Test de conformité basé sur l'architecture logicielle

Elena Leroux*, Flavio Oquendo* et Qin Xiong*

*IRISA, Université de Bretagne-Sud, Vannes, France
{elena.leroux | flavio.oquendo | qin.xiong}@irisa.fr

Résumé. Au cours des deux dernières décennies, l'architecture logicielle a joué un rôle central dans le développement des systèmes logiciels. Il fournit une description de haut niveau pour les systèmes complexes de grande taille en utilisant des abstractions appropriées pour les composants du système et pour leurs interactions. Dans notre travail, l'architecture logicielle est décrite en utilisant un langage de description architecturale (*Architecture Description Language* ou ADL) formel appelé π-ADL-C&C. L'un des objectifs de cet ADL est de permettre la validation formelle d'un système implémenté, par rapport à son modèle architectural. Dans l'article, nous proposons une approche fondée sur le test de conformité pour valider l'implémentation du système par rapport à son architecture. Les tests architecturaux sont dérivés à partir d'un système de transitions, représentant la structure de l'architecture d'un système et de ses comportements, et sont exécutés sur le système sous test. Pour illustrer notre approche, nous utilisons l'exemple de la machine à café.

1 Introduction

Au cours des dernières années, une croissance continue, en taille et en complexité, des systèmes logiciels et matériels a été observée. Les problèmes, liés à l'écriture du code lors du développement d'un système, qui étaient importants dans le passé, par exemple le choix de la structure de données et des algorithmes, sont devenus moins importants que ceux liés à la conception du système. Cela est dû, non seulement à la quantité accrue de code, mais aussi à la nécessité de distribuer différents composants du système et de les faire interagir de manière complexe. Pour faire face à ces problèmes et passer au niveau d'abstraction supérieur pour concevoir et développer un système logiciel, la notion d'architecture logicielle est apparue. L'architecture logicielle a rapidement été considérée comme une sous-discipline importante du génie logiciel, voir Shaw et Garlan (1996), car elle a permis aux développeurs de systèmes logiciels : (1) d'abstraire les détails des différents composants du système, (2) de représenter le système comme un ensemble de composants avec des connecteurs associés qui décrivent les interactions (a) entre les composants et (b) entre les composants et l'environnement, et (3) de guider la conception du système et son évolution. Afin de décrire l'architecture logicielle d'un système, un ensemble de langages formels et semi-formels a été proposé, voir Malavolta et al. (2013); Medvidovic et Taylor (2000). Ces langages de description architecturale

permettent de spécifier une architecture selon différents points de vue. Les deux points de vue suivants de la perspective d'exécution sont fréquemment utilisés dans le domaine de l'architecture logicielle.

Le point de vue structurel est spécifié en termes : (1) de composants (*c'est-à-dire,* les unités de calcul d'un système), (2) de *connecteurs* (*c'est-à-dire,* les interconnexions entre les composants pour réaliser les interactions entre eux) et (3) de *configurations* de composants et de connecteurs. Ainsi, une description architecturale, du point de vue structurel, doit fournir une spécification architecturale formelle, en termes de composants et de connecteurs, et expliciter la façon dont ils sont composés ensemble.

Le point de vue comportemental est spécifié en termes : (1) d'actions qu'un système exécute où dans lesquelles il participe, (2) de relations entre les actions pour spécifier les comportements et (3) de comportements de composants et de connecteurs, et la façon dont ils interagissent.

Un des défis d'ADL est sa capacité de réaliser la validation et/ou la vérification des systèmes logiciels (1) très tôt dans leur cycle de vie, ainsi que (2) tout au long du processus de leur développement. Le langage π-ADL, voir Oquendo (2004), qui est un langage de description formel des architectures logicielles d'un système en cours de développement, a été conçu afin de répondre à ce défi. Il permet de concevoir des spécifications architecturales qui peuvent être exécutées sur une machine virtuelle de π-ADL. Cela permet la validation du système logiciel par la simulation, mais aussi par le test comme décrit dans l'article.

L'analyse et la validation, en utilisant par exemple les techniques de tests, d'un système logiciel jouent un rôle crucial dans le processus de développement du système. C'est l'une des raisons de l'intérêt grandissant pour l'utilisation des modèles architecturaux afin de tester les comportements des systèmes par rapport à leur spécification architecturale. Le *test logiciel*, voir Myers (1979), est un processus de la vérification dynamique des comportements du système par l'observation de l'exécution du système sur un cas de test sélectionné. Plusieurs contributions ont été proposées pour résoudre le problème de la validation de systèmes logiciels par rapport à l'architecture logicielle par le biais de tests, notamment Richardson et Wolf (1996); Stafford et al. (1997); Bertolino et Inverardi (1996); Jin et Offutt (2001); Muccini et al. (2004); Bertolino et al. (2003); Muccini et al. (2005c); Harrold (1998). Nous discutons ses travaux dans la Section 5.

Dans cet article, nous nous intéressons au *test de conformité basé sur un modèle*, voir Beizer (1990); Tretmans (1992). Cette technique permet de générer des cas de test à partir d'un modèle représentant le comportement d'un système logiciel afin de vérifier que ce système se comporte conformément à sa spécification. Nous utilisons le système de transitions symboliques à entrées-sorties (IOSTS) comme modèle. Ce modèle est généré automatiquement à partir d'une spécification architecturale formelle écrite en π-ADL. Le but de ce papier est de proposer une approche pour la validation d'un système logiciel en utilisant l'architecture logicielle et d'illustrer sa faisabilité avec un exemple simple.

Le reste de cet article est structuré comme suit : la Section 2 présente le langage π-ADL utilisé pour la conception d'architectures logicielles. Elle présente également un

exemple simple, qui est utilisé tout au long de cet article dans le but d'illustrer notre approche. La Section 3 décrit brièvement le formalisme des IOSTS, qui sont utilisés pour modéliser (1) une spécification architecturale d'un système logiciel écrite en π-ADL, et (2) les cas de tests abstraits générés à partir de cette spécification. La Section 4 présente notre approche en expliquant comment générer des cas de test à partir d'une spécification architecturale π-ADL et comment les exécuter sur un système logiciel boîte noire sous test. La Section 5 résume nos travaux, les positionne par rapport aux autres travaux réalisés dans le domaine du test de conformité basé sur l'architecture logicielle et donne un bref aperçu des travaux connexes. La Section 6 termine le papier avec des remarques sommaires.

2 π-ADL – un langage de description d'architecture

Dans cette section, nous présentons brièvement le langage π-ADL, que nous utilisons pour la description de l'architecture d'un système en cours de développement et nous l'illustrons avec l'exemple d'une machine à café.

2.1 Aperçu du langage π-ADL

Le langage π-ADL, voir Oquendo (2004), conçu dans le cadre du projet européen ArchWare, est un langage formel qui possède une base théorique solide. Il est fondé sur le π-calcul typé d'ordre supérieur, voir Sangiorgi (1992). Il permet la conception d'architectures logicielles dans la perspective de leur exécution. En outre, le langage π-ADL possède une machine virtuelle permettant (1) l'exécution des spécifications architecturales et, par conséquent, (2) leur validation par simulation. Dans la suite, nous expliquons brièvement comment le langage π-ADL peut être utilisé pour la définition formelle d'une architecture logicielle. Une architecture π-ADL est décrite en termes de composants, de connecteurs et de leur composition.

