Test de conformité basé sur l'architecture logicielle

Elena Leroux, Flavio Oquendo, Qin Xiong

Validation de systèmes en utilisant l'architecture logicielle : l'approche fondée sur le test

Elena Leroux
IRISA
Université de Bretagne-Sud
Vannes, France
elena.leroux@irisa.fr

Flavio Oquendo
IRISA
Université de Bretagne-Sud
Vannes, France
flavio.oquendo@irisa.fr

Qin Xiong
IRISA
Université de Bretagne-Sud
Vannes, France
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 logiciel est décrite en utilisant un langage de description architecturale (ADL) formel conçu dans le projet européen ArchWare et 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 cet article, nous proposons une approche fondée sur le test de conformité pour valider l'implémentation du système par rapport à son architecture. Les cas de test architecturaux abstraits sont dérivés à partir d'un système de transitions symboliques à entrées-sorties (IOSTS) représentant la structure de l'architecture d'un système et de ses comportements. Ils sont ensuite traduits dans des cas de test concrets qui peuvent être exécutés sur le système sous test. Pour illustrer et valider notre approche, nous utilisons l'étude de cas de la machine à café.

Mots-clés

Architecture logicielle, langage de description d'architecture, test de conformité, test d'architecture, validation.

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 [18] 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 [9]. Ces langages de description architecturale (ADL) 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 [13], 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 en utilisent l'architecture logicielle jouent un rôle crucial dans le processus de

CAL '13 Toulouse, France

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* [12] 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 d'un système logiciel par raport à l'architecture logicielle par le biais de tests, notamment : l'élaboration de plans de test en utilisant des critères de test basés sur CHAM (Chemical Abstract Machine) [16], les techniques d'analyse de dépendance appelées chaînage [19], l'utilisation du test architectural afin de tester les propriétés extrafonctionnelles d'un système sous test [3], la proposition de critères, de techniques et d'outils automatisés pour la spécification et la génération de tests au niveau du système à partir de descriptions architecturales [4], l'utilisation de l'architecture logicielle pour le test de régression [11, 8], approche qui couvre l'ensemble du cycle de test architectural à partir de la génération des tests jusqu'à leur exécution sur le système sous test [10, 6, 5, 2].

Dans cet article, nous nous intéressons au *test de conformité basé sur un modèle* [1, 20]. 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 une spécification architecturale d'un système logiciel écrite en π-ADL, et 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 sous test boîte noire. La Section 5 décrit les outils utilisés et/ou développés pour notre approche. 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 π-ADL

Le langage π-ADL [13], 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 [17]. 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 les ports externes et le 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. Ce langage est facile à apprendre et utiliser. 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 Figure 4 montre l'architecture abstraite de la machine à café en termes de composants et de connecteurs. 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'une annulation de commande (par le connecteur *Cancel()*), et ensuite, soit délivre la boisson (par le connecteur *Deliver()*) ou rend la monnaie (par le connecteur de *Return(Natural)*). Elle est représentée par deux composants : *Payment* et *Beverage*.

L'utilisateur de la machine à café, en utilisant l'interface de la machine, envoi 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

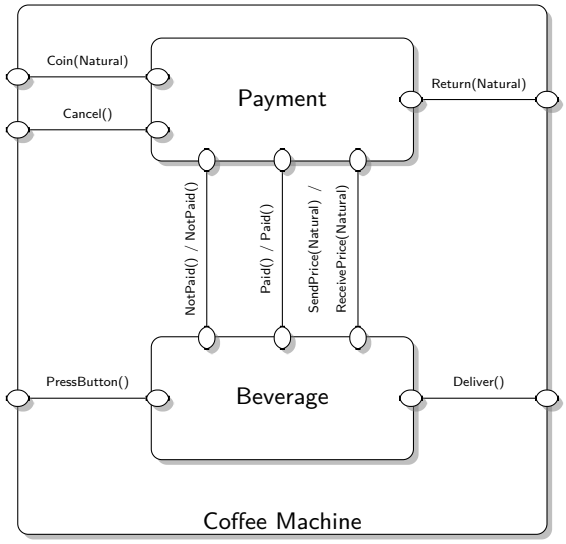

Figure 1: L'architecture de la machine à café.

