BITC, notamment en fournissant le squelette du composant BITC ainsi qu'un certain nombre d'opérations de test générées automatiquement. D'autre part, nous finalisons un outils permettant la génération d'un tester graphique minimal.

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, nous

cherchons notamment à définir des méthodes et des outils formels permettant de traiter la composition logicielle dès la phase d'analyse et non plus seulement lors des phases de conception logicielle. En effet, sur l'aspect méthodologique, 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 (Meijler *et al.*, 1998). Dans ce sens, sur l'aspect formalisation, nous travaillons sur une étude approfondie de la relation *Tout-Partie* et notamment à sa meilleure modélisation dans UML (Belloir *et al.*, 2001). 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. D'autre part, nous envisageons de définir un cadre formel permettant la prise en compte le plus tôt possible des propriétés comportementales du composant, qui par extension offrira la possibilité de définir formellement les propriétés sémantiques liées à la certification.

Enfin, afin de proposer un cadre de travail cohérent, nous étudions la possibilité, par la suite, d'intégrer nos différents travaux dans un environnement permettant une prise en charge du développement d'applications basées composants de la phase d'analyse jusqu'à la phase de validation, en passant évidement par les phases de conception et 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

Atkinson C. *et al.*, *Component-Based Product Line Engineering with UML*, Addison-Wesley, 2001.

Atkinson C., Groß H.-G., Barbier F., *Built-In Contract Testing in Component-Based Software Development*, submitted.

Belloir N., Bruel J.-M., Barbier F. « Formalisation de la relation Tout-Partie : application à l'assemblage des composants logiciels », *Actes des Journées Composants :flexibilité du système au langage*, Besançon, 25-26 octobre 2001.

Crnkovic I., Schmidt H., Stafford J., Wallnau K., « Anatomy of a Research Project in Predictable Assembly », *Fifth ICSE Workshop on Component-Based Software Engineering* - White paper, 2002.

Harel D., « Statecharts : a visual formalism for complex systems », *Science of Computer Programming*, Vol 8, pp 231-274, 1987.

Heiler S., « Semantic Interoperability », *ACM Computing Surveys*, vol 27, n :2, pp 271-273, 1995.

Jézéquel J.-M., Deveaux D., Le Traon Y., « Reliable Objects : Lightweight Testing for OO Languages », *IEEE Software*, Juillet / Août 2001.

Meyer B., *Object-Oriented Software Construction*, Prentice Hall, 1997.

Meijler T., Nierstasz O., « Beyond Objects: Components », *Cooperative Information Systems, Trends and Directions*, Academic Press, pp. 81-110, 1998.

OMG, *Unified Modeling Language Specification*, Object Management Group, 2000.

Szyperski C., *Component Software: Beyond Object-Orientded Programming*, Addison-Wesley, 1999.

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.