Partie 8

Test de composants logiciels

Jean-Michel Bruel, Franck Barbier, Nicolas Belloir et Fabien Roméo

Résumé

Nous traitons dans ce chapitre du test de composant « en situation ». Contrairement au monde de l'électronique par exemple, la seule condition pour qu'un composant validé cesse soudainement de bien se comporter, vient d'une modification de son environnement. Nous traitons donc du test d'intégration et détaillons une approche concrète visant à doter un composant d'un certain nombre de fonctionnalités internes, dont la personnalisation et l'usage, aident au moment du déploiement à évaluer le composant, voire à le re-configurer dynamiquement. Nous discutons en particulier de test intégré et illustrons cette idée via la mise en œuvre d'une librairie Java pour les composants sur étagère notamment (COTS components).

Introduction

Le développement d'applications basées composants (CBSE – *Component-Based Software Engineering*) implique une modification importante du cycle de développement du logiciel (cf. chapitre 4). Intégrer un composant dans une application entraîne en général son utilisation, et donc son test, assez tard dans le cycle de développement. Ce composant peut même être « acquis » par l'application au moment de l'exécution (e.g., plates-formes logicielles de type CORBA, cf. chapitre 11). Dans ce type de développement, les phases d'assemblage et de validation sont primordiales.

La distinction entre le développement de composants et le développement d'applications utilisant des composants a un impact au niveau des besoins de tests [MeNi98,HaLS99]. En effet, le développeur de composants 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. D'une part, il est supposé validé par le

fournisseur, et d'autre part le code source est généralement indisponible, rendant les techniques de test traditionnelles caduques. Il est en revanche indispensable de tester son intégration dans l'architecture développée (communication avec les autres composants, adéquation des interfaces, etc.). 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. L'expérience montre que quelque soit le niveau de certification supposé d'un composant, il est indispensable de le tester dans son nouvel environnement. De plus, certains composants logiciels intervenant dans des applications critiques (temps réel) nécessitent parfois de pouvoir être vérifiés en situation d'exécution. Il est donc primordial, pour un utilisateur de composants, de disposer de moyens d'appréhender cette activité de validation.

Nous traitons dans ce chapitre du test du comportement d'un composant « en situation » (déploiement) et détaillons une approche concrète visant à doter un composant d'un certain nombre de fonctionnalités internes, dont la personnalisation et l'usage aident, au moment du déploiement, à évaluer le composant, voire à le re-configurer dynamiquement. Nous discutons en particulier de tests intégrés (*Built-In Test* – BIT [Comp01]) et illustrons cette idée par la mise en œuvre d'un outil Java plus particulièrement dédié aux composants sur étagère (COTS *components*).

Ce chapitre est organisé de la façon suivante : nous traitons rapidement dans une première section du test de composants en général, principalement pour le comparer avec le test d'applications classiques. Dans la deuxième section, nous détaillons une approche industrielle concrète permettant d'intégrer les fonctionnalités de test dans des composants. Cette approche, générale, est illustrée et mise en œuvre au travers d'un outil Java que nous avons développé et qui est décrit dans la troisième section. Nous concluons enfin ce chapitre en faisant le point des techniques disponibles et des enjeux restant à traiter dans ce domaine. Pour faciliter la lecture, les termes importants ou en anglais sont mis en *italique* et les éléments du modèle BIT sont indiqués dans la police Tahoma.

1 Test de composants

L'activité de test est une activité complexe, impliquant, entre autres, la conception des jeux de test, leur exécution, leur interprétation, etc. Ce chapitre n'a pas pour objectif de traiter de tous ces aspects. Nous souhaitons montrer l'importance d'automatiser le plus possible cette activité, et de fournir l'ensemble des moyens et supports à cette activité cruciale dans le développement d'application basées sur les assemblages de composants.

Validation et test

Ce chapitre étant consacré aux tests de composant, il nous faut rappeler la place du test de composant par rapport à l'activité plus générale de validation. En effet, dans le développement d'application classique, on oppose parfois « validation *a priori* », où par exemple le code est contraint par les outils de développement (génération automatique, outils formels, *model-checking*…), et « validation *a posteriori* » où l'implémentation est mise à l'épreuve par rapport à la spécification du système désiré (preuve formelle si la spécification l'est, test du système en situation…). Dans le cas particulier du CBSE, même les composants les plus sûrs nécessitent d'être testés dans leur environnement d'exécution, car seul cet environnement final, généralement indisponible aux concepteurs des composants, possède les caractéristiques susceptibles de générer des erreurs au niveau des composants eux-mêmes, comme de leur assemblage. De plus, cette activité de test doit être disponible à tout moment vu le caractère dynamique et changeant des environnements actuels (e.g., distribution, caractère temps-réel).

Dans ce chapitre nous ne présentons que les aspects « test » de l'approche développée dans notre équipe. Nous travaillons activement en parallèle sur les aspects générations de code sûr et formalisation des assemblages de composants [BaBB03]. Ces aspects sont en effet indispensables à un véritable environnement de développement CBSE. Citons par exemple une des conclusions du rapport « Logiciels et systèmes à haute confiance » du bureau américain de coordination sur la recherche et le développement en technologie de l'information [HCSS01] :

> *"For example, automated validation technology is needed that can consider "correct-by-construction" techniques (…) As one example, methods might be explored that are analogous to proof-carrying code for certifying software."*

Tests unitaires, d'intégration, d'interopérabilité, de conformité

Le test des logiciels a beaucoup évolué ces dernières années à cause du caractère de communication et de distribution de plus en plus généralisé des applications. Par exemple l'automatisation de la génération de tests à partir d'une description formelle d'un système a atteint une certaine maturité [JéDL01]. Cependant, l'avancement de ces techniques concerne le plus souvent les méthodes de test classiques. Quand ces méthodes sont appliquées aux tests de composants imbriqués dans un système complexe, elles se révèlent alors inadaptées ainsi que les architectures proposées pour effectuer les tests.

En effet, contrairement aux approches de test traditionnelles, les méthodes de test de composants sont basées sur le fait que la plupart du temps le composant doit être traité comme une boîte noire, accessible seulement à partir d'un nombre limité de services dont le fonctionnement (comportement) n'est pas toujours clairement (formellement) défini. De plus, le test de composants doit éviter de tester des parties du système considérées comme libres d'erreurs, en se concentrant sur les interactions entre composants et les éventuelles nouvelles erreurs qu'elles génèrent. De nouvelles approches, de nouvelles stratégies, voire de nouvelles architectures de test, qui tiennent compte de ces spécificités, sont donc nécessaires. Ce sont notamment ces raisons qui ont conduit à des projets européens comme Component+ [Comp01], dédié à cette problématique, et dont les résultats seront présentés dans la section 2.

