Test intégré dans les composants logiciels

Nicolas Belloir - Jean-Michel Bruel - 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 functionnality is to develop in-situ test scenarios. This in particular concerns COTS built to be deployed in heterogeneous environments. This technology, developed within the Component+ European project, is here illustrated here via a home automation 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 dans ce contexte est fondé sur le principe de composition de composants interagissant entre eux. Cette composition est réalisée en connectant les interfaces des composants fournissant des services, avec celles des clients qui requièrent ces services. Cependant, la composition de composant n'est actuellement que syntaxique et pose donc beaucoup de problèmes. Parmi ceux-ci citons les recherches portant sur l'interopérabilité sémantique [HEI 95], la composabilité [MEI 98], ou encore la prédiction des comportements des assemblages [CRN 02].

A cause de leur haut niveau de réutilisabilité, les composants nécessitent de faire une distinction entre les besoins du développeur du composant (son fournisseur) et ceux de son utilisateur (acheteur, client le cas échéant) [MEI 98]. L'expérience montre que quelque soit le niveau de certification / qualité de service annoncé d'un composant, il est fondamental de donner à ses utilisateurs la possibilité de le tester en situation dans son nouvel environnement (ce qui est appelé "run-time testing" dans le reste de cet article). Dans ce contexte, il est nécessaire pour un composant d'être capable de montrer qu'il se comporte selon sa spécification dans la phase finale de développement, ou dans sa phase d'intégration et de déploiement. Dans ce but, nous avons développé une librairie Java implémentant ces principes. Pour faciliter l'intégration de composant, nous préconisons une technique de conception spécifique pour les composants basée sur une représentation de leur comportement par les Statecharts de Harel [HAR 87]. L'obstacle que présente l'accès au code source des composants (notamment avec les COTS) est une des préoccupations récurrentes liées à la CBSE. Dans le contexte Java, nous utilisons alors les capacités de réflexion de ce langage pour permettre un accès dynamique, lors de l'exécution, aux propriétés internes. Des protocoles de test sont spécialement écrits une fois pour toute et aident considérablement la manière dont les composants peuvent être acquis, évalués et finalement (ré)utilisés.

Nous présentons dans cet article les principales orientations de notre approche et nous illustrons l'utilisation de notre librairie. Pour cela, dans la section 2, nous présentons le contexte plus général du test de composant. Notre approche est illustrée au moyen d'un exemple concret dans la section 3. Finalement, nous concluons dans la section 4 et présentons des perspectives.

2. Le test des composants

A la manière des composants électroniques, les composants logiciels “sont intégrés” dans les architectures logicielles sans modification à priori. Dans certains cas, les composants peuvent même être incorporés dans les applications lors de l’exécution (comme dans les plates-formes de type CORBA). Ce type de développement a pour effet d’accroître l’importance des phases d’assemblage et de validation vis à vis de celle d’implémentation. De plus, il est nécessaire de faire une distinction entre le développement d’un composants et le développement d’une application utilisant ce composant [MEI 98]. Cette distinction a une impact au niveau des besoins de tests : le développeur du composant construit des tests qui sont alors dédiés pour le test unitaire du composant, et ce, au sein de son environnement de développement ; l’utilisateur, en revanche, ne teste généralement pas le composant lui-même (il est supposé être validé, et même certifié, par le fournisseur). Il est néanmoins essentiel de tester son intégration dans un contexte réel d’exécution (communication avec les autres composants, tolérance aux fautes, conformité avec le comportement spécifié . . .).

Notons l’existante de travaux dans le domaine du test intégré dans les composants ont plutôt porté sur le self-test [WAN 98] (c’est à dire l’exécution automatique de tests), ou encore sur l’amélioration des définitions des jeux de test [JEZ 01].

2.1. Built-In Test : l’approche Component+

Comme nous l’avons dit avant, les deux phases dans la construction d’applications basées composants nécessitant de nouvelles méthodes de test sont la phase d’intégration et celle de validation. Les deux phases de test correspondantes sont le contract testing et le QoS testing. La Figure 1 illustre ces deux approches.