Les composants sont représentés par des ports externes et leur comportement interne. Leur rôle architectural consiste à spécifier les éléments de calcul d'un système logiciel. L'accent est mis sur le calcul pour fournir la fonctionnalité du système. Les ports sont décrits en terme de connexions entre un composant et son environnement. Leur rôle architectural est de mettre en place des connexions fournissant une interface entre le composant et son environnement. Les protocoles peuvent être utilisés par les ports et entre les ports.

Les connecteurs sont des points d'interaction de base. Leur rôle architectural est de fournir des canaux de communication entre deux éléments architecturaux. Un composant peut envoyer ou recevoir des valeurs via des connexions. Elles peuvent être déclarées comme des connexions de sortie (les valeurs peuvent seulement être envoyées), des connexions d'entrée (les valeurs peuvent seulement être reçues) ou des connexions d'entrée-sortie (les valeurs peuvent être envoyées ou reçues).

Du point de vue d'une boîte noire, seulement les ports des composants et des connecteurs, ainsi que les valeurs envoyées ou reçues par une connexion sont observables. Du point de vue d'une boîte blanche, les comportements internes sont également observables.

π-ADL est une famille de différents ADL. Le langage π-ADL-C&C permet de décrire une architecture à un niveau relativement abstrait. Il permet la conception rapide d'architectures par l'utilisation des notions de composant et de connecteur. Le langage π-ADL-Spec est une forme canonique de π-ADL. Enfin, le langage π-ADL.NET est un ADL de bas niveau, qui rend possible l'exécution d'une spécification architecturale, étant donné qu'elle est équipée d'une machine virtuelle.

2.2 Exemple

Dans cette section, nous présentons l'exemple simple d'une machine à café, qui sera utilisé tout au long de l'article. La Fig. 4 montre l'architecture abstraite de la machine à café en termes de composants et de connecteurs.

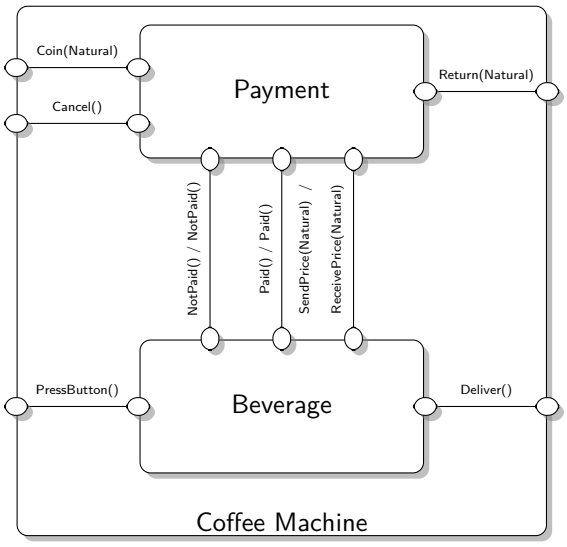

FIG. 1 – *L'architecture de la machine à café.*

Cette machine à café accepte les pièces (par le connecteur *Coin(Natural)*), la commande d'une boisson (par le connecteur *PressButton()*) et la demande d'annulation de commande (par le connecteur *Cancel()*), et ensuite, soit délivre la boisson (par le connecteur *Deliver()*), soit rend la monnaie (par le connecteur de *Return(Natural)*). Elle est représentée par deux composants : *Payment* et *Beverage*.

Le client de la machine à café, en utilisant l'interface de la machine, envoie une demande pour une boisson qui sera reçue par le composant *Beverage*. L'objectif de ce composant est (1) de stocker l'information sur la disponibilité et le prix d'un café, (2) d'attendre jusqu'à ce que le bouton de choix de la boisson soit enfoncé, (3) de communiquer le prix au composant *Payment*, (4) de préparer un café et (5) de le remettre à un client. Le composant *Beverage* sert le café quand les deux conditions suivantes sont remplies : premièrement, un client a payé suffisamment (cette information doit être reçue du composant *Payment*) et deuxièmement, le café n'est pas en rupture de stock. Si la première condition n'est pas satisfaite, le composant *Beverage* attend une autre demande de café, puis vérifie à nouveau si le paiement est suffisant. Si la seconde condition n'est pas satisfaite, alors la livraison de café est impossible et le composant *Beverage* est bloqué.

Les demandes de paiement et d'annulation d'une commande en provenance de l'utilisateur de la machine à café sont traitées par le composant *Payment*. Ce composant permet (1) de mémoriser la somme d'argent déjà payée par le client, le nombre de pièces introduites dans la machine à café et le prix d'un café reçu à partir du composant *Beverage*, (2) de communiquer les informations relatives au paiement suffisant/insuffi-

sant au composant *Beverage*, (3) rendre la monnaie si le bouton *Cancel* a été enfoncé ou si le client a introduit plus de pièces que le nombre maximal autorisé par la machine à café et (4) de rendre la différence entre le prix et le montant payé dans le cas d'une livraison de café.

Notez que les composants *Beverage* et *Payment* communiquent non seulement avec leur environnement, mais aussi entre eux. En effet, le composant *Beverage* envoie le prix d'un café par le connecteur *SendPrice(Natural)* au composant *Payment*. Ce dernier reçoit le prix au moyen du connecteur *ReceivePrice(Natural)*. En outre, le composant *Payment* indique au composant *Beverage* si le client a payé assez ou non en utilisant les connecteurs *Paid()* et *NotPaid()*.

2.3 Description d'architecture en utilisant π-ADL-C&C

Dans la section précédente, nous avons décrit de façon informelle la structure et le comportement de la machine à café. Dans cette section, nous expliquons comment cette structure et ce comportement peuvent être formalisés en utilisant le langage π-ADL-C&C. Nous commençons par la description des deux composants de la machine à café qui sont les composants *Beverage* (voir Fig. 3) et *Payment* (voir Fig. 2).

Le composant *Beverage* est déclaré comme une abstraction (voir ligne 1 de la Fig. 3) avec deux paramètres : (1) *cBeverageQuantity* indiquant la quantité de boisson dans la machine à café, (2) *cPrice* indiquant le prix de la boisson. Les ports externes de ce composant sont présentés sur les lignes 3-9 et décrits en termes de connexions : *PressButton*, *Paid*, *NotPaid*, et *SendPrice*, *Deliver*, où les trois premières connexions permettent de recevoir l'information de l'environnement (elles sont déclarées comme des connexions d'entrée à l'aide du mot-clé `in`) et les deux dernières permettent d'envoyer l'information à l'environnement (elles sont déclarées comme des connexions de sortie en utilisant le mot-clé `out`). Notez que, la connexion *SendPrice* permet d'envoyer une valeur de type *Natural* (voir ligne 6) afin d'être en mesure de communiquer le prix de la boisson.