Mais avant de détailler les aspects de test des composants, explicitons quelques termes sont liés à cette activité. On distingue généralement quatre types de tests, en fonction des phases de développement auxquelles ils sont liés. Les premiers tests, appelés *tests unitaires*, concernent les composants élémentaires de l'application à tester. La phase suivante concerne les *tests d'intégration*, correspondant au moment où les composants élémentaires sont assemblés. On teste ensuite les éléments fonctionnels de l'application en examinant comment les différents composants collaborent, on parle ainsi de *test d'interopérabilité*. On teste enfin l'application complète dans son environnement d'exécution. On parle alors parfois de tests système, mais comme ils consistent à vérifier l'absence de fautes ou de comportements incorrects de l'implémentation par rapport à la spécification du système attendu, on parle le plus souvent de *tests de conformité*. En général ce genre de test consiste à envoyer des entrées au système (donnée, ou signal) en attendant de sa part une sortie particulière. On parle alors de *tests actifs*, le testeur maîtrisant complètement les événements envoyés à l'implémentation. On peut également pratiquer un *test passif* (on parle alors plutôt de *monitoring*), consistant à laisser s'exécuter le système tout en observant et en analysant son comportement.

Test des assemblages de composants

De nombreux travaux ont pour objet de trouver des outils capables de tester, vérifier et certifier les composants logiciels. Citons à titre d'exemple le programme RNTL *Cote* (impliquant l'INRIA, le LSR-Imag, Softeam, France Télécom R&D et Gemplus). Le but était de fournir aux développeurs de composants des techniques de test automatique de composants applicables aux technologies couramment employées dans le domaine.

Nous nous intéressons plus particulièrement dans ce chapitre aux développeurs d'applications basées composants plus qu'aux développeurs de composants. Nous ne traitons donc pas par exemple de la génération de test ou des problèmes de couverture des jeux de tests. L'objectif de la technique présentée dans la section suivante est de fournir aux assembleurs l'ensemble des outils permettant de mettre en œuvre leur activité de test.

Pour une étude complémentaire des différentes approches de test de composants existantes, nous renvoyons le lecteur à [Bhor01], mais pour conclure cette section sur le test de composants, nous mentionnerons les trois questions principales que doivent se poser les concepteurs de composants testables, proposées par Gao [Gao00] :

> *"The first question is how to design and define the common architecture and test interfaces for testable components. The next question is how to generate testable components in a systematic way. The final question is how to control and minimize program overheads and resources for supporting tests of testable components."*

La technique que nous présentons dans ce chapitre tente de répondre précisément à ces trois questions.

2 Built-In Test : intégration de fonctionnalités de test

L'approche présentée ici est issue du projet européen Component+ (IST 1999-20162[1]). Ce projet a développé une nouvelle technique permettant d'intégrer des fonctionnalités de test (*Built-In Test* ou plus simplement BIT dans la suite de ce document) à des composants logiciels. Une architecture utilisant ces fonctionnalités a été définie. Elle permet le développement et le test de *composants testables*. Dans le projet, trois types de tests ont été identifiés :

- *Contract testing* : concernant le niveau assemblage, et basé « état » quand la description des composants se fait par des machines à états ;
- *Quality of service testing* : concernant aussi le niveau assemblage, mais traitant plus particulièrement des aspects comme la résistance aux pannes, la maîtrise des temps de latence ou encore la trace des exécutions ;
- *COTS-based product testing* : concernant le niveau acquisition, et traitant plus particulièrement des composants sur étagère.

En effet, comme nous l'avons dit précédemment, 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. Pour la première phase, appelée *Contract Testing* dans le projet, le test est basé sur le

[1] Pour de plus amples détails consulter : `http://www.component-plus.org/`

concept de contrat entre entité au sens de Meyer [Meye97], et pour la deuxième phase, appelée *QoS Testing* dans le projet, le test est basé sur une approche de supervision (*monitoring*) proche des mécanismes d'administration des systèmes distribués. Nous focalisons sur la première phase. La Figure 1 illustre la place de ces deux types de test. Le troisième type de test, dédié aux composants sur étagère traite du problème particulier de ce type de composants. Nous y reviendrons longuement.

La technologie développée au sein du projet a été mise en œuvre sur quatre études de cas et neuf projets pilotes industriels représentants une large palette de domaines d'application. Les résultats de ces études ont été très positifs, montrant une amélioration significative de la qualité des composants développés et bien sûr de leur « testabilité ». Les efforts d'intégration ont ainsi été considérablement diminués (environ 50%).

Après avoir, dans ce qui suit, examiné les deux grandes activités de test définies dans le projet, nous détaillons le développement de composants testables, et enfin nous traitons du cas particulier des composants COTS.

Tests basés Contrat versus test basés Qualité de Service (QoS)

Le *contract testing* permet de vérifier 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. Cette approche, basée sur la notion de contrat définit par Meyer [Meye97], 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 des services qu'il propose, et de vérifier la pertinence 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).

Le *Quality of Service testing*, pour sa part, est un processus continue de vérification du comportement d'un assemblage de composants qui coopèrent dans un système donné. Cette activité de test est menée au niveau système et recouvre des aspects comme les temps d'exécution, les charges de processeurs, les inter-blocages, etc. La Figure 1 synthétise le positionnement de ces deux types de test l'un par rapport à l'autre.