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/insuffisant au composant *Beverage*, (3) rendre la monaie 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 logicielle 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 de deux composants de la machine à café, *c'est-à-dire,* les composants *Beverage* (voir Figure 2) et *Payment* (voir Figure 3).

Le composant Beverage. Le composant *Beverage*, montré à la Figure 2, est déclaré comme une abstraction (voir ligne 1) avec deux paramètres naturels : (1) *cBeverageQuantity* indiquant la quantité de boisson dans la machine à café, (2) *cPrice* indiquant le prix de la boisson. Les ports externes

```
1  component Beverage is abstraction(cBeverageQuantity : Natural, cPrice : Natural){
2
3    port is {
4      connection PressButton is in().
5      connection Deliver is out().
6      connection SendPrice is out(Natural).
7      connection Paid is in().
8      connection NotPaid is in().
9    }
10
11   drink is abstraction(vBeverageQuantity : location[Natural]){
12     if (vBeverageQuantity >= cBeverageQuantity) then{
13       via PressButton receive.
14       drink(vBeverageQuantity)
15     }else{
16       via PressButton receive.
17       via SendPrice send cPrice.
18       choose{
19         via NotPaid receive.
20         drink(vBeverageQuantity)
21       or
22         via Paid receive.
23         via Deliver send.
24         vBeverageQuantity := vBeverageQuantity'+1.
25         drink(vBeverageQuantity)
26       }
27     }
28   }.
29   behaviour is {
30     drink(location(0))
31   }
32 }
```

Figure 2: Le composant beverage de la machine à café exprimé en π-ADL-C&C.

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).

Le composant Payment. La description formelle du composant *Payment* est donnée sur la Figure 3. Elle est similaire à l'un des composants *Beverage*. Par conséquent, nous ne la détaillons pas dans cet article.

```
1  component Payment is abstraction(cCoinNumber: Natural){
2
3    port is {
4          connection Coin is in (Natural).
5          connection Return is out (Natural).
6          connection Cancel is in ().
7          connection ReceivePrice is in (Natural).
8          connection Paid is out ()
9          connection NotPaid is out ()
10   }.
11
12   paying is abstraction(
13          cCoinNumber: Natural,
14          vPaid: location[Natural],
15          vCoinNumber: location[Natural],
16          vPrice: location[Natural]
17        ){
18     choose {
19       if vCoinNumber < cCoinNumber then {
20         via Coin receive pCoin : Natural.
21         vPaid := vPaid'+pCoin.
22         vCoinNumber := vCoinNumber'+1.
23         paying(cCoinNumber, vPaid, vCoinNumber, vPrice)
24       } else {
25         via Return send vPaid.
26         paying(cCoinNumber, location(0), location(0), location(0))
27       }
28     or
29       via ReceivePrice receive pPrice : Natural.
30       vPrice := pPrice.
31       paying(cCoinNumber, vPaid, vCoinNumber, vPrice)
32     or
33       via Cancel receive.
34       via Return send vPaid.
35       paying(cCoinNumber, location(0), location(0), location(0))
36     or
37       if vPaid >= vPrice then {
38         via Paid send.
39         via Return send (vPaid-vPrice).
40         paying(cCoinNumber, location(0), location(0), location(0))
41       } else {
42         via NotPaid send.
43         paying(cCoinNumber, vPaid, vCoinNumber, vPrice)
44       }
45     }
46   }.
47   behaviour is {
48     paying(cCoinNumber, location(0), location(0), location(0))
49   }
50 }
```

Figure 3: Le composant payment de la machine à café exprimé en π-ADL-C&C.

L'architecture de la machine à café. L'architecture de la machine à café est formellement décrite sur la Figure 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.

```
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 }
```

Figure 4: L'architecture d'une machine à café exprimée en π-ADL-C&C.

3. MODÈLE SOUS-JACENT POUR LA GÉNÉRATION DE CAS DE TEST

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 [14, 7], nous traduisons automatiquement une spécification architecturale dans le modèle de bas niveau appelé système de transitions symboliques à entrées-sorties (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 [21]. L'explication intuitive est donnée ci-dessous en utilisant l'exemple montré sur la Figure 5, qui représente le composant *Payement* de la machine à café. A noter que, le composant *Beverage* peut aussi être modélisé par un IOSTS comme il est montré sur la Figure 6.