Le comportement du composant *Beverage* est indiqué sur les lignes 29-31. Il est décrit comme un appel à l'abstraction *drink* avec un paramètre de valeur 0. La valeur 0 initialise la variable *vBeverageQuantity* en mémorisant la quantité de boisson déjà utilisée. Le corps de l'abstraction *drink* décrit formellement le comportement du composant *Beverage* de la machine à café, expliqué de manière informelle dans la Section 2.2. Plus précisément, le composant *Beverage* vérifie si la quantité de boisson est suffisante ou non (voir ligne 12). Dans les deux cas ci-dessus, il permet au client d'appuyer sur le bouton (voir les lignes 13 et 16), mais

(1) dans le dernier cas (la quantité de boisson est insuffisante), le composant est bloqué (voir l'appel à la même abstraction *drink* avec la même valeur du paramètre *vBeverageQuantity* sur la ligne 14), tandis que

(2) dans le premier cas (la quantité de boisson est suffisante), le composant communique le prix de la boisson en utilisant la connexion *SendPrice* (voir ligne 17), et ensuite (a) soit retourne dans son état initial (voir l'appel à l'abstraction *drink* sur la ligne 20), s'il a reçu la notification de paiement insuffisant par la connexion

NotPaid (voir ligne 19), ou (b) délivre la boisson en utilisant la connexion *Deliver* (voir ligne 23) et augmente *vBeverageQuantity* de un (voir ligne 24), s'il a reçu la notification de paiement suffisant à travers la connexion *Paid* (voir ligne 22) et revient à son état initial (voir l'appel à l'abstraction *drink* à la ligne 25).

```
1  component Payment is abstraction(cCoinNumber: Natural){
2   port is {
3      connection Coin is in (Natural).
4      connection Return is out (Natural).
5      connection Cancel is in ().
6      connection ReceivePrice is in (Natural).
7      connection Paid is out ()
8      connection NotPaid is out ()
9   }.
10  paying is abstraction(
11     cCoinNumber: Natural,
12     vPaid: location[Natural],
13     vCoinNumber: location[Natural],
14     vPrice: location[Natural]
15   ){
16    choose {
17     if vCoinNumber < cCoinNumber then {
18       via Coin receive pCoin : Natural.
19       vPaid := vPaid'+pCoin.
20       vCoinNumber := vCoinNumber'+1.
21       paying(cCoinNumber, vPaid, vCoinNumber, vPrice)
22     } else {
23       via Return send vPaid.
24       paying(cCoinNumber, location(0), location(0),
                location(0))
25     }
26    or
27      via ReceivePrice receive pPrice : Natural.
28      vPrice := pPrice.
29      paying(cCoinNumber, vPaid, vCoinNumber, vPrice)
30    or
31      via Cancel receive. via Return send vPaid.
32      paying(cCoinNumber, location(0), location(0),
              location(0))
33    or
34      if vPaid >= vPrice then {
35        via Paid send. via Return send (vPaid-vPrice).
36        paying(cCoinNumber, location(0), location(0),
              location(0))
37      } else {
38        via NotPaid send.
39        paying(cCoinNumber, vPaid, vCoinNumber, vPrice)
40      }
41    }
42  }.
43  behaviour is {
44    paying(cCoinNumber, location(0), location(0),
            location(0))
45  }
46 }
```

FIG. 2 – *Le composant Payment exprimé en π-ADL-C&C.*

```
1  component Beverage is abstraction(cBeverageQuantity :
         Natural, cPrice : Natural){
2   port is {
3     connection PressButton is in().
4     connection Deliver is out().
5     connection SendPrice is out(Natural).
6     connection Paid is in().
7     connection NotPaid is in().
8   }
9   drink is abstraction(vBeverageQuantity :
          location[Natural]){
10    if (vBeverageQuantity >= cBeverageQuantity) then{
11      via PressButton receive. drink(vBeverageQuantity)
12    }else{
13      via PressButton receive. via SendPrice send cPrice.
14      choose{
15        via NotPaid receive. drink(vBeverageQuantity)
16      or
17        via Paid receive. via Deliver send.
18        vBeverageQuantity := vBeverageQuantity'+1.
19        drink(vBeverageQuantity)
20      }
21    }
22  }.
23  behaviour is {
24    drink(location(0))
25  }
26 }
```

FIG. 3 – *Le composant Beverage exprimé en π-ADL-C&C.*

```
1  architecture CoffeeMachine is abstraction() {
2   behaviour is {
3     compose{
4       beverage is Beverage(10, 3)
5     and
6       payment is Payment(10)
7     } where {
8       payment::ReceivePrice unifies beverage::SendPrice
9       and
10      payment::Paid unifies beverage::Paid
11      and
12      payment::NotPaid unifies beverage::NotPaid
13    }
14  }
15 }
```

FIG. 4 – *L'architecture d'une machine à café exprimée en π-ADL-C&C.*

Le composant *Payment* est formellement décrit sur la Fig. 2. Sa description est similaire à celle du composant *Beverage*. Par conséquent, nous ne la détaillons pas dans cet article.

L'architecture de la machine à café est formellement décrite sur la Fig. 4. C'est une abstraction dont le comportement (voir 2-14) est composé de deux composants instanciés *Beverage(10,3)* et *Payment(10)* (voir les lignes 3-7). Ces composants communiquent via les connexions unifiées montrées sur les lignes 8-9.

3 Modèle sous-jacent pour la génération de tests

Dans cet article, nous nous intéressons au test de conformité d'un système en cours de développement par rapport à sa spécification architecturale exprimée au niveau de l'utilisateur à l'aide du langage π-ADL-C&C. Afin d'être en mesure de générer des cas de test à l'aide de l'outil STG (Symbolic Test Generator), voir Ployette et Ponscarme (2007); Clarke et al. (2002), nous traduisons automatiquement une spécification architecturale dans le modèle de bas niveau appelé IOSTS. Nous utilisons les IOSTS pour décrire les spécifications architecturales, les objectifs de test et les cas de test et supposons que l'implémentation boîte noire du système peut être décrite par un IOSTS dont seule l'interface externe est connue. La syntaxe et la sémantique formelles des IOSTS sont définies dans Zinovieva-Leroux (2004). L'explication intuitive est donnée ci-dessous en utilisant l'exemple montré sur la Fig. 5, qui représente le composant *Payement* de la machine à café. Le composant *Beverage* peut aussi être modélisé par un IOSTS comme il est montré sur la Fig. 6.

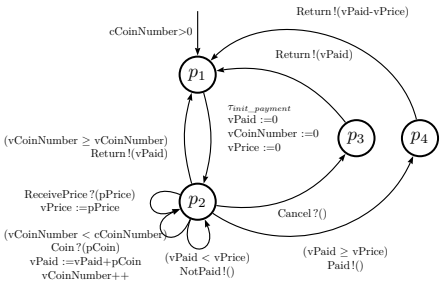

FIG. 5 – *Le composant Payment de la machine à café, modélisé par un IOSTS.*

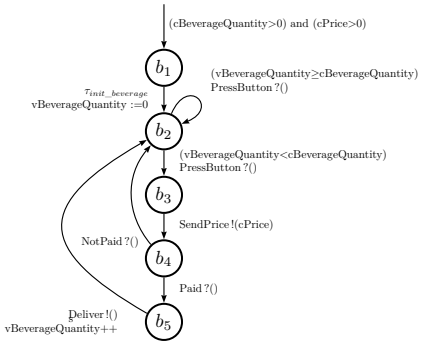

FIG. 6 – *Le composant Beverage de la machine à café, modélisé par un IOSTS.*

Syntaxe informelle. Un IOSTS est constitué de *localités*, par exemple, p_1, p_2, p_3 et p_4, où p_1 est la *localité initiale*, et de transitions. Les transitions sont étiquetées avec des *actions*, des *gardes* et des *affectations de variables*. Par exemple, la transition dont l'origine est p_2 et la destination est p_2 a la garde $(vCoinNumber < cCoinNumber)$, l'action d'entrée *Coin ?* portant le paramètre *pCoin* et deux affectations de variable : $vPaid := vPaid + pCoin$ et $vCoinNumber + +$. L'ensemble des actions est divisé en trois sous-ensembles disjoints d'actions d'*entrée*, d'actions de *sortie* et d'actions *internes*. Les actions d'entrée/sortie interagissent avec l'environnement et peuvent porter des données depuis/vers lui, tandis que les actions internes sont utilisées pour les calculs internes. Par convention, le nom des actions d'entrée (*resp.* de sortie) se terminent par " ?" (*resp.* " !"). L'IOSTS de la Fig. 5 a deux actions d'entrées : *Coin ?* et *Cancel ?*, trois actions de sorties : *Paid !*, *NotPaid !*, *Return !*, et une action interne : $\tau_{init\_payment}$.