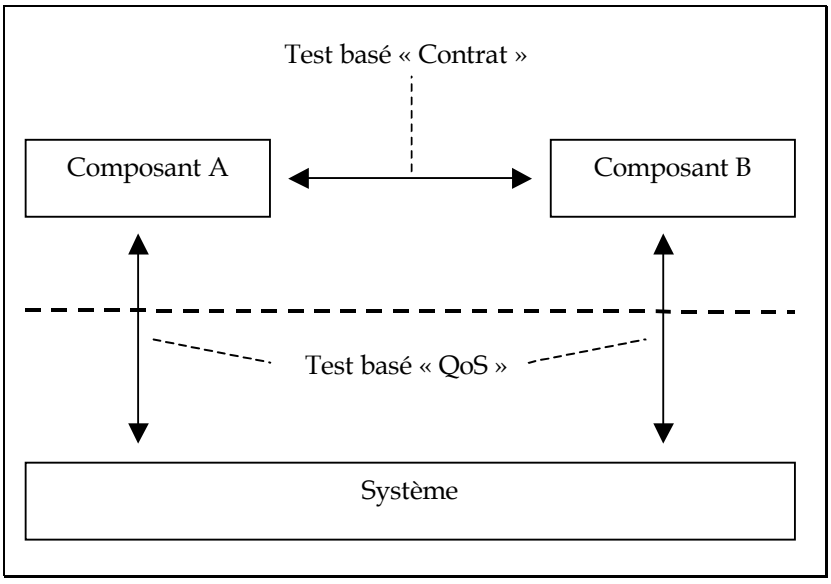

Fig. 1. Les deux grands types de test retenus

Développement de composants testables

Le modèle de composant retenu dans le projet est basé sur celui de la méthode KobrA [Atki01], lui-même basé sur la définition de Szyperski [Szyp99] :

> *«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. L'approche prise dans Component+ vise à intégrer les modèles et techniques proposés comme une extension de l'approche classique CBSE. Un composant est défini par un ensemble d'interfaces qui permettent d'en apprécier les fonctionnalités. Parmi ces interfaces on différencie les interfaces fournies (*provided*), par lesquelles le client du composant accède aux services qu'il propose, et les interfaces requises (*required*), qui définissent les services dont le composant a lui-même besoin pour fonctionner correctement.

Composants testables

Pour mettre en œuvre la testabilité des composants, la notion de *composant testable* est introduite. Un composant testable (BIT-component) comprend les interfaces fonctionnelles du composant, plus une ou plusieurs interfaces de test (cf. Figure 2). Le concepteur du composant fournit ces interfaces aux utilisateurs.

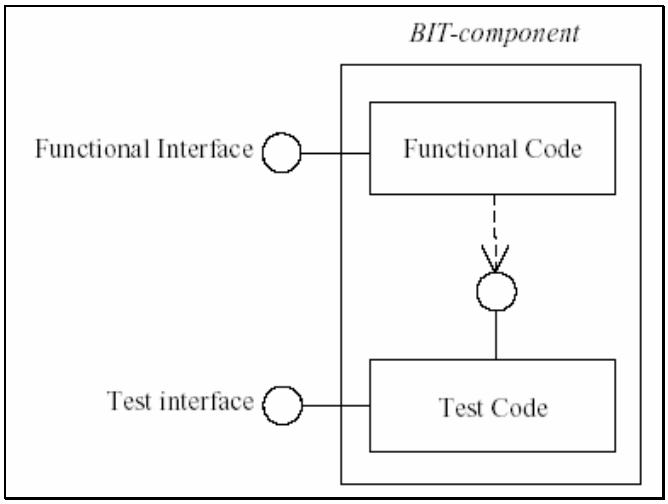

Figure 2. Concept de composant testable (BIT-component)

Parmi les erreurs qui peuvent se produire dans un système basé composant, il y a celles propres à un composant donné, et qui peuvent être détectées directement par des tests intégrés au composant (e.g., cohérence des données), et celles d'un niveau système (e.g., inter-blocages) qui nécessitent des entités de test externes au composant lui-même. Une véritable architecture de test est donc proposée.

Architecture de test

L'architecture BIT est basée sur les éléments suivants :

- BIT-components*:* des composants testables au sens de ceux définis précédemment,
- Testers*:* des composants utilisant les services des BIT-components pour détecter des erreurs de niveau système,
- Handlers*:* des composants qui traitent les erreurs détectées au niveau du BIT-component ou au niveau du Tester.
- System constructor*:* un élément conceptuel responsable de l'instanciation des différents éléments architecturaux ainsi que de leur interconnexion.

Au niveau BIT component, les erreurs internes au composant testable sont détectées. Au niveau tester, les erreurs externes (système) provenant d'une mauvaise interaction entre composants sont détectées. L'architecture spécifie un certain nombre d'interfaces particulières et bien définies, que doivent fournir les implémentations des éléments architecturaux (BIT-components, testers, handlers et system constructors). L'approche BIT n'impose aucune architecture de composant particulière (cf. chapitre 9). C'est pourquoi seules les interfaces et les types de données, ainsi que certaines constantes sont définies de manière précise. La documentation résultante du projet Component+ fournit tous les détails sur ces éléments. Le lecteur est renvoyé à ces documents (notamment le « Vade Mecum », véritable mode d'emploi didactique [VM03]) pour plus de détails.

Application spécifique aux composants COTS

Nous examinons dans cette partie le cas particulier des composants COTS. D'une part il s'agit de la partie du projet dont notre équipe a eu la responsabilité, et de plus, c'est lorsque ces composants

sont utilisés dans une approche CBSE que le besoin de fonctionnalités de test est le plus critique. Mais commençons tout d'abord par rappeler la définition d'un composant COTS [MeOb01] :

"[a COTS component is…] A product that is:
- *Sold, leased, or licensed to the general public,*
- *Offered by a vendor trying to profit it,*
- *Supported and evolved by the vendor, which retains the intellectual property rights,*
- *Available in multiple, identical copies,*
- *Used without internal modification by a consumer".*

Retenons que la caractéristique principale d'un composant COTS est l'impossibilité, pour son utilisateur, de le modifier d'une quelconque manière. Ceci exclut par exemple l'ensemble des activités classiques de test basées sur l'exploration du code, et pose le problème de la vérification du comportement du composant par rapport aux spécifications du client. Nous croyons que l'avenir des composants COTS tient dans leur capacité à être configuré et testé en situation par leur client. C'est pourquoi nous avons concentré nos efforts sur l'application de l'approche BIT à ce type de composants.