Le contract testing vérifie si un composant déployé dans un nouvel environnement est capable de fournir les services qu’il est supposé délivrer, c’est à dire s’il remplit son contrat vis à vis des composants qui utilisent ses services. C’est approche est basée sur la notion de contrat définit par Meyer dans [MEY 97]. Cette approche repose essentiellement sur la mise en accessibilité au client de moyens de test permettant la définition de contrats mais également un accès aux états du composant. Cela permet au client de placer le composant dans un état spécifique, d’invoquer un ou plusieurs services, et de vérifier la correction du résultat, notamment au niveau de l’état final dans lequel se trouve le composant (il faut que ce soit celui spécifié). Ces contrats de tests sont typiquement ceux réalisés lorsqu’un système est configuré pour la première fois, ou lorsqu’une reconfiguration est réalisée (par exemple, remplacement d’un composant par un autre au cours de la vie du composant).

Le QoS testing se concentre sur la vérification du comportement du composant envers la totalité de l'application ou du système. Il s'agit de tester que le composant lorsqu'il est déployé (en exécution) se comporte bien comme il doit le faire d'une part, et d'autre part qu'il est bien adapté au système (et inversement) d'autre part. Ce dernier type de test concerne par exemple les problématiques de type performance, mémoire, interblocage . . .

Dans la suite de cet article, nous nous intéressons essentiellement au contract testing.

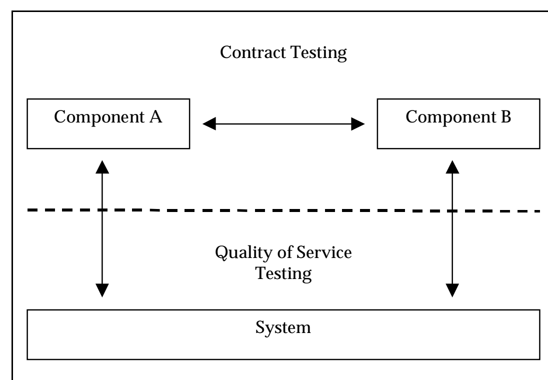

Figure 1 – Contract Testing Vs QoS Testing

2.2. Contrat de testabilité intégré

Nous considérons un composant comme un agrégat de sous-composants qui sont les implémentations des opérations qu'il fournit [BAR 02]. En Java, un tel composant est réalisé via une classe qui possède des champs dont les types sont de ceux de ses sous-composants et ce, récursivement. La Figure 2 décrit la micro-architecture dans laquelle un composant anonyme est connecté avec les classes prédéfinies de notre librairie. Un composant BIT est construit de telle sorte qu'il acquière toute les propriétés de "Component". Dans la Figure 2, une dépendance UML est utilisée dans le but de laisser le choix d'un mécanisme de programmation adéquat (l'héritage est souvent utilisé). L'interface BIT testability contract, détaillée dans la section Figure 3, est un ensemble d'opérations qui sont systématiquement utilisées dans un cas de test BIT, lui-même systématiquement utilisé par un testeur BIT. BIT test case est une classe Java qui a figé des protocoles de test, nommés "initialiser le test", "établir les condition d'exécution", "récupérer les résultats et/ou

les échecs" et “finaliser le test”. Chaque composant BIT doit personnaliser (surcharger) ces actions basiques en tenant compte, opportunément, des valeurs propres de “Component”. Le testeur BIT permet de développer des scénarios de test : séquences, résultats attendus, politique de séquencement...

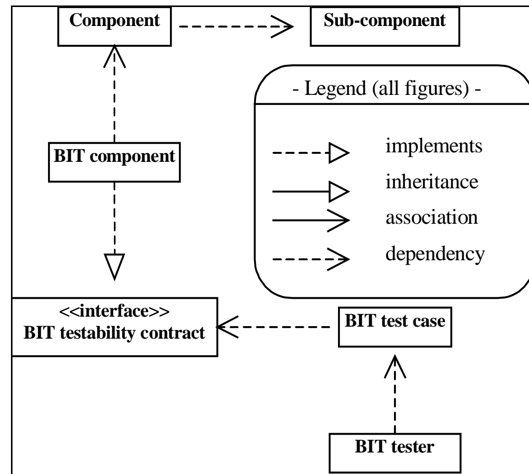

Figure 2 – Dépendances dans le contrat de testabilité intégré

Un grand bénéfice de cette approche est que la majeure partie du processus des tests ne dépend pas de la spécificité du composant évalué. Par exemple, des composants de concurrence qui peuvent être achetés afin de répondre à des besoins très spécifiques, peuvent être comparés sur la base du même cadre de test. D'ailleurs, la substance des tests est construite réellement dans “BIT component”. Le composant BIT peut être déployé dans le but de mesurer la qualité de service en exécution. Notons cependant que dans ce cas là, il faut prêter attention aux charges d'exécution et de ressource : le composant BIT crée une certaine surcharge comparé au composant seul. Un inconvénient de notre approche est que les sous-composants sont des entités entièrement encapsulées, et comme tels, il est difficile d'analyser et de déterminer des diagnostics à un niveau profond d'agrégation. Ainsi, dans le but de fournir un meilleur accès à l'intérieur du composant, nous avons amélioré la librairie pour prendre en compte les états parallèles et concurrents.