Il fonctionne avec des données symboliques constituées de *variables*, de *constantes* et de *paramètres*. Intuitivement, les *variables* sont données pour calculer, les *constantes*

sont des constantes symboliques et les *paramètres* sont donnés pour communiquer avec l'environnement. Notez que la portée des paramètres est seulement une transition étiquetée par une action qui porte ces paramètres. Ainsi, si la valeur d'un paramètre doit être utilisée dans des calculs ultérieurs, elle doit être mémorisée en utilisant une affectation à une variable.

Sémantique informelle. Soit l'IOSTS représentant le composant *Payement* de la machine à café (voir Fig. 5) . Le paiement commence dans la localité p_1 avec une certaine valeur de la constante *cCoinNumber* satisfaisant la condition initiale $cCoinNumber > 0$, *c'est-à-dire* que, le nombre de pièces acceptées par la machine à café est strictement positif. Ensuite, il emprunte la transition étiquetée par l'action interne $\tau_{init\_payment}$, affecte les trois variables : *vPaid* qui stocke le montant déjà payé, *vCoinNumber* qui mémorise le nombre de pièces introduites dans la machine et *vPrice* qui contient le prix de la boisson, à 0, et rejoint la localité p_2. Ensuite, le composant *Payement* de la machine à café attend soit :

- une pièce de monnaie, désignée par l'action d'entrée *Coin ?* qui porte, dans le paramètre *pCoin*, la valeur de la pièce introduite. Les variables *vPaid* et *vCoinNumber* sont augmentées respectivement de *pCoin* et de 1. Notez que l'action *Coin ?* peut être exécutée seulement dans le cas où le nombre des pièces de monnaie déjà insérées est inférieur à la valeur de la constante de *cCoinNumber*. Dans le cas contraire, le composant *Payement* renvoie le montant déjà versé (par l'action de sortie *Return !(vPaid)*) et retourne à la localité initiale p_1. Ou bien,
- le prix d'une boisson, désigné par l'action d'entrée *ReceivePrice ?* qui porte dans le paramètre *pPrice* l'information par rapport au prix de la boisson. Dans ce cas, la variable *vPrice* est initialisée à la valeur de *pPrice*.

Dans les deux cas ci-dessus, la machine reste dans la localité p_2. Si le paiement est suffisant, *c'est-à-dire*, $vPaid \geq pPrice$, le composant *Payement* émet d'abord l'action de sortie *Paid !()* et se déplace vers la localité p_4. Puis, il rend (en utilisant l'action de sortie *Return !(pPrice − vPaid)*) la différence entre le montant payé et le prix d'une boisson, *c'est-à-dire* $pPrice - vPaid$, et se déplace vers la localité initiale p_1. Dans le cas contraire, le composant *Payement* envoie l'action de sortie *NotPaid !()* et reste dans la localité p_2. Notez que dans la localité p_2, l'action d'entrée *Cancel ?* peut être reçue, ce qui signifie que le bouton *Cancel* a été enfoncé. Dans ce cas, le composant *Payement* renvoie le montant déjà versé (en utilisant l'action de sortie *Return !(vPaid)*) et retourne à sa localité initiale p_1.

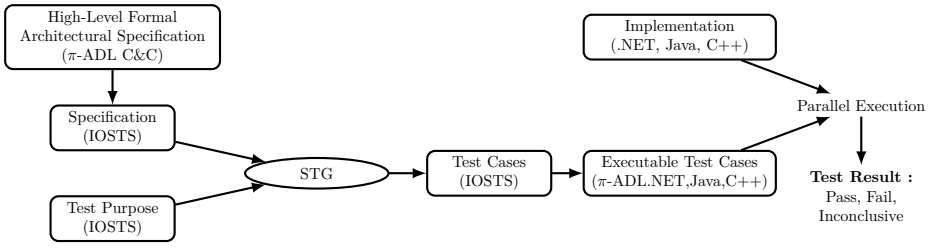

FIG. 7 – *Notre approche pour la validation architecturale.*

4 Approche

Dans cette section, nous décrivons l'approche que nous utilisons pour la validation de l'implémentation d'un système en cours de développement en utilisant une spécification architecturale de ce système. Cette approche est décrite dans la Fig. 7 et présentée ci-dessous.

4.1 De π-ADL-C&C à IOSTS

Dans cette section, nous décrivons de façon informelle la transformation d'une spécification architecturale exprimée en π-ADL-Spec dans son modèle IOSTS. Nous utilisons l'exemple du composant *Payement*, montré sur la Fig. 2 et appelé $S_{\pi\text{-ADL-C\&C}}$, afin d'illustrer cette transformation. Le résultat de la transformation est l'IOSTS représenté sur la Fig. 5 et appelé S_{IOSTS}.

1. Chaque abstraction π-ADL-C&C correspond à un modèle IOSTS. Par exemple, l'abstraction montrée sur les lignes 1-46 de $S_{\pi\text{-ADL-C\&C}}$ correspond à S_{IOSTS} qui modélise les comportements du composant *Payement* de la machine à café.
2. Les connexions d'une abstraction π-ADL-C&C deviennent les actions d'entrée/-sortie de l'IOSTS correspondant. Par exemple, les connexions de $S_{\pi\text{-ADL-C\&C}}$, *c'est-à-dire, Coin*, *Cancel* et *Return*, *Paid*, *NotPaid* (voir les lignes 3-8), sont les actions d'entrée/sortie de S_{IOSTS}.
3. Chaque préfixe d'entrée et de sortie d'une abstraction π-ADL-C&C, dont la syntaxe respective est "`via connection receive value`" et "`via connection send value`", se transforme en une transition d'IOSTS étiquetée avec une action correspondant à **connection** portant les paramètres correspondants à `value` de ce préfixe. Par exemple, le code π-ADL-C&C des lignes 17-25 correspond à deux transitions de S_{IOSTS} partant de la localité p_2 et étiquetées avec les actions *Coin ?* et *Return !*. Chaque préfixe silencieux, indiqué par le mot-clé "`unobservable`", est traduit en une transition d'IOSTS étiquetée avec une action interne. A noter que, toutes les affectations suivant un préfixe deviennent des affectations de la transition correspondant à ce préfixe. De plus, si le préfixe est entouré par la structure conditionnelle "`if(condition) then{...}`", alors sa transition correspondante, dans le modèle IOSTS, est gardée par `condition`.
4. Une séquence de préfixes d'entrée, de sortie et silencieux dans le langage π-ADL-C&C est modélisée par la séquence des transitions correspondantes dans le modèle IOSTS. Par exemple, la séquence "`via Cancel receive.via Return send vPaid`" de $S_{\pi\text{-ADL-C\&C}}$ (voir la ligne 31) est représentée par deux transitions consécutives $(p_2, Cancel?(), p_3)$. $(p_3, Return!(pPaid), p_1)$ de S_{IOSTS} (voir Fig. 5).
5. La structure "`choice`" de π-ADL-C&C permet de modéliser une localité d'un IOSTS avec plusieurs transitions sortantes. Par exemple, le code des lignes 16-41 de $S_{\pi\text{-ADL-C\&C}}$ correspond à p_2 de S_{IOSTS} et à six transitions sortantes de p_2.
6. Un appel à une abstraction dans le langage π-ADL-C&C, signifie que la transition correspondante à un préfixe précédé par cet appel, devrait être redirigé vers l'une des localités déjà créées de l'IOSTS. Par exemple, l'appel de la ligne 21 du

$S_{\pi\text{-ADL-C\&C}}$ signifie que la transition de S_{IOSTS} étiquetée par l'action d'entrée *Coin ?* devrait rester dans la même localité, alors que l'appel de la ligne 24 signifie que la transition étiquetée par l'action de sortie *Return !* devrait aller à p_1.