L'idéal serait que chaque composant COTS livré par un vendeur dispose d'une palette de fonctionnalités de test. Ce type de composants COTS pourrait alors être considéré de la même manière que les composants testables en général, et s'intégrerait parfaitement dans l'approche BIT. Malheureusement il n'en est rien, ou du moins pas encore. Un composant COTS qui ne dispose que d'interfaces fonctionnelles est un piètre élément observable, et encore moins contrôlable en comparaison d'un composant BIT. Nous nous sommes donc attachés à montrer qu'il était possible, dans une certaine mesure, de fournir ces fonctionnalités de test à un composant COTS qui en serait dénué. Faire un composant BIT d'un composant COTS implique naturellement que ce composant puisse être un minimum « manipulé », grâce par exemple à des capacités d'introspection du langage de programmation utilisé, ou bien encore par la présence de méta-information, c'est à dire que le composant soit porteur d'information sur lui-même, et accessibles à l'utilisateur. Les langages modernes comme Java, Python ou C#, disposent de mécanismes d'introspection, qui permettent, à partir du code exécutable, d'obtenir par exemple la liste des méthodes fournies. Considérés par certains comme une violation du principe d'encapsulation, ces mécanismes n'en sont pas moins fort utiles pour notre problématique. Ce sont les mécanismes de réflexion de Java qui nous ont notamment permis de développer un générateur automatique de BIT code pour composant COTS détaillé dans la section suivante.

3 Une technologie disponible en Java : la librairie BIT/J

Introduction

Nous sommes actuellement en phase de finalisation de la librairie BIT/J. Nous avons développé un outil de génération automatique qui, à partir d'un composant Java , génère une partie importante du code des fonctionnalités de test ainsi qu'une interface de test permettant de manipuler le composant à distance à l'aide de la technologie JMX [SUN02]. L'utilisation du générateur est détaillé à la fin de cette section (cf. Figure 6). Le générateur produit quatre fichiers : le composant BIT, le testeur et deux fichiers nécessaires à JMX pour le fonctionnement du test à distance, l'interface JMX et l'agent JMX. L'utilisateur doit ensuite finaliser le composant BIT en fonction de sa politique de

test. Nous présentons dans la sous-section suivante la librairie elle-même avant de présenter l'outil de génération de code BIT dans la sous-section suivante.

La librairie BIT/J

Nous considérons un composant comme un agrégat de sous-composants qui sont les implémentations des opérations qu'il fournit. En Java, un tel composant est réalisé la plupart du temps via une classe qui possède des champs dont les types sont de ceux de ses sous-composants et ce, récursivement. La Figure 3 décrit la micro-architecture dans laquelle un composant anonyme est connecté avec les classes prédéfinies de l'approche BIT.

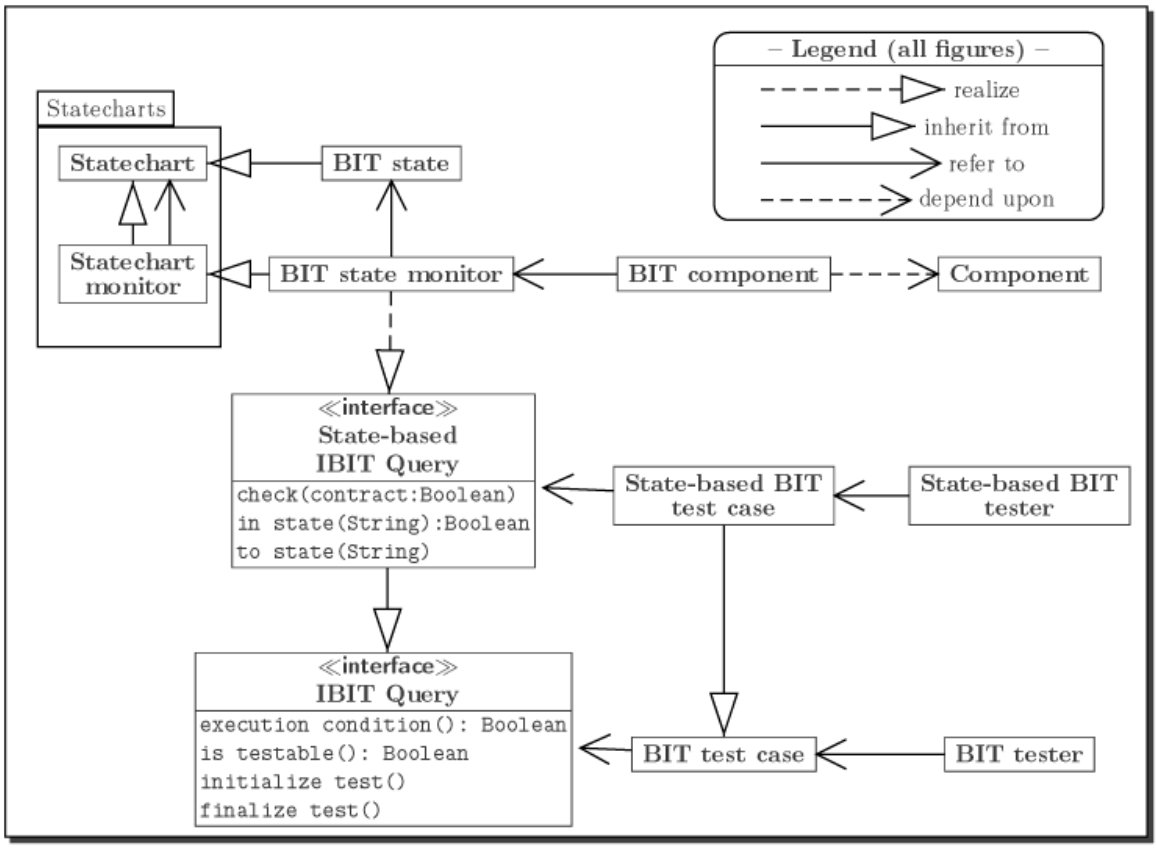

Fig. 3 - Version étendue (basée état) de la librairie

Un composant BIT (BIT component) est construit de telle sorte qu'il acquiert toutes les propriétés du composant lui-même (Component). Dans la Figure 3, une dépendance UML est utilisée dans le but de laisser le choix d'un mécanisme de programmation adéquat (par exemple, l'héritage est souvent utilisé). L'interface IBITQuery, regroupe l'ensemble des opérations qui sont systématiquement utilisées dans un cas de test BIT (BIT test case), lui-même systématiquement utilisé par un testeur BIT (BIT tester). Cette interface est fondamentale et minimale car c'est à travers elle qu'un composant COTS se transforme en composant testable.

Nous avons introduit dans notre librairie une version étendue de cette interface, State-based IBITQuery, qui, en réutilisant une librairie existante implémentant les *Statecharts* [Harel], permet

l'expression du comportement du composant. A partir d'une représentation du comportement du composant sous forme d'un automate à états (un statechart de préférence pour bénéficier pleinement des capacités de la librairie), il est en effet possible de tester et de configurer le composant selon les besoins. Bien que de telles spécifications soient communes dans les systèmes en temps réel par exemple, on ne peut pas toujours en fournir. Ainsi, un travail de ré-ingénierie est parfois nécessaire pour extraire des spécifications comportementales à partir d'un composant existant.