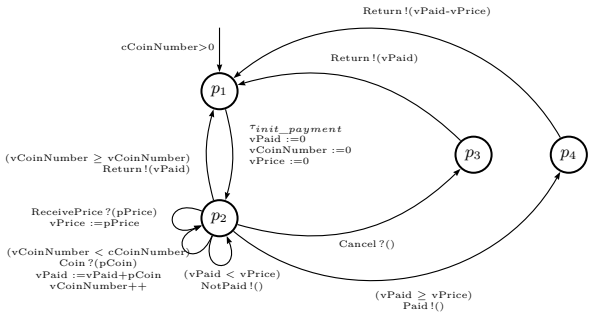

Figure 5: Le composant *Payment* de la machine à café, modélisé par un IOSTS.

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

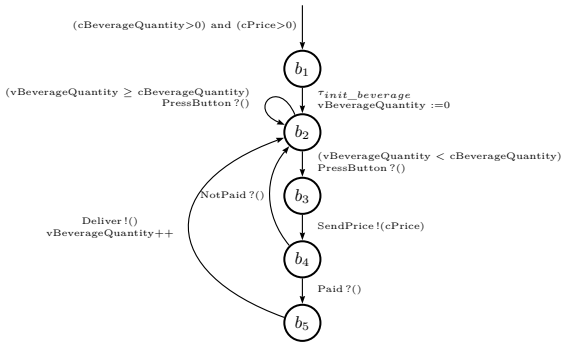

Figure 6: Le composant *Beverage* de la machine à café, modélisé par un IOSTS.

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 Figure 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.

3.1 Sémantique des IOSTS

Sémantique informelle. Soit l'IOSTS (voir Figure 5) représentant le composant *Payement* de la machine à café. 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.

Sémantique formelle. Un *état* s est une paire $\langle l, \vartheta \rangle$, où l est une localité et ϑ est une valuation des constantes et des variables, par exemple, $s = \langle Coin,\ cCoinNumber{=}10,\ vPrice{=}3,\ vPaid{=}2,\ vCoinNumber{=}4 \rangle$. Un *état initial* $s^0 = \langle l^0, \vartheta^0 \rangle$ est un état où l^0 est la localité initiale, et ϑ^0 est une valuation des constantes et des variables qui satisfait la condition initiale. On note S (*resp.* S^0) l'ensemble de tous les états (*resp.* les états initiaux). Une *action valuée* α est une paire $\langle a, \omega \rangle$, où a est une action et ω est une valuation des paramètres de a. Par exemple, $\alpha = \langle Coin, pCoin = 1 \rangle$ ou $\alpha = \langle \tau_{init\_payment} \rangle$. On note par $\Lambda = \Lambda^? \cup \Lambda^! \cup \Lambda^\tau$ l'ensemble des actions valuées qui est divisé en trois sous-ensembles d'actions valuées d'entrée, d'actions valuées de sortie et d'actions internes. Ensuite, nous définissons la *relation de transition* $\rightarrow$ comme l'ensemble des triplets $\langle s, \alpha, s' \rangle$, où $s = \langle l, \vartheta \rangle$, $s' = \langle l', \vartheta' \rangle$ sont des états et $\alpha = \langle a, \omega \rangle$ est une action valuée telle que ϑ, ω sont des valuations des constantes, des variables et des paramètres qui satisfont la garde d'une transition t avec pour origine l et pour destination l' et qui est étiquetée avec l'action a, et ϑ' est la nouvelle valuation des variables et des constantes obtenues à partir de ϑ par les affectations des variables de t.

Definition 1. Un *comportement* β est une séquence d'états et d'actions valuées à partir d'un état initial et suivant la relation de transition, *c'est-à-dire,*

$$\beta : \quad s^0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \dots s_{n-1} \xrightarrow{\alpha_n} s_n$$

où $\rightarrow$ est la relation de transition, $s^0 \in S^0$, et pour tout $i \in [1, n] : s_i \in S$, $\alpha_i \in \Lambda$.