La composition des deux composants (abstractions) est modélisée par la composition parallèle entre deux IOSTS avec synchronisation sur les actions qui doivent communiquer entre elles. La spécification architecturale de la machine à café est le résultat de la composition entre deux IOSTS (voir Fig. 5 et Fig. 6) utilisés pour modéliser les comportements des composants *Payement* et *Beverage* de la machine à café. Cette spécification est utilisée pour dériver des cas de test, mais nous ne l'avons pas montré dans l'article en raison de sa taille.

4.2 Génération symbolique de cas de test

La génération symbolique de cas de test consiste à générer, à partir de la spécification architecturale formelle d'un système en cours de test et d'un objectif de test décrivant un ensemble de comportements à tester, un programme réactif, appelé cas de test, qui observe une implémentation du système pour détecter les comportements non-conformes, tout en essayant de contrôler l'implémentation dans le but de satisfaire l'objectif de test. L'outil STG, voir Ployette et Ponscarme (2007); Clarke et al. (2002), que nous utilisons pour la génération de cas de test, prend en entrée une spécification et un objectif de test, représentés sous la forme d'un IOSTS, puis produit un IOSTS décrivant un cas de test abstrait. Dans la Section 4.1, nous avons décrit comment obtenir la spécification sous la forme d'un IOSTS à partir de son écriture dans le langage π-ADL-C&C. Ci-dessous, nous expliquons les notions d'objectif de test et cas de test.

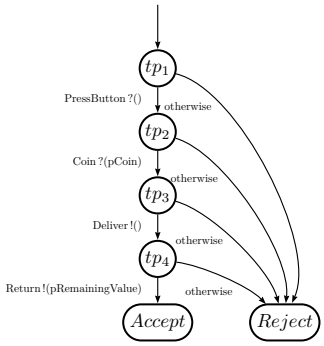

FIG. 8 – *L'objectif de test (IOSTS).*

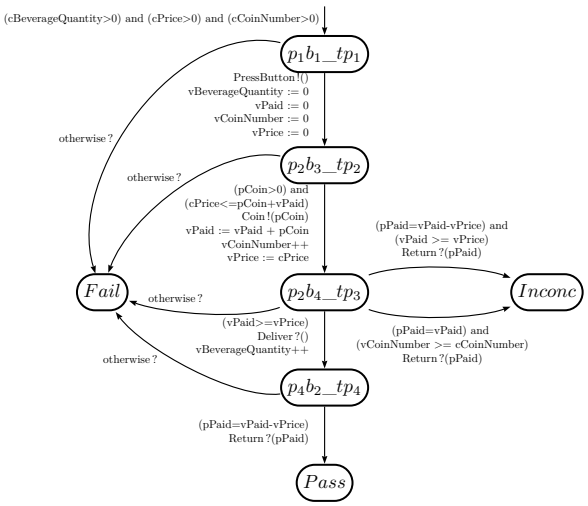

FIG. 9 – *Le cas de test (IOSTS).*

Objectif de test. Un objectif de test est utilisé pour sélectionner les comportements de la spécification qui vont être exercés par le cas de test généré. L'exemple d'un objectif

de test est montré sur la Fig. 8. Il sélectionne, à partir de la spécification de la machine à café, un cas de test qui exerce une livraison de café dans le cas où (2) la touche de choix de la boisson est actionnée et (2) une seule pièce de monnaie, qui doit être suffisante pour un paiement de café, est insérée dans la machine.

```
component TestCase is abstraction(
  cCoinNumber : Natural, // 10
  cBeverageQuantity : Natural, // 15
  cPrice : Natural) // 2
  port is {
    connection Coin is out (Natural).
    connection Cancel is out ().
    connection PressButton is out().
    connection Return is in (Natural).
    connection Deliver is in().
  }.
  ...
  P2B3_TP2 is abstraction(
    vBeverageQuantity : location[Natural],
    vPaid : location[Natural],
    vCoinNumber : location[Natural],
    vPrice : location[Natural]
  ){
    choose {
      pCoin : location(4).
      if ((cPrice' <= pCoin'+vPaid') and (pCoin' > 0)) then{
        via Coin send pCoin. vPaid := vPaid'+pCoin.
        vCoinNumber := vCoinNumber'+1. vPrice := cPrice'.
        P2B4_TP3(vBeverageQuantity',vPaid',vCoinNumber',vPrice')
      }
    or
      via Deliver receive. Fail()
    or
      via Return receive pPaid : location[Natural]. Fail()
    }
  }
  P2B4_TP3 is abstraction(
    vBeverageQuantity : location[Natural],
    vPaid : location[Natural],
    vCoinNumber : location[Natural],
    vPrice : location[Natural]
  ){
    choose {
      via Deliver receive.
      if (vPaid'>=vPrice') then{
        vBeverageQuantity := vBeverageQuantity'+1.
        P4B2_TP4(vBeverageQuantity',vPaid',vCoinNumber',vPrice')
      }else{ Fail() }
    or
      via Return receive pPaid : location[Natural].
      if ((pPaid'=vPaid'-vPrice') and (vPaid'>=vPrice')) then{
        Inconclusive()
      }else{ Fail() }
    or
      via Return receive pPaid : location[Natural].
      if ((pPaid'=vPaid') and (vCoinNumber'>=cCoinNumber'))
            then{
        Inconclusive()
      }else{ Fail() }
  }
  ...
  Pass is abstraction(){ print("PASS") }
  ...
  behaviour is { P1B1_TP1(0,0,0,0) }
}
```

FIG. 10 – *Le cas de test (π-ADL-C&C).*

L'objectif de test est un IOSTS, avec des localités et des transitions. La génération de cas de test est réalisée à travers le calcul du produit entre l'IOSTS de spécification et l'IOSTS d'objectif de test. Ainsi, les localités dans le cas de test sont des paires constituées d'une localité de la spécification et d'une localité de l'objectif de test, et les transitions entre ces localités sont ajoutées lorsque (1) une action d'une transition de la spécification a la même étiquette qu'une action de l'objectif de test, ou (2) la spécification est capable d'avancer sur une action interne. Les localités "Accept" et "Reject" de l'objectif de test indiquent les localités du cas de test qui doivent être interprétées comme finales. La localité "Accept" indique une exécution réussie des tests alors que la localité "Reject" indique le comportement de la spécification de la machine à café qui ne nous intéresse pas pour le moment.

L'objectif de test (Fig. 8) a été construit pour sélectionner un comportement qui (1) commence avec l'action *PressButton ?()*, (2) attend une pièce de monnaie (voir *Coin ?(pCoin)*), puis (3) délivre un café à l'aide de l'action *Deliver !()* et (4) renvoie le reste de la somme qui a été payée (voir *Return !(pRemainingValue)*). A noter que, nous ne nous intéressons pas au test des comportements de la machine à café lors d'une annulation de commande. C'est pourquoi l'action *Cancel* mène à la localité "Reject". Par souci de simplicité, toutes les transitions de la Fig. 8 menant à "Reject" sont étiquetées avec *otherwise*. Cela indique que nous ne sommes intéressés que par les actions autorisées. Par exemple, dans la localité $p_1b_1\_tp_1$, l'action autorisée est *PressButton ?()*, toutes les autres, *c'est-à-dire, Cancel ?()*, *Coin ?(pCoin)*, *Deliver ?()* et *Return ?(pReminingValue)*, vont à "Reject".