BIT test case est une classe Java qui a figé des protocoles de test, dont le rôle est de : "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 BIT tester permet de développer des scénarios de test : séquences, résultats attendus, politique de séquencement.

Par rapport à l'architecture précédemment décrite, la librairie BIT/J n'implémente pas la notion d'Handler. L'interprétation des résultats de test est une activité très dépendante de l'application et assez peu d'éléments de cette activité sont généralisables.

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, les composants COTS, achetés afin de répondre à des besoins spécifiques, peuvent être comparés sur la base du même cadre de test. Un inconvénient de notre approche est que les sous-composants sont des entités entièrement encapsulées, et comme telles, il est difficile d'analyser et de déterminer des diagnostics à un niveau profond d'agrégation. Nous ne traitons pas par exemple de la composition des parties BIT de sous-composants testables. Mais ceci n'est pas gênant, étant donné que nous considérons surtout des COTS, et donc des composants « boîte noire » [Barb03].

L'outillage de la librairie BIT/J

Un testeur graphique distribué

En complément de l'implémentation proprement dite des concepts BIT, notre librairie fournit également la possibilité de prendre en compte la phase de test dans un environnement distribué. Cette fonctionnalité nous a en effet paru essentielle car les composants sont souvent déployés sur des systèmes distants. En utilisant la technologie JMX de Sun, nous avons réalisé une extension du testeur de la librairie pour le test à distance (cf. Figure 4).

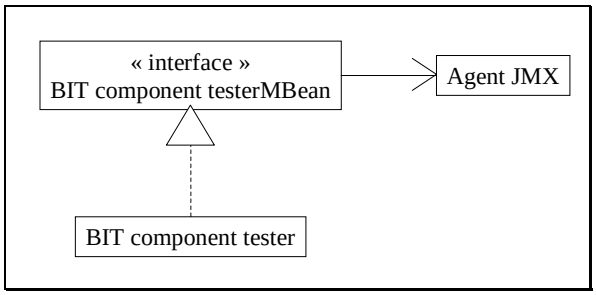

Figure 4 – Testeur distribué

Cette technologie supporte le concept de *Managed Bean* (MBean) qui est un composant capable d'être administré à distance (*monitoring*, prise de contrôle du composant, …). Nous avons donc étendu le testeur BIT sous la forme d'un MBean. Nous fournissons également l'agent JMX nécessaire à la communication entre les machines distantes. L'agent, déployé sur le même système que le composant, agit comme un serveur qui transmet une vue du testeur aux différents clients qui

y sont connectés. Notre implémentation utilise le protocole HTTP et représente les données au format HTML ce qui permet de tester le composant à partir d'un simple navigateur Web (cf. Figure 5)

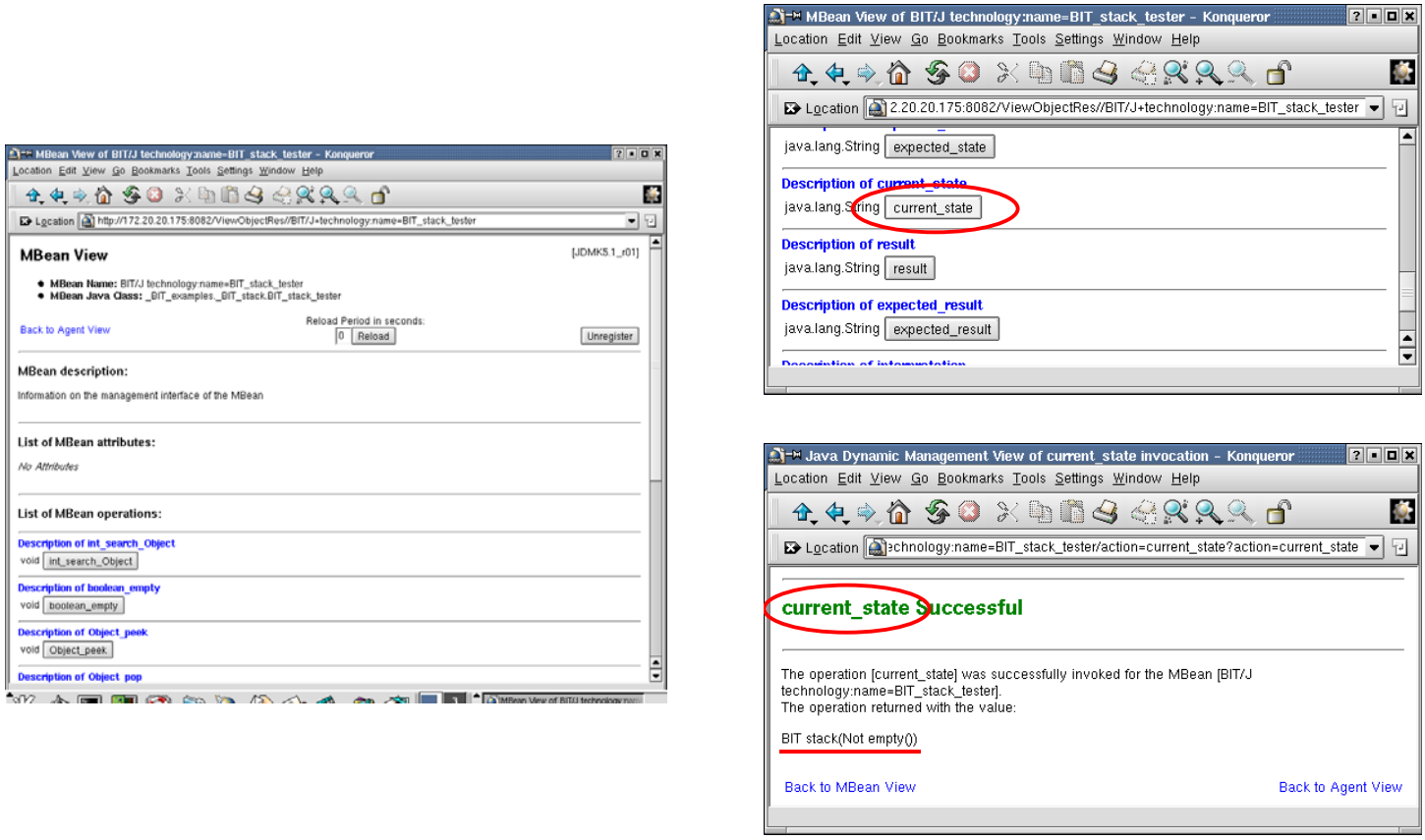

Figure 5 - Exécution d'une opération de test via un navigateur Web[2]