Pour décrire les comportements observables des IOSTS nous définissons la relation $\Rightarrow$ comme suit :

- $s \overset{\varepsilon}{\Rightarrow} s' \quad \triangleq \quad (s = s') \vee (\exists s_0, \dots, s_n \in S.\ s = s_0 \xrightarrow{\tau_1} s_1 \dots s_{n-1} \xrightarrow{\tau_n} s_n = s')$, où pour tout $i \in [1, n] : \tau_i \in \Lambda^\tau$;
- $s \overset{\alpha}{\Rightarrow} s' \quad \triangleq \quad \exists s_1, s_2 \in S.\ s \overset{\varepsilon}{\Rightarrow} s_1 \xrightarrow{\alpha} s_2 \overset{\varepsilon}{\Rightarrow} s'$, où $\alpha \in \Lambda^? \cup \Lambda^!$.

Definition 2. Un *comportement observable* β est une séquence d'états et d'actions valuées d'entrée ou de sortie, *c'est-à-dire*, $\beta : \quad s^0 \overset{\alpha_1}{\Rightarrow} s_1 \overset{\alpha_2}{\Rightarrow} s_2 \dots s_{n-1} \overset{\alpha_n}{\Rightarrow} s_n$ où $s^0 \in S^0$, et pour tout $i \in [1, n] : \ s_i \in S,\ \alpha_i \in \Lambda^? \cup \Lambda^!$.

Definition 3. Une *trace* σ est la sous-séquence d'un comportement observable $\beta : s^0 \overset{\alpha_1}{\Rightarrow} s_1 \overset{\alpha_2}{\Rightarrow} s_2 \dots s_{n-1} \overset{\alpha_n}{\Rightarrow} s_n$ qui est composé d'actions valuées d'entrée ou de sortie, *c'est-à-dire,* $\sigma : \quad \alpha_1 \alpha_2 \dots \alpha_n$ où pour tout $i \in [1, n] : \alpha_i \in \Lambda^? \cup \Lambda^!$.

3.2 Relation de conformité

La relation de conformité définit l'ensemble des implémentations d'un système qui sont correctes par rapport à sa spécification architecturale. Intuitivement, une implémentation est conforme à une spécification si, pour chaque trace de la spécification, l'implémentation ne produit que des sorties qui sont autorisées par la spécification. Pour définir formellement la relation de conformité, nous définissons d'abord l'ensemble des états dans lesquels un IOSTS M peut se trouver après une trace observable σ : $(M\ after\ \sigma) \triangleq \{s \in S \mid \exists s^0 \in S^0.\ s^0 \stackrel{\sigma}{\Rightarrow} s\}$, et l'ensemble des actions de sortie (*resp.* d'entrée) valuées qui peuvent être générées par M quand il est dans un état s parmi l'ensemble des états $\tilde{S}$: $Out(\tilde{S}) \triangleq \{\alpha \in \Lambda^! \mid \exists s \in \tilde{S}.\ s \stackrel{\alpha}{\rightarrow}\}$ (*resp.* $In(\tilde{S}) \triangleq \{\alpha \in \Lambda^? \mid \exists s \in \tilde{S}.\ s \stackrel{\alpha}{\rightarrow}\}$), où $s \stackrel{\alpha}{\rightarrow} \triangleq \exists s' \in S.\ s \stackrel{\alpha}{\rightarrow} s'$. Enfin, notons $Traces(M)$ l'ensemble des traces de M. A noter que si une trace σ n'appartient pas à $Traces(M)$, alors $Out(M\ after\ \sigma)$ et $In(M\ after\ \sigma)$ sont les ensembles vides.

Definition 4. La *relation de conformité* entre deux IOSTS IUT et $Spec$ avec des constantes identiques fixées est défini comme suit :
$(IUT\ \mathbf{conf}\ Spec) \triangleq$
$\forall \sigma \in Traces(Spec).Out(IUT\ after\ \sigma) \subseteq Out(Spec\ after\ \sigma)$

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 Figure 7 et présentée ci-dessous.

4.1 De π-ADL-C&C à π-ADL-Spec

L'objectif de la première étape de notre approche est de transformer une spécification architecturale écrite en π-ADL-C&C en sa forme canonique en π-ADL-Spec. Pour illustrer cette transformation, nous utilisons le composant *Payement* de la machine à café dont le code π-ADL-C&C est montré sur la Figure 3. Le résultat de la transformation est représenté sur la Figure 8.