Cas de test. L'IOSTS, représentant le cas de test qui a été généré en utilisant la spécification architecturale de la machine à café et l'objectif de test de la Fig. 8, est montré sur la Fig. 9. A noter que, ce cas de test est spécifique à l'objectif de test expliqué dans le paragraphe ci-dessus. Des objectifs de test différents permettent de générer des cas de test différents.

Les étapes de calcul effectuées sont identiques à celles de la spécification. L'orientation des actions (*c'est-à-dire,* entrée/sortie) est inversée car le cas de test devient un générateur de commandes et un récepteur de réponses qui est complémentaire à une implémentation de la spécification architecturale.

La localité étiquetée "Pass" de la Fig. 9 indique qu'une interaction correcte entre le testeur et le système en cours de test a eu lieu. La méthode de génération symbolique de tests génère également des transitions à partir de chaque localité vers une nouvelle localité "Fail" qui absorbe les réponses incorrectes du système en cours de test et conduit à l'état "Fail", indiquant la non-conformité de l'implémentation. Pour chaque action d'entrée erronée reçue par le testeur, le cas de test génère une transition vers "Fail" étiquetée, par souci de clarté de la présentation, avec l'action *otherwise ?* à partir de chaque localité de l'IOSTS.

A noter que, le test montré sur la Fig. 9, comme tous les tests générés par cette méthode, intègre son propre oracle. Toutes les étapes de calcul nécessaires pour vérifier l'exactitude des résultats numériques sont extraites de la spécification et utilisées par le testeur pour vérifier les arguments quand ils sont reçus. Ceci est en contraste avec les techniques de génération de test qui produisent simplement une séquence d'entrées pour conduire l'implémentation à travers un chemin spécifique.

4.3 Du cas de test abstrait au cas de test exécutable

Dans cette section, nous expliquons comment un cas de test abstrait, représenté par un IOSTS, est traduit en un cas de test concret qui peut être exécuté sur l'implémentation boîte noire d'un système en cours de test. Tout d'abord, le cas de test, illustré par la Fig. 9 et appelé TC_{IOSTS}, est traduit en le composant en π-ADL-C&C, montré sur la Fig. 10 et appelé $TC_{\pi\text{-ADL-C\&C}}$, comme suit :

1. Les constantes symboliques de TC_{IOSTS}, telles que *cCoinNumber*, *cBeverageQuantity* et *cPrice*, sont transformées en paramètres du composant de test $TC_{\pi\text{-ADL-C\&C}}$ (voir les lignes 2-4).
2. Les actions d'entrée/sortie de TC_{IOSTS}, telles que *Deliver ?*, *Return ?* et *Coin !*, *Cancel !*, *PressButton !*, jouent le rôle de connecteurs dans $TC_{\pi\text{-ADL-C\&C}}$ (voir les lignes 6-10).
3. Chaque localité de TC_{IOSTS} est transformée en une abstraction de $TC_{\pi\text{-ADL-C\&C}}$. Toutes les abstractions, sauf celles correspondant aux verdicts de test, ont le même nombre de paramètres. Ces paramètres correspondent aux variables de TC_{IOSTS}. Par exemple, la localité $p_2b_3\_tp_2$ de TC_{IOSTS} est traduite en l'abstraction P2B3_TP2 (voir les lignes 13-31), qui possède quatre paramètres qui sont : *vBeverageQuantity*, *vPaid*, *vCoinNumber* et *vPrice*. A noter que, les localités spéciales, telles que *Pass*, *Fail* et *Inconclusive*, correspondent aux abstractions sans paramètres (par exemple, la localité *Pass* correspond à l'abstraction représentée

par le code de la ligne 56). Le rôle de ces abstractions est de produire un verdict de test.

4. Pour chaque localité de TC_{IOSTS}, chaque transition sortante de cette localité est traduite en un cas de la structure "`choose`" de l'abstraction correspondant à cette localité. Par exemple, la transition t_1 qui a pour origine la localité $p_2b_3\_tp_2$ et pour destination la localité $p_2b_4\_tp_3$, et qui est étiquetée avec l'action de sortie *Coin !(pCoin)*, correspond au premier cas de la structure "`choose`" de l'abstraction P2B3_TP2 (voir les lignes 20-25). A noter que, la destination de t_1 est modélisée par un appel à l'abstraction P2B4_TP3.

   Le code, correspondant à une transition gardée et étiquetée avec une action de sortie, est entouré par la structure "`if(...)then{...}`", où la garde de cette transition apparaît comme une condition. Afin d'emprunter une transition étiquetée avec une action de sortie avec des paramètres, un cas de test doit générer automatiquement des valeurs pour ces paramètres satisfaisant la garde de cette transition si elle est présente. Pour l'instant, ces paramètres sont instanciés avec les valeurs choisies par le concepteur du test. Par exemple, le paramètre *pCoin* est instancié avec 4. Cette valeur satisfait la garde de la transition t_1, *c'est-à-dire,* $(pCoin > 0)$ and $(cPrice \leq pCoin + vPaid)$ si le prix de la boisson est 3, par exemple.

   Le code, correspondant à une transition gardée et étiquetée avec une action d'entrée, est entouré par la structure "`if(...)then{...}else{...}`", où la garde de cette transition apparaît comme une condition. L'action d'entrée doit être appelée juste avant cette structure étant donné que nous avons besoin de connaître les valeurs reçues de ses paramètres. A noter que, si la garde/condition n'est pas satisfaite, alors le cas de test génère le verdict "Fail". Par exemple, le code correspondant aux lignes 39-43, modélise deux transitions de TC_{IOSTS} sortant de la localité $p_2b_4\_tp_3$ et étiquetées avec l'action de *Delivery ?()*. L'une d'elles permet d'accéder à la localité $p_4b_2\_tp_4$, si la garde $g : vPaid \geq pPrice$ est satisfaite et l'autre va à la localité "Fail", si la garde g n'est pas satisfaite.

5. Le comportement du composant de test $TC_{\pi\text{-ADL-C\&C}}$ est modélisé par un appel à l'abstraction P1B1_TP1 qui correspond à la localité initiale de TC_{IOSTS}.

Pour obtenir un cas de test exécutable, un cas de test exprimé dans le langage π-ADL-C&C est automatiquement traduit dans un programme de test exécutable concret exprimé dans le langage π-ADL.NET (voir l'article Leroux et al. (2013)).

4.4 Exécution du cas de test

La dernière étape de notre approche consiste à compiler et exécuter le cas de test écrit en π-ADL.NET qui été obtenu à partir d'un cas de test abstrait, représenté par un IOSTS. Ce cas de test est exécuté sur une implémentation boîte noire réelle du système en cours de développement, où l'exécution est modélisée par la composition parallèle entre le cas de test et l'implémentation avec synchronisation sur les actions d'entrée/sortie communes. Les résultats d'une exécution de test sont les suivants : "Pass", qui signifie qu'aucune erreur n'a été détectée et que l'objectif de test a été satisfait, "Inconclusive" – aucune erreur n'a été détectée, mais l'objectif de test n'a pas

été satisfait, ou "Fail" – l'implémentation présente une non-conformité par rapport à la spécification architecturale dans un comportement ciblé par l'objectif de test.