Un générateur automatique de code

Afin de faciliter l'utilisation de la librairie, nous avons réalisé un générateur automatique de code. A partir d'un composant dont on veut accroître la testabilité, il génère le squelette du composant BIT correspondant et le testeur graphique distribué associé que nous avons présenté dans la section précédente. Il laisse seulement l'écriture à la charge du développeur, de ce que l'on veut tester mais pas le protocole de test lui-même prédéfini. La librairie est ainsi plus efficace à utiliser ce qui est important dans le cadre des développements rapides propres au CBSE.

Tout comme la librairie, ce logiciel a été implémenté en Java. Le choix de ce langage revêt plusieurs avantages. Tout d'abord, l'interface graphique du générateur utilise l'architecture d'interface utilisateur graphique Swing de la plate-forme Java. Swing est une architecture portable qui peut s'exécuter sur n'importe quel système possédant une machine virtuelle Java (JVM). Ainsi, les développeurs utilisant notre logiciel ne sont pas contraints à un système particulier. Un autre

[2] L'exemple choisi est la Stack de Java revue via un composant appelé BIT_stack.

avantage de ce choix réside dans le mécanisme de réflexion de Java. Le mécanisme de réflexion offre aux programmes Java une faculté d'introspection qui permet d'analyser dynamiquement le code compilé de composants Java (classe prédéfinie de la librairie Java : *Class*). Dans le générateur de code, nous avons donc utilisé la réflexion Java pour analyser le composant à traiter et créer sur mesure le composant BIT correspondant ainsi que son testeur JMX.

Un exemple pas à pas

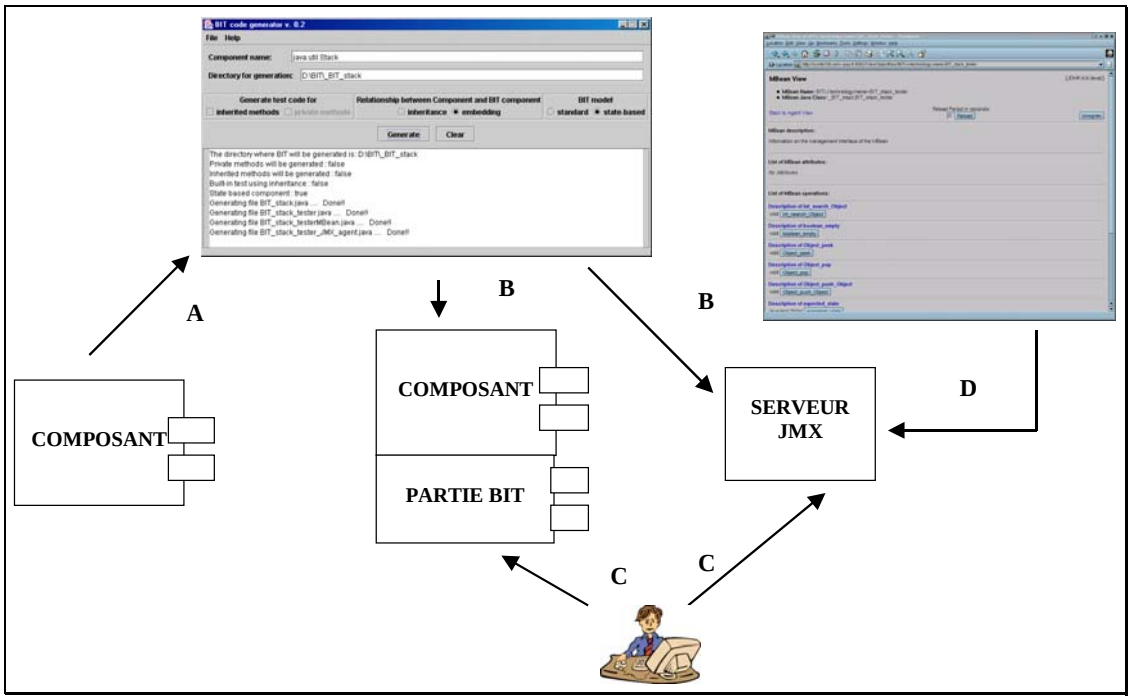

Figure 6. - Création d'un composant BIT et de son testeur

La Figure 6 illustre le processus de génération du composant testable et du testeur graphique distribué associé que nous avons présenté dans la section précédente. Le processus de mise en œuvre de la technologie BIT se déroule globalement en quatre étapes. Nous illustrons ces étapes par la construction d'une version BIT du composant Stack de Java. Le composant Stack est distribué dans le SDK de Java sous sa forme compilée sans son code source. Nous considérons donc le composant Stack comme un composant sur étagère.

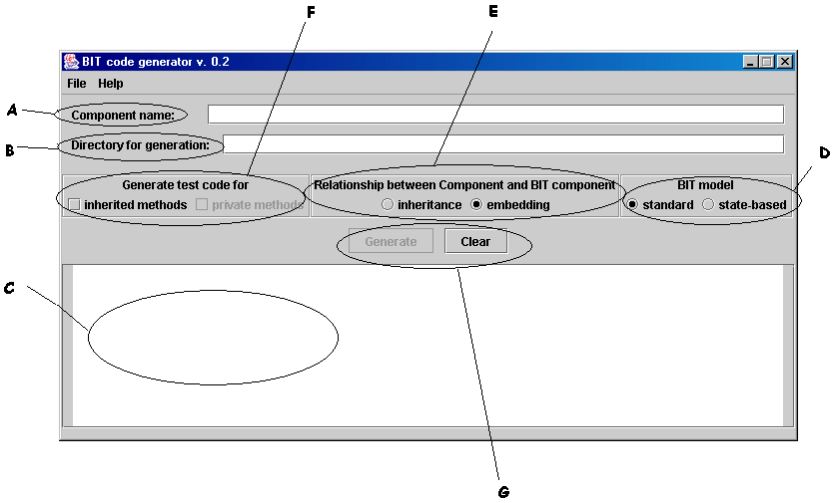

Figure 7. Interface graphique du générateur

La première étape consiste à lancer le générateur et à le configurer en lui spécifiant le composant à traiter (Figure 7-A) et les options de génération (Figure 7-D,E,F). Trois options peuvent être sélectionnées:

- La première (D) concerne le choix du modèle BIT (standard ou basé état). Le modèle BIT standard est l'approche la plus simple (interface de test minimale). Le modèle BIT basé état est le modèle implémentant l'interface permettant de décrire le comportement du composant à l'intérieur du composant BIT et de le tester.