2.3. Contrat de testabilité basé « état »

La Figure 3 montre une interface étendue appelée “State-based BIT testability contract” à partir de laquelle un composant BIT peut-être défini. Les flèches blanches avec des pointillés correspondent à la relation Java d'“implémentation”

entre une classe et une interface. Cette seconde approche d'exécution de Built-In Contract Testing est plus contraignante dans le sens où un automate d'états est nécessaire pour le test des composants. Celui-ci décrit formellement le comportement du composant. Bien que des telles spécifications soient communes dans les systèmes en temps réel par exemple, on ne peut pas toujours en fournir. Ainsi, un travail de réingénierie est parfois nécessaire pour extraire des spécifications comportementales à partir d'un composant existant. Nous développons un exemple compréhensif dans la section suivante.

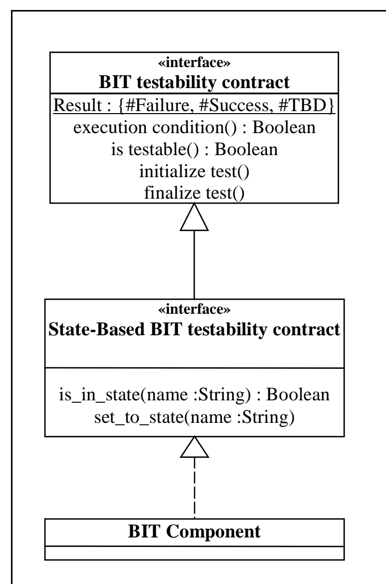

Figure 3– Extension de la librairie BIT/J pour le contrat de testabilité basé état

3. Un exemple concret de mise en œuvre

Dans le but d'illustrer l'applicabilité des concepts présentés au-dessus, nous présentons, premièrement, le composant Thermostat Programmable. Ensuite, nous discutons de la construction du composant BIT correspondant, en insistant sur un processus précis pas à pas.

3.1. La librairie BIT/J

La Figure 4 est une représentation complète de la librairie BIT/J. Le test intégré peut premièrement être mis en œuvre par les trois principaux éléments nommés BIT testability contract, BIT test case et BIT tester. Dans ce cas, la technologie BIT ne traite que l'évaluation des résultats de calcul, de l'environnement d'exécution et des erreurs. Pour des cas plus compliqués, trois éléments basés états sont requis : State-based BIT testability contract, State-based BIT test case et State-based BIT tester. Puisque ces trois derniers utilisent le formalisme de Harel appelé Statecharts [HAR 87], une sous-librairie liée (le package "Statecharts" : en haut et à gauche de la Figure 4) est réutilisé (c'est à dire Statechart et Statechart monitor). BIT state et BIT state monitor sont des spécialisations contextuelles de la sous-librairie dans le but de créer une connexion avec State-based BIT testability contract, State-based BIT test case et State-based BIT tester.

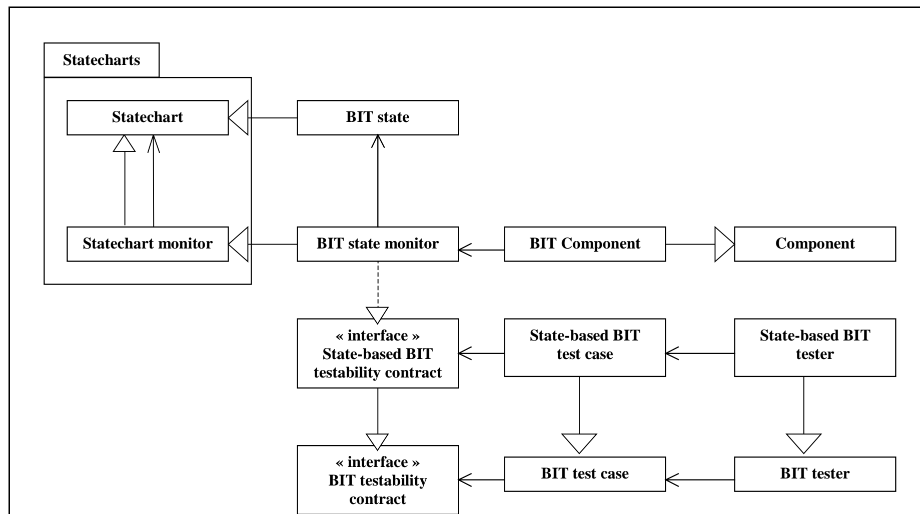

Figure 4 - Architecture de la librairie BIT/J

3.2. 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.