1. Tous les composants et leurs comportements internes déclarés comme des abstractions sont traduits comme les abstractions individuelles des comportements. Ces abstractions individuelles peuvent être instanciées plus tard comme des comportements par une application. Par ailleurs, afin de permettre un appel récursif d'une instance d'abstraction, cette abstraction doit être déclarée comme une abstraction récursive dans le langage π-ADL-Spec en utilisant le mot-clé "`recursive`".

   Par exemple, le composant *Payement* (voir les lignes 1-46 de la Figure 3) correspond à son abstraction individuelle figurant sur les lignes 44-46 de la Figure 8 et son comportement interne "paying" (voir les lignes 12-45 de la Figure 3) correspond à l'abstraction récursive montrée sur les lignes 1-42 de la Figure 8.

   Notons que, les paramètres des composants et des comportements internes sont les mêmes que les paramètres des abstractions individuelles correspondantes. Voir, par exemple, la ligne 1 de la Figure 3 et la ligne 44 correspondante de la Figure 8.

2. Toutes les connexions, déclarées dans un composant π-ADL-C&C (voir, par exemple, les lignes 4-9 de la Figure 3), doivent être déclarées dans une abstraction de π-ADL-Spec dans laquelle elles sont utilisées (voir les lignes 7-12 de la Figure 8). A noter que la syntaxe de la déclaration d'une connexion a été modifiée. Par ailleurs, dans le langage π-ADL-Spec, nous n'avons pas besoin de préciser si la connexion est utilisée pour recevoir ou pour envoyer des informations à partir de/à son environnement.

3. Lors de la transformation d'une spécification π-ADL-C&C en une spécification π-ADL-Spec, nous avons aussi besoin de réaliser quelques modifications syntaxiques mineures telles que : (1) l'apostrophe "`'`" utilisée pour la récupération de la valeur stockée dans une localité doit être déplacée juste devant le nom de la localité (voir, par exemple, la ligne 15 de la Figure 8), (2) le point "`.`" à la fin de chaque instruction (voir, par exemple, la ligne 7 de la Figure 8) est transformé en un point-virgule "`;`".

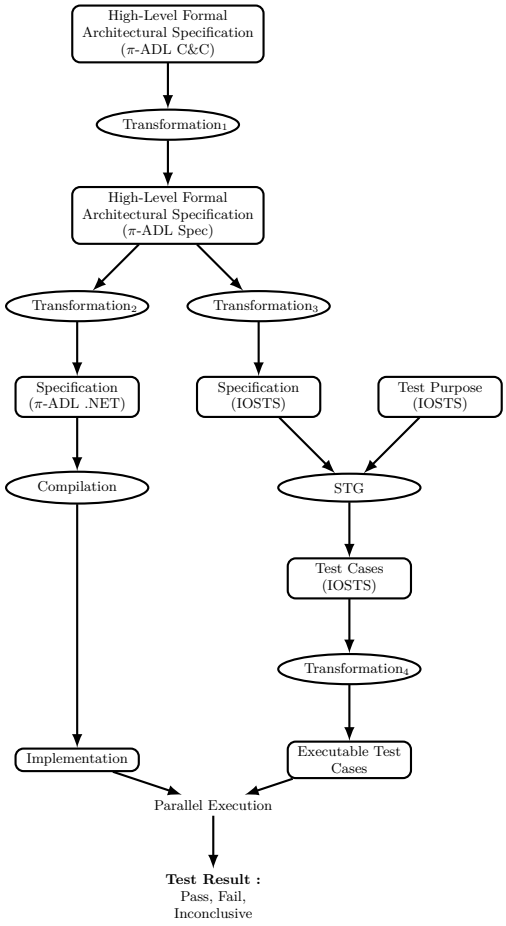

Figure 7: Notre approche pour la validation architecturale.

4.2 De π-ADL-Spec à π-ADL.NET

Afin d'obtenir une implémentation d'un système prête à être compilée et exécutée, nous devons transformer la spécification π-ADL-Spec dans le code π-ADL.NET. Cette section décrit brièvement quelques points importants de cette

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

Figure 8: Le composant payment de la machine à café exprimé en π-ADL-Spec.

transformation et l'illustre en utilisant les Figures 8 et 9.