- La seconde option (E) concerne le type de relation entre le composant et sa version BIT. Deux types de relation sont possibles : le composant BIT peut hériter du composant ou l'encapsuler comme un attribut classique.

- La troisième option (F) concerne la possibilité de prendre en compte ou non les opérations héritées du composant initial afin de les tester (cas où une opération fournie par l'interface est héritée).

Dans notre exemple du composant Stack, nous avons sélectionné le modèle BIT basé état, l'option d'encapsulation du composant et nous n'accédons pas aux opérations héritées par le composant (Figure 8).

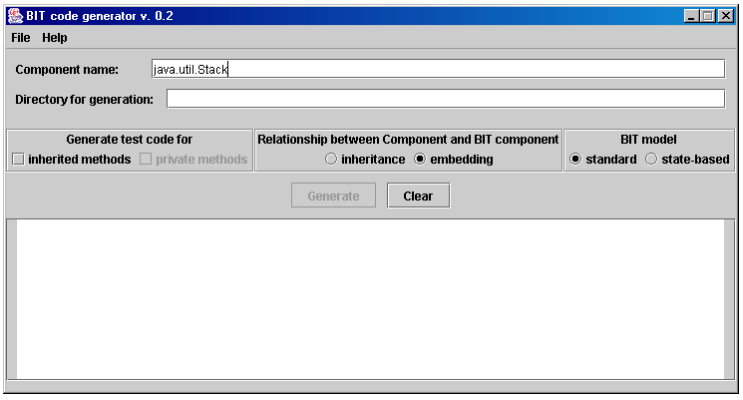

Figure 8. Configuration du générateur pour le composant Stack

La seconde étape consiste en la génération elle-même. Une fois le composant cible et le répertoire de destination des fichiers générés (Figure 7-B) définis, le bouton « Generate » (Figure 7-G) devient activable. Le résultat du processus de génération et les fichiers générés apparaissent dans la zone de texte (Figure 7-C). En appuyant sur ce bouton, la génération est exécutée et quatre fichiers sont créés :

BIT_stack.java : le composant BIT

BIT_stack_tester.java : le testeur JMX

BIT_stack_tester_JMX_agent.java : l'agent JMX nécessaire pour utiliser le testeur

BIT_stack_testerMBean.java : l'interface JMX spécifiant les opérations qui peuvent être testées.

Lors de la troisième étape, le développeur modifie et complète les fichiers générés (Figure 7-C). Dans le cas d'un modèle BIT basé état comme dans notre exemple de la Stack, le développeur doit décrire l'automate d'états du composant BIT dans le squelette généré (Figure 9).

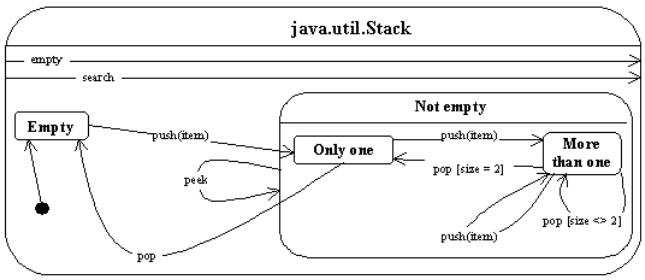

Figure 9. Automate d'états du composant Stack

Pour le composant Stack, quatre états sont définis : *Empty*, *Only one*, *More than one* et *Not empty*. Ces états doivent être déclarés dans la méthode initialisant le comportement du composant (init_behavior).

```
protected void init_behavior() throws Statechart_exception
{
/* state defitions and formal relationships here */

_Empty = new BIT_state("Empty");
_Only_one =new BIT_state("Only one");
_More_than_one = new BIT_state("More than one");

_Not_empty = (BIT_state) (_Only_one.xor(_More_than_one)).name("Not empty");

_BIT_stack = new BIT_state_monitor(_Empty.xor(_Not_empty),"BIT stack");

_Empty.inputState();

}
```

Not empty est un état composé des états *Only one* et *More than one*. Le composant Stack ne doit jamais être dans l'état *Not empty* et dans l'état *Empty* simultanément. Le moniteur d'états (state monitor) _BIT_stack est déclaré pour vérifier cette propriété. Enfin, le composant est initialisé dans l'état *Empty* (inputState).

L'interface de test est générée automatiquement, elle ne nécessite aucune modification.

La troisième partie du squelette du composant BIT est composé d'opérations correspondant aux opérations du composant original. Chaque opération générée comprend un appel à l'opération d'origine. Dans le cas basé état, le développeur doit coder les transitions spécifiées par l'automate du composant. Par exemple, pour l'opération *push* du composant Stack, on définit trois transitions : *Empty* vers *Only one*, *Only one* vers *More than one* et *More than one* vers *More than one*.

```
public java.lang.Object push(java.lang.Object o1) throws
Statechart_exception {
java.lang.Object result = _stack.push(o1);
/* state transitions here: _BIT_stack.fires(fromState,
toState); */

_BIT_stack.fires(_Empty, _Only_one);
_BIT_stack.fires(_Only_one, _More_than_one);
_BIT_stack.fires(_More_than_one, _More_than_one);

_BIT_stack.run_to_completion(); // cycle d'interprétation ininterruptible
return result;
}
```

Dans cette phase, le développeur peut également ajouter des opérations de test spécifiques. Enfin, il décrit quelles opérations seront manipulables via l'interface JMX (fichier BIT_stack_testerMBean.java pour la Stack).

La quatrième étape (Figure 7-D) est l'étape d'exécution des tests. Pour ce faire, le développeur lance le serveur JMX et se connecte au composant BIT via un simple navigateur Web. Il peut le faire localement ou depuis une machine distante (applications embarquées ou distribuées).

Un exemple plus complet de mise en œuvre peut être trouvé dans [BeBB04].