5 Résumé et travaux connexes

L'objectif principal de cet article est de proposer une approche qui permet : (1) de concevoir facilement l'architecture d'un système logiciel en cours de développement en utilisant le langage π-ADL et (2) de valider la conformité de l'implémentation (ou bien d'un prototype exécutable de l'architecture) de ce système par rapport à sa spécification architecturale en utilisant une technique de test.

Comme il est mentionné dans Malavolta et al. (2013); Medvidovic et Taylor (2000), plusieurs travaux proposent différents langages, formels et semi-formels, de description d'architecture (*Architecture Description Language* ou ADL). Certains de ces ADL s'appuient sur les systèmes de transitions étiquetées (*Labeled Transitions Systems* ou LTS) utilisés pour modéliser les comportements d'une architecture logicielle, par exemple, CHAM (*Chemical Abstract Machine*), voir Inverardi et Wolf (1995), FSP (*Finite State Process*), voir Magee et al. (2009), ou π-ADL, voir Oquendo (2004). Nous avons choisi π-ADL comme langage pour la conception de l'architecture, car le travail décrit dans l'article est fondé sur nos travaux antérieurs. Une fois que l'architecture d'un système logiciel est conçue en utilisant le langage π-ADL-C&C de haut niveau, elle est ensuite raffinée dans l'architecture de bas niveau, exprimée en π-ADL.NET, qui peut être compilée, en utilisant le compilateur de Qayyum et Oquendo (2008) développé dans notre équipe de recherche, puis exécutée sur la plate-forme .NET.

Le choix du langage π-ADL nous permet d'utiliser des méthodes formelles dans le but d'assurer la qualité d'un système en cours de développement. En effet, π-ADL est un ADL formel bien fondé sur une base théorique. De plus, le comportement des systèmes conçus peut être exprimé en utilisant les systèmes de transitions. Dans cet article, nous utilisons une technique de test afin de valider la conformité de l'implémentation d'un système par rapport à sa spécification architecturale, et donc d'assurer la qualité de ce système. Ce travail est basé sur notre technique proposée dans la thèse Zinovieva-Leroux (2004). Il utilise également notre outil Ployette et Ponscarme (2007); Clarke et al. (2002) permettant la génération automatique de tests pour les programmes réactifs (écrit en Java ou C++) à partir des spécifications de bas niveau modélisées par des systèmes de transitions symboliques d'entrées-sorties (*Input-Output Symbolic Transitions Systems* ou IOSTS).

Les travaux les plus proches de ceux de l'article sont ceux de Muccini, Bertolino, et Inverardi (Muccini et al. (2004); Bertolino et al. (2003, 2001, 2000)). Leur approche permet la génération automatique de cas de tests abstraits à partir des comportements d'un système en cours de test qui est modélisé par des LTS. Les cas de test sont sélectionnés (1) en utilisant d'abord un LTS abstrait qui permet d'abstraire les actions du système qui ne sont pas intéressantes pour le moment, puis (2) en appliquant le critère de couverture de McCabe aux cas de tests abstraits obtenus (à noter que autres critères peuvent également être utilisés). Une des difficultés de cette approche, qui est soulignée par les auteurs, est d'établir une relation entre l'architecture d'un système, qui est très abstraite, et l'implémentation du système, qui est concrète. Cette relation

est nécessaire afin d'obtenir des cas de test concrets, qui peuvent être exécutés sur le système en cours de test, à partir de cas de tests abstraits.

Dans cet article, nous proposons l'approche permettant de générer des cas de tests abstraits à partir de la spécification architecturale d'un système écrit en π-ADL et modélisé par un IOSTS. Nous utilisons la notion d'objectif de test, comme mécanisme de sélection des cas de tests, afin de pouvoir de tester les comportements spécifiques du système. L'inconvénient est que nous ne générons pas les objectifs de test automatiquement, donc leur élaboration nécessite une intervention humaine. Par contre, la traduction des cas de test abstraits en cas de test concrets est assez directe et simple dans notre approche, voir l'article de Leroux et al. (2013). Pour terminer la section, nous survolons rapidement d'autres travaux connexes qui étaient réalisés dans le domaine du test architectural. A noter que, cette liste n'est certainement pas exhaustive.

Richardson et Wolf (1996) ont défini six critères de couverture architecturale et les ont utilisé pour la génération des plans de test à partir de l'architecture logicielle modélisée par CHAM en adaptant les techniques existantes basées sur les spécifications pour le domaine du test architectural. Stafford et al. (1997, 1998) proposent des techniques d'analyse de dépendance basées sur l'architecture logicielle et appelées chaînage. Bertolino et Inverardi (1996) utilisent le test d'architecture afin de tester les propriétés extra-fonctionnelles d'un système sous test. Tracz (1996) montre comment utiliser l'architecture logicielle spécifique à un domaine (*Domain-Specific Software Architecture* ou DSSA) afin de capturer les propriétés structurelles et temporelles d'un système en cours de développement. Il donne quelques idées sur la façon dont les architectures peuvent être spécifiées pour pouvoir effectuer leur analyse et leur test. Rosenblum (1998) adapte sa stratégie de test basée sur des composants au test de systèmes logiciels basé sur l'architecture. Cette approche est basée sur les modèles architecturaux pouvant être simulés, exécutés ou utilisés pour réaliser le test d'intégration ou le test de régression sur l'implémentation d'un système logiciel. Enfin, l'auteur décrit comment les modèles formels, combinés à des modèles architecturaux, peuvent être utilisés pour guider le test logiciel. Jin et Offutt (2001) définissent plusieurs critères de couverture, et proposent des techniques et des outils qui permettent d'automatiser le processus de spécification et de génération de tests à partir de descriptions architecturales du système. Harrold (1998) présente des approches pour l'utilisation de l'architecture logicielle afin de réaliser efficacement le test de régression. Dans Harrold (2000), elle aborde également l'utilisation de l'architecture logicielle dans le but d'effectuer le test logiciel. Muccini et ses collègues se sont également intéressés aux tests de régression. Leur contribution à ce sujet peut être trouvée dans Muccini et al. (2005c,a,b). Ces travaux explorent la question de savoir comment le test de régression peut être systématiquement appliqué à l'architecture logicielle afin de réduire le coût de régénération des tests pour des systèmes modifiés. Les auteurs s'intéressent à deux types de modifications d'un système logiciel, qui sont : (1) la modification de l'architecture du système et (2) la modification de l'implémentation. De plus, Bertolino (2007) discute de différents travaux importants réalisés dans le domaine du test logiciel et liste les défis les plus importants à traiter dans ce domaine.

6 Conclusion

Cet article a présenté une approche formelle pour tester les systèmes logiciels au niveau architectural. En particulier, cette approche a été appliquée aux systèmes logiciels conçus en utilisant le langage de description architectural de haut niveau appelé π-ADL. L'approche est basée sur la génération symbolique de test, qui (1) génère automatiquement les cas de test afin de vérifier la conformité d'un système par rapport au comportement d'une spécification architecturale choisie par les objectifs de test ; (2) détermine automatiquement si les résultats de l'exécution d'un test sont corrects par rapport à la spécification architecturale. Elle effectue la dérivation de tests comme un processus symbolique, jusqu'à la génération du code source d'un programme de test. La raison pour utiliser des techniques symboliques plutôt qu'énumératives est que la génération symbolique de test nous permet de produire (1) des cas de test plus généraux avec des paramètres et des variables qui doivent être instanciés seulement avant l'exécution des cas de test et (2) des cas de test plus lisibles par les humains. Nous avons validé notre approche sur un exemple simple de machine à café.