1. Pour chaque abstraction de π-ADL-Spec, sa liste des paramètres, contenant plus d'un paramètre (voir, par exemple, les lignes 1-5 de la Figure 8), est encapsulé comme une valeur du type "`view`" dans le code π-ADL.NET (voir respectivement les lignes 1-5 de la Figure 9). Chaque valeur du type `view[label`$_1$`:T`$_1$`,...,label`$_n$`:T`$_n$`]` est une vue `view(label`$_1$`=v`$_1$`,...,label`$_n$`=v`$_n$`)`, où pour $i \in [1, n]$, chaque valeur v_i est de type T_i, et chaque étiquette $label_i$ a le même nom que son paramètre correspondant dans le code π-ADL-Spec. La raison est que le langage π-ADL.NET ne peut pas passer une liste de paramètres par valeur.
2. Chaque appel à une abstraction π-ADL-Spec portant des paramètres, qui permettent d'établir les communications entre les comportements et les abstractions (voir, par exemple, la ligne 19 de la Figure 8), est transformé, en code π-ADL.NET, dans une action de sortie envoyant ces paramètres via la connexion portant le même nom que l'abstraction π-ADL-Spec correspondante (voir, les lignes 20-23 de la Figure 9).
3. Chaque type "`location`" du langage π-ADL-Spec (voir, par exemple, la ligne 3 de la Figure 8) est traduit dans le type de la valeur stockée dans cette localité (voir la ligne 3 de la Figure 9).
4. Il y a aussi quelques différences syntaxiques mineures entre les langages π-ADL-Spec et π-ADL.NET telles que, par exemple, les déclarations des connexions (voir la ligne 7 de la Figure 8 et la ligne 7 correspondante

```
1  value paying is abstraction(args:view[
2    cCoinNumber: Integer,
3    vPaid: Integer,
4    vCoinNumber: Integer,
5    vPrice: Integer]
6  ){
7    Coin : connection[Integer];
8    Return : connection[Integer];
9    Cancel : connection[Void];
10   ReceivePrice : connection[Integer];
11   Paid : connection[Void];
12   NotPaid : connection[Void];
13   pCoin : Integer;
14
15   choose {
16     if (args::vCoinNumber < args::cCoinNumber) do {
17       via Coin receive pCoin;
18       args::vPaid = args::vPaid+pCoin;
19       args::vCoinNumber = args::vCoinNumber+1;
20       via paying send view(cCoinNumber: args::cCoinNumber,
21                            vPaid: args::vPaid,
22                            vCoinNumber: args::vCoinNumber,
23                            vPrice: args::vPrice);
24     } else do{
25       via Return send vPaid;
26       via paying send view(cCoinNumber: args::cCoinNumber,
27                            vPaid: 0,
28                            vCoinNumber: 0,
29                            vPrice: 0);
30     }
31   or
32     via ReceivePrice receive pPrice : Natural;
33     vPrice = pPrice;
34     via paying send view(cCoinNumber: args::cCoinNumber,
35                          vPaid: args::vPaid,
36                          vCoinNumber: args::vCoinNumber,
37                          vPrice: args::vPrice);
38   or
39     via Cancel receive;
40     via Return send vPaid;
41     via paying send view(cCoinNumber: args::cCoinNumber,
42                          vPaid: 0,
43                          vCoinNumber: 0,
44                          vPrice: 0);
45   or
46     if (vPaid >= vPrice) do {
47       via Paid send;
48       via Return send (vPaid-vPrice);
49       via paying send view(cCoinNumber: args::cCoinNumber,
50                            vPaid: 0,
51                            vCoinNumber: 0,
52                            vPrice: 0);
53     } else do {
54       via NotPaid send;
55       via paying send view(cCoinNumber: args::cCoinNumber,
56                            vPaid: args::vPaid,
57                            vCoinNumber: args::vCoinNumber,
58                            vPrice: args::vPrice);
59     }
60   }
61 };
62
63 value Payment is abstraction(cCoinNumber: Integer){
64   via paying send view(cCoinNumber: args::cCoinNumber,
65                        vPaid: 0,
66                        vCoinNumber: 0,
67                        vPrice: 0);
68 }
```

Figure 9: Le composant payment de la machine à café exprimé en π-ADL.NET.

de la Figure 9).