Conclusion et enjeux

Nous avons abordé dans ce chapitre le test de composants logiciels. Nous avons souligné l'importance de doter les composants de capacités de test. Les comportements individuels des composants sont souvent inclus dans leur partie cachée. Une spécification (si possible formelle) des comportements est donc primordiale pour rendre effectif le test de composant. Celle-ci n'est pas toujours disponible, et est le plus souvent insuffisante si le composant ne fournit pas d'outil spécifique à son évaluation en situation. Le *Built-In Test* que nous avons longuement détaillé dans ce chapitre, est une technique pour aider à créer la confiance nécessaire au plein essor du CBSE. Les vendeurs peuvent équiper leurs composants avec cette technologie afin de les tester directement, spécialement lorsque leur environnement de déploiement est inconnu ou changeant. Cette approche améliore la puissante idée de la « conception par contrat » de Meyer en permettant la description d'assertions (pré-conditions, post-conditions et invariants) basées sur les états complexes parallèles, et parfois concurrents, du composant qui s'articulent largement sur les sous-composants. Nous avons illustré plus particulièrement dans ce chapitre l'application que nous faisons de la technologie BIT aux composants sur étagères, les composants COTS. La spécificité

des composants COTS justifie pleinement la nécessité pour les composants logiciels de disposer de fonctionnalités de test.

Notre approche peut paraître limitée par le fait qu'elle ne s'applique qu'à des composants possédant des capacités d'introspection (mécanisme de réflexion pour la librairie BIT/J par exemple). Il faut noter que d'une part c'est un minimum pour pouvoir manipuler un élément « boîte noire », qui de plus est extérieur, et que d'autre part, l'introspection est maintenant un mécanisme disponible dans de plus en plus de langages (Java, Python, C#), ou d'infrastructures (EJB, CCM, .NET). Les intergiciels réflexifs ont été d'ailleurs récemment définis comme une priorité du 6ème programme cadre européen sur les systèmes distribués. De plus, comme nous l'avons illustré par l'existence d'un générateur automatique, cette approche assure que les protocoles de test sont écrits une fois pour toute et que 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.

Pour ce qui est des leçons à retenir de l'utilisation du *Built-In test*, les expériences menées sur des projets industriels pilotes démontrent l'efficacité de l'approche pour la phase d'intégration, où 50% environ d'effort est économisé. La technologie est un peu trop récente pour faire des bilans à encore plus grande échelle. Il y a en fait deux manières de considérer le BIT : soit les fournisseurs de composants 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 ajoutent de la valeur ajoutée à leurs composants COTS en les dotant de fonctionnalités de test paramétrables. C'est le cas idéal, mais le moins réaliste pour l'instant. Nous avons travaillé à fournir des solutions au deuxième cas, en permettant de générer automatiquement des composants testables pour notre librairie BIT/J à partir de composants ordinaires (mais supportant l'introspection).

Pour résumer les problèmes restant à traiter en la matière, et qui restent donc des « challenges » pour nos recherches futures dans le domaine, nous retiendrons :

- la substitutabilité des composants ou comment automatiser les processus de choix et de remplacement (défaillance, inadéquation…). Plus généralement, l'administration de composants devient une activité d'ingénierie logicielle de plus en plus importante. La technologie BIT est indéniablement un outil s'inscrivant dans cette évolution générale ;

- l'évaluation distante de composants en environnements hétérogènes dont notamment les applications basées sur la technologie sans fil. Là encore, l'instabilité d'applications hautement distribuées demande l'intégration de composants à très haute confiance et adaptatifs aux situations. Là encore, la technologie BIT est un outil incontournable.

Les auteurs tiennent à remercier leur collègues du projet Component+ pour leur contribution à ces travaux.

Bibliographie

[Atki01] Atkinson, C., et al. "Component-Based Product-Line Engineering with UML", Addison-Wesley, London, 2001.

[BaBB03] Barbier, F., Belloir, N., Bruel, J.M., Incorporation of Test Functionality in Software Components. 2nd Intl. Conference on COTS-Based Software Systems, Ottawa, Canada, 10.-12. February 2003.

[Barb03] Franck Barbier. Built-in-test vade mecum – part V, Built-in Test for COTS-based Products. Component+ Internal Report. 2003. Available at http://www.component-plus.org/pdf/reports/bitvm_pv.pdf.

[BeBB04] Nicolas Belloir, Jean-Michel Bruel, and Franck Barbier. Intégration du test dans les composants logiciels. *Numéro spécial de la revue L'Objet,* X(1):89-102, 2004.

[Bohr01] Adrita Bhor. Software Component Testing Strategies. 2001. Available at http://www-netra.ics.uci.edu/~abhor/ics221/comp_test.htm.

[Comp01] Component+ Project Technical Report, “Built-in Testing for Component-based Development”, http://www.component-plus.org.

[Gao00] Jerry Gao. Component Testability and Component Testing Challenges, 2000. Available at http://www.sei.cmu.edu/cbs/cbse2000/papers/18/18.html.

[HaLS99] Mary Jean Harrold, Donglin Liang, Saurabh Sinha, An Approach To Analyzing and Testing Component Based Software, Proceedings of the First International ICSE Workshop on Testing Distributed Component-Based Systems, Los Angeles, CA, May 1999.

[HCSS01] National Coordination Office for Information Technology Research and Development, January 2001. High Confidence Software and Systems Research Needs, available from: http://www.ccic.gov/iwg/hcss.html.

[JéDL01] Jean-Marc Jézéquel, Daniel Deveaux and Yves Le Traon. « Reliable Objects : Lightweight Testing for OO Languages ». IEEE Software, July/august 2001.

[MeNi98] Theo Dirk Meijler and Oscar Nierstrasz, "Beyond Objects: Components", pp. 49-78, in Michael P. Papazoglou and Gunter Schlageter, eds. "Cooperative Information Systems -- Trends and Directions", Academic Press, 1998.

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

[MeOb01] Meyers and Oberndorf. *Managing Software Acquistion – Open Systems and COTS Products,* Addison-Wesley, 2001.

[SUN02] Sun, “Java Management Extensions (JMX) 1.2. Available at : http://java.sun.com/products/JavaManagement, 2002.

[Szyp99] Szyperski, C., Component Software: Beyond Object-Oriented Programming, Addison Wesley, 1999.

[VM03] Built-in-test Vade Mecum. Component+ Internal Report. 2003. Available at http://www.component-plus.org/pdf/reports/.