Nous donnons ici la spécification de l'interface fournie par le composant Thermostat Programmable (voir Figure 5). Les opérations dans l'interface sont requises par les clients. Elles masquent l'intérieur du composant Thermostat Programmable. Le test traditionnel est confiné à l'activation de ces opérations bien que les résultats intermédiaires soient cachés, les états et les erreurs possibles doivent être connus pour évaluer le fait que le composant soit conforme à sa spécification lors de son déploiement.

3.3. L'implémentation BIT du composant Thermostat Programmable

3.3.1. Approche Générale.

L'implémentation du composant BIT et son évaluation se déroulent en quatre étapes. La première peut être réalisée de différentes manières selon la nature du composant : en effet, l'approche diffère s'il a été construit entièrement "from scratch", c'est à dire conçut depuis le début comme un composant BIT, ou si il a été construit à partir d'un composant existant. Notre méthode consiste, premièrement, à lier la partie BIT au composant original. Deuxièmement, il est nécessaire d'implémenter la spécification du comportement dans la partie BIT. Cette description du comportement est basée sur la technique des Statecharts de Harel. La troisième étape consiste à fournir une implémentation pour l'interface concernant les contrats de testabilité standards. Enfin, lors de la dernière étape, un client donné du composant BIT définit des cas de test spécifiques joués par le testeur.

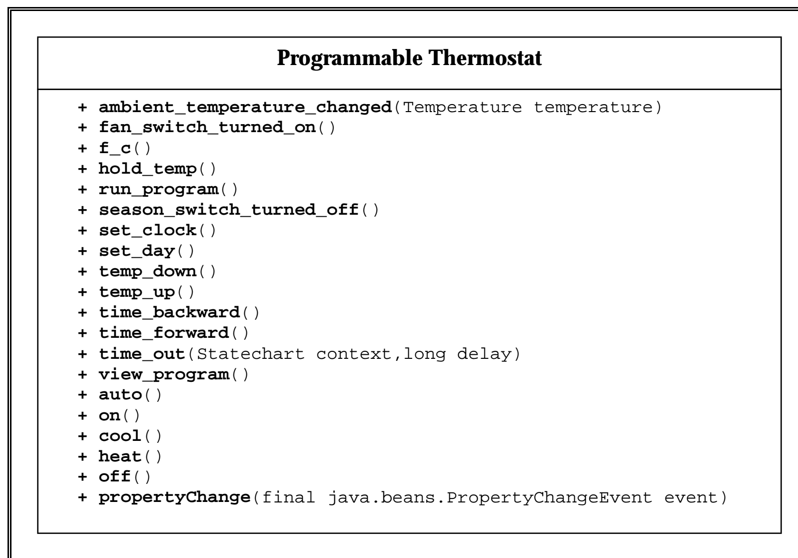

Figure 5 - Interface du Thermostat Programmable

3.3.2. Etape 1 : Lier le composant à sa version BIT.

Il existe plusieurs manières pour lier un composant à sa partie BIT. Premièrement, la partie BIT peut hériter du composant. Cette méthode est intéressante dans le sens où elle permet de manipuler le composant lui-même directement. En effet, il est alors possible, par exemple, d'accéder aux attributs et aux opérations "protected", dans le but de positionner le composant dans un état particulier avant l'exécution d'un test spécifique. Cela permet également de décrire directement son comportement en fonction de ses valeurs d'attribut si le composant a été initialement conçu pour supporter la technologie BIT. Deuxièmement, le composant peut être inclus comme une valeur de champs particulier dans sa version BIT. L'avantage majeure de cette approche est le respect de l'encapsulation puisque les attributs et les opérations "protected" ne sont pas accessibles. Dans la suite de cet exemple, nous avons choisis la première approche.