4.3 De la spécification architecturale à l'implémentation

L'objectif de cette étape de notre approche est d'obtenir un système logiciel exécutable. Pour atteindre cet objectif, nous utilisons le compilateur π-ADL [15] développé en C# par Z.Qayyum et exécutable sur la plateforme .NET. Ce compilateur prend en entrée un code π-ADL.NET et le transforme en un système exécutable. Nous exécutons alors ce système sur une machine virtuelle persistante développée pour l'exécution de descriptions architecturales fondées sur la sémantique opérationnelle de π-ADL (voir l'Annexe A).

4.4 De π-ADL-Spec à IOSTS

Dans cette section, nous décrivons de façon informelle la transformation d'une spécification architecturale expri-

mée en π-ADL-Spec dans son modèle IOSTS. Nous utilisons l'exemple du composant *Payement*, montré sur la Figure 8 et appelé $S_{\pi\text{-ADL-Spec}}$, afin d'illustrer cette transformation. Le résultat de la transformation est l'IOSTS représenté sur la Figure 5 et appelé S_{IOSTS}.

1. Chaque abstraction π-ADL-Spec correspond à un modèle IOSTS. Par exemple, l'abstraction montrée sur les lignes 44-46 de $S_{\pi\text{-ADL-Spec}}$ correspond à S_{IOSTS} qui modélise les comportements du composant *Payement* de la machine à café.
2. Les connexions d'une abstraction π-ADL-Spec deviennent les actions d'entrée/sortie de l'IOSTS correspondant. Par exemple, les connexions de $S_{\pi\text{-ADL-Spec}}$, *c'est-à-dire, Coin*, *Cancel* et *Return*, *Paid*, *NotPaid* (voir les lignes 7-12), sont les actions d'entrée/sortie de S_{IOSTS}.
3. Chaque préfixe d'entrée et de sortie, dont la syntaxe respective est "`via connection receive value`" et "`via connection send value`", d'une abstraction π-ADL-Spec 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-Spec des lignes 15-23 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` mentionnée dans cette structure.
4. Une séquence de préfixes d'entrée, de sortie et silencieux dans le langage π-ADL-Spec est modélisée par la séquence des transitions correspondantes dans le modèle IOSTS. Par exemple, la séquence "`via Cancel receive.via Return send 'vPaid`" de $S_{\pi\text{-ADL-Spec}}$ (voir les lignes 29-30) est représentée par deux transitions consécutives $(p_2, Cancel?(), p_3)$. $(p_3, Return!(pPaid), p_1)$ de S_{IOSTS} (voir Figure 5).
5. La structure "`choice`" de π-ADL-Spec permet de modéliser une localité d'un IOSTS avec plusieurs transitions sortantes. Par exemple, le code des lignes 14-41 de $S_{\pi\text{-ADL-Spec}}$ correspond à la localité p_2 de S_{IOSTS} et à six transitions sortantes de p_2.
6. Un appel à une abstraction dans le langage π-ADL-Spec, signifie que la transition correspondant 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 19 du $S_{\pi\text{-ADL-Spec}}$ 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 22 signifie que la transition étiquetée par l'action de sortie *Return !* devrait aller à la localité 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 Figure 5 et Figure 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.

4.5 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 un 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 [14, 7], 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 4.4, nous avons décrit comment obtenir la spécification sous la forme d'un IOSTS à partir de son écriture dans le langage π-ADL-Spec. Ci-dessous, nous expliquons les notions d'objectif de test et cas de test.

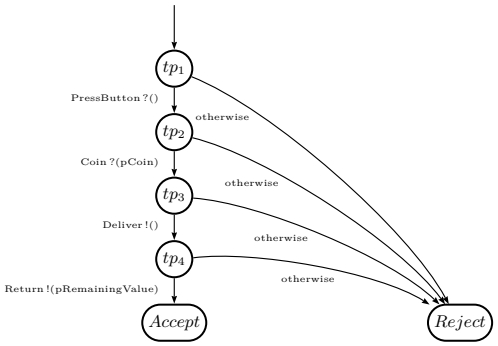

Figure 10: L'objectif de test représenté par un IOSTS.

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 Figure 10. 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.

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 de la Figure 10 a été construit pour choisir 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 Figure 10 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 ?(pRemainingValue)*, vont à la localité "Reject".