Comme il a été mentionné dans l'article, certaines étapes de notre approche sont semi-automatisées, donc, le premier sens de notre futur travail est de rendre l'approche complètement automatique, de la génération de test jusqu'à l'exécution du test. Pour démontrer la faisabilité et l'utilité de notre approche, nous prévoyons de l'appliquer à une étude de cas réaliste. Deuxièmement, nous avons l'intention de travailler sur l'implémentation d'un mécanisme pour calculer automatiquement des objectifs de test à partir de la spécification d'architecture du système, en utilisant, par exemple, les critères de couverture pour remplacer les objectifs de tests écrits à la main. Troisièmement, nous prévoyons d'étendre notre approche en y incorporant une technique de vérification par modèle (model checking) afin de permettre la vérification automatique des parties critiques d'un système en cours de développement.

Références

Beizer, B. (1990). *Software Testing Techniques*. New York : Van Nostrand Reinhold.

Bertolino, A. (2007). Software testing research : Achievements, challenges, dreams. In *Proc. of the Future of Software Engineering*, ICSE'07, pp. 85–103. IEEE-CS Press.

Bertolino, A., F. Corradini, P. Inverardi, et H. Muccini (2000). Deriving test plans from architectural descriptions. In *Proc. of the 22nd Int. Conf. on Software Engineering*, ICSE'00, New York, NY, USA, pp. 220–229. ACM.

Bertolino, A. et P. Inverardi (1996). Architecture-based software testing. In *Proc. of ISAW-2 and Viewpoints'96 on SIGSOFT '96 workshops*, ISAW'96, New York, NY, USA, pp. 62–64. ACM.

Bertolino, A., P. Inverardi, et H. Muccini (2001). An explorative journey from architectural tests definition downto code tets execution. In *Proc. of IEEE Int. Symposium on Software Reliability Engineering*, ICSE'01, pp. 211–220.

Bertolino, A., P. Inverardi, et H. Muccini (2003). Formal methods in testing software architectures. In *SFM*, pp. 122–147.

Clarke, D., T. Jéron, V. Rusu, et E. Zinovieva (2002). STG : A Symbolic Test Generation tool. In *Proc. of the 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of System (TACAS'02)*, Volume 2280 of *LNCS*, Grenoble, France, pp. 470–475.

Harrold, M. J. (1998). Architecture-based regression testing of evolving systems. In *Proc. of the Int. Workshop on the Role of Software Architecture In Testing and Analysis*, ROSATEA'98, pp. 73–77.

Harrold, M. J. (2000). Testing : a roadmap. In *Proc. of the Conf. on The Future of Software Engineering*, ICSE'00, New York, NY, USA, pp. 61–72. ACM.

Inverardi, P. et A. L. Wolf (1995). Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Trans. on Software Eng. 21*(4), 373–386.

Jin, Z. et J. Offutt (2001). Deriving tests from software architectures. In *Proc. of the IEEE Int. Symposium on Software Reliability Engineering*, ICSE'01, pp. 308–313.

Leroux, E., F. Oquendo, et Q. Xiong (2013). Validation de systèmes en utilisant l'architecture logicielle : l'approche fondée sur le test. In *7ème Conférence francophone sur les architectures logicielles*, CAL'13, pp. –.

Magee, J., J. Kramer, R. Chatley, S. Uchitel, et H. Foster (2009). Ltsa - labelled transition system analyser. Available at http ://www.doc.ic.ac.uk/ltsa/.

Malavolta, I., P. Lago, H. Muccini, P. Pelliccione, et A. Tang (2013). What industry needs from architectural languages : A survey. *IEEE Trans. Software Eng. 39*(6), 869–891.

Medvidovic, N. et R. N. Taylor (2000). A classification and comparison framework for software architecture description languages. *IEEE Trans. on Software Eng. 26*(1), 70–93.

Muccini, H., A. Bertolino, et P. Inverardi (2004). Using software architecture for code testing. *IEEE Trans. on Software Engineering 30*(3), 160–171.

Muccini, H., M. Dias, et D.Richardson (2005a). Towards software architecture-based regression testing. In *Workshop on Architecting Dependable Systems (WADS)*, Volume 30 :4 of *ICSE'05*, St. Louis, Missouri (USA), pp. 1–7. ACM.

Muccini, H., M. Dias, et D.Richardson (2005b). Towards software architecture-based regression testing. Technical report, University of L'Aquila.

Muccini, H., M. S. Dias, et D. J. Richardson (2005c). Reasoning about software architecture-based regression testing through a case study. In *Proc. of the Computer Software and Applications Conf.*, COMPSAC'05, pp. 189–195.

Myers, G. (1979). *The Art of Software Testing*. John Wiley & Sons.

Oquendo, F. (2004). π-adl : an architecture description language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures. *SIGSOFT Softw. Eng. Notes 29*(3), 1–14.

Ployette, F. et F.-X. Ponscarme (2007). The STG tool page. Available at http ://www.irisa.fr/prive/ployette/stg-doc/stg-web.html.

Qayyum, Z. et F. Oquendo (2008). The π-adl.net project : an inclusive approach to adl compiler design. *WSEAS Transactions on Computers 7*(5), 414–423.

Richardson, D. J. et A. L. Wolf (1996). Software testing at the architectural level. In *Proc. of ISAW and Viewpoints'96 on SIGSOFT'96 workshops*, ISAW'96, New York, NY, USA, pp. 68–71. ACM.

Rosenblum, D. (1998). Challenges in exploiting architectural models for software testing. In *Proc. of the Int. Workshop on the Role of Software Architecture in Testing and Analysis*, ROSATEA'98, Italy, pp. 49–53.

Sangiorgi, D. (1992). *Expressing Mobility in Process Algebras : First-Order and Higher-Order Paradigms*. Ph. D. thesis, University Edinburgh, UK.

Shaw, S. et D. Garlan (1996). *Software Architecture : Perspectives on an Emerging Discipline*. Prentice Hall.

Stafford, J., D. Richardson, et A. Wolf (1998). Aladdin : A tool for architecture-level dependence analysis of software systems. Technical Report CU-CS-858-98, University of Colorado.

Stafford, J. A., D. J. Richardson, et A. L. Wolf (1997). Chaining : A software architecture dependence analysis technique.

Tracz, W. (1996). Test and analysis of software architectures. In *Proc. of the 1996 ACM SIGSOFT Int. Symposium on Software Testing and Analysis*, ISSTA'96, New York, NY, USA, pp. 1–3. ACM.

Tretmans, G. J. (1992). *A Formal Approach to Conformance Testing*. Ph. D. thesis, University of Twente, the Netherlands.

Zinovieva-Leroux, E. (2004). *Symbolic methods in test generation for reactive systems with data*. Ph. D. thesis, University of Rennes 1, France.

Summary

In the last two decades, software architecture has played a central role in the development of software systems. It provides a high-level description for large-size and complex systems using suitable abstractions of the system's components and their interactions. In our work, the software architecture is described using a formal Architecture Description Language (ADL) designed in the ArchWare European Project, π-ADL-C&C. One of the purposes of this ADL is to allow formal validation of an implemented system with respect to its architectural model. In this paper, we propose a test-based approach for validating a software system with respect to its architecture by using conformance testing. The architectural abstract tests are derived from an Input-Output Symbolic Transition System (IOSTS) representing the architecture structure and behaviors, which are then translated into concrete tests to be executed on the system under test. To illustrate our approach we use the coffee machine example.