3.3.3. Etape 2 : Description du comportement à l'aide d'un automate d'états.

Pour décrire le comportement du composant, il doit être spécifié au moyen d'un automate d'états se conformant au formalisme de Harel (Figure 6).

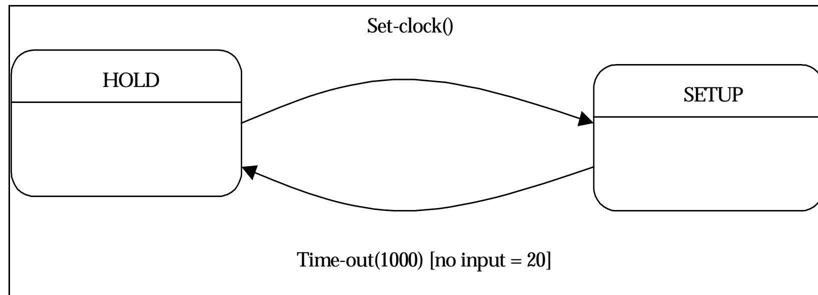

Figure 6 – Extrait de l'automate du Thermostat Programmable

Chaque état est un attribut du composant BIT de type BIT_state (ou Statechart). Un attribut spécifique et unique, représentant l'automate d'états est également implémenté dans le composant BIT de type BIT_state_monitor (ou Statechart_monitor). Ci-dessous nous avons la déclaration de quatre états : Run, Hold, Control et Current_date_and_time_displaying ainsi que d'un automate d'états :

```
protected Statechart _Run;
protected Statechart _Hold;
protected Statechart _Control;
protected Statechart _Current_date_and_time_displaying;
...
protected Statechart_monitor _Programmable_thermostat;
```

Ci-dessous, nous avons l'instanciation de simples occurrences d'état. Il est également possible de construire des états complexes, récursivement. Pour les connecter ensemble, il faut utiliser les opérateurs *and* (concurrence entre les états) et *xor* (exclusivité entre les états). Leur gestion est réalisée à travers leur affectation (par exemple, on affecte à _Operate une expression spécifiant la manière dont ses sous-états sont reliés entre eux).

```
// Instanciation d'états simples
_Run = new BIT_state("Run");
_Hold = new BIT_state("Hold");
_Control = new BIT_state("Control");
// . . .
```

```
// Instanciation d'un état complexe
_Operate=((_Run).xor(_Hold)).and(…)).name("Operate");
```

Il est également possible d'associer une opération avec une transition entrante (ou sortante ou interne). Cette opération est alors automatiquement exécutée *Ces opération peuvent être celles des sous-composants*. Cela permet la corrélation entre les états des composants BIT et les activation des sous-composants. Cela permet également de tracer les chemins d'exécution en cas d'erreur à un haut niveau de profondeur.

```
// Accès automatique dynamique à l'action interne
// "set_time" à travers l'opération "time-out"
// fournie par l'interface du composant BIT.
_Operate = (((_Run) …).name("Operate")).internalAction("time-
out", this, "set_time",null);
// ...
```

Une fois tous les états définis, l'automate d'états du composant BIT doit être déclaré et instancié :

```
// Automate d'états
_Programmable_thermostat = new BIT_state_monitor
((_Operate).xor(_Setup)…);
```

Finalement, les états initiaux de l'automate d'états doivent être établis. Dans notre exemple, les états initiaux sont Hold and Current_date_and_time_displaying.

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

3.3.4. Etape 3 : Définition de l'interface de test.

Nous avons, dans l'étape 2, spécifié le comportement du composant initial dans le composant BIT. 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 BIT. 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 BIT 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(), . . .).

L'extrait de code suivant illustre ce type d'opération :

```
// In the BIT Component
public boolean Configuration_1 () throws . . . {
     set_to_state("Hold");

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

3.3.5. Etape 4 : Instanciation et exécution des cas de test BIT.

Le tester doit en premier lieu instancier le composant BIT. 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 BIT 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 BIT component in a specific state before the test
bc.set_to_state("Hold");

// Definition on the test case: the third argument is the
// expected result
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();

// Get the test case result
System.err.println("\tInterp : " + sbbtc1.interpretation());
```

Deuxièmement, le client du composant BIT peut exécuter une opération de test définie dans le composant BIT. On peut par exemple, si on reprend l'exemple précédent, encapsuler l'appel à la méthode set_clock( ) dans une opération du composant BIT (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 operation.
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());
```

3.3.6. Analyse de la qualité de service.

Bien que notre librairie soit initialement dédiée au contract testing, elle permet également la prise en compte de certains tests portant sur la qualité de service offerte par le composant. Un exemple de mauvais fonctionnement que la technologie BIT peut aider à détecter est illustré ci-dessous. Le Thermostat Programmable est conçu pour être automatiquement forcé dans l'état Hold si, lorsqu'il est dans l'état Setup, aucune action n'est effectuée durant 20 secondes. Nous simulons un défaut au moyen

d'une attente fictive. Les différents systèmes d'exploitation peuvent conduire à des résultats différents pour un test spécifique.

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

     set_to_state("Setup");

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

     /* Test if the BIT component is in the Hold State */
     if (is_in_state("Hold"))
            …
     else
            …
}
```

4. Conclusion et futurs travaux

Les comportements individuels des composants sont souvent inclus dans leur partie cachée. Une spécification formelle des comportements est une première étape pour le rendre effectivement crédible mais est insuffisante. Le Built-In Test est une autre étape pour aider à créer la confiance. Les vendeurs peuvent équiper leurs composants avec la technologie de test développée ici afin de les tester directement, spécialement lorsque leur environnement de déploiement est inconnu. Nous améliorons la puissance de l'idée de la "conception par contrat" de Meyer en permettant d'une part la description d'assertions (pré-conditions, post-conditions et invariants) basées sur les états complexes parallèles, et parfois concurrent, du composant qui s'articulent largement sur les sous-composants. D'autre part, nous basons notre approche sur les capacités de réflexion de Java qui assurent que les protocoles de test sont écrits une fois pour toute. Dans ce contexte, les propriétés des composants sont accessibles dynamiquement en exécution. Le processus de test est lancé à l'aide d'objets cas de test permettant l'utilisation des interfaces de contrats de testabilité, qui sont implémentés par les composants BIT. Il y a en fait deux manières de travailler : soit les fournisseurs adhèrent à notre technique de conception de composant, soit les utilisateurs doivent établir des composants BIT à partir des composants ordinaires. Dans le premier cas, les fournisseurs ajoute de la valeur à leurs composants COTS en les dotant de fonctionnalités de test paramétrables. Dans le deuxième cas, des composants doivent être adaptés afin de pouvoir être adaptés à la bibliothèque BIT/J. Limitée au monde Java, notre approche est cependant un support concret pour la notion de Built-In Test qui a été définie plus théoriquement dans d'autres articles. Le composant Thermostat Programmable est dans cet esprit un exemple représentatif. Nous pouvons, par exemple, mesurer les événements liés au timer qui peuvent varier beaucoup d'un environnement à un autre. Finalement, nous

fournissons également un processus pas à pas de développement basé sur la rationalité. La perspective la plus significative est l'intégration de la bibliothèque BIT/J dans le framework JMX (Java Management eXtensions). Les composants BIT pourront alors être évalués via des navigateurs Web. Nous avons l'intention dans un futur proche de développer cette approche à distance dans un projet qui vise à offrir la possibilité d'acquérir des composants sur le Web.

5. Bibliographie

[BAR 02] BARBIER F., « Composability for Software Components : an Appoach Based on the Whole-Part Theory », December 2002, Proceedings of The Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2002), Greenbelt, Maryland.

[CRN 02] 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.

[HAR 87] HAREL D., « Statecharts : a visual formalism for complex systems », *Science of Computer Programming*, vol. 8, 1987, p. 231–274.

[HEI 95] HEILER S., « Semantic Interoperability », *ACM Computing Surveys*, vol. 27, n2, 1995, p. 271–273.

[JEZ 01] JEZEQUEL J.-M., DEVEAUX D., TRAON Y. L., « Reliable Objects : Lightweight Testing for OO Languages », *IEEE Software*, vol. 18, n4, 2001, p. 76–83.

[MEI 98] MEIJLER T. D., NIERSTRASZ O., « *Cooperative Information Systems – Trends and Directions* », chapitre Beyond Objects : Components, p. 49–78, Academic Press, San Diego, CA, 1998.

[MEY 97] MEYER B., *Object-Oriented Software Construction*, Prentice Hall, Second Edition, 1997.

[WAN 98] WANG Y., KING G., PATEL D., COURT I., STAPLES G., ROSS M., PATEL S., « On Built-in Test and Reuse in Object-Oriented Programming », *ACM Software Engineering Notes*, vol. 23, n4, 1998, p. 60–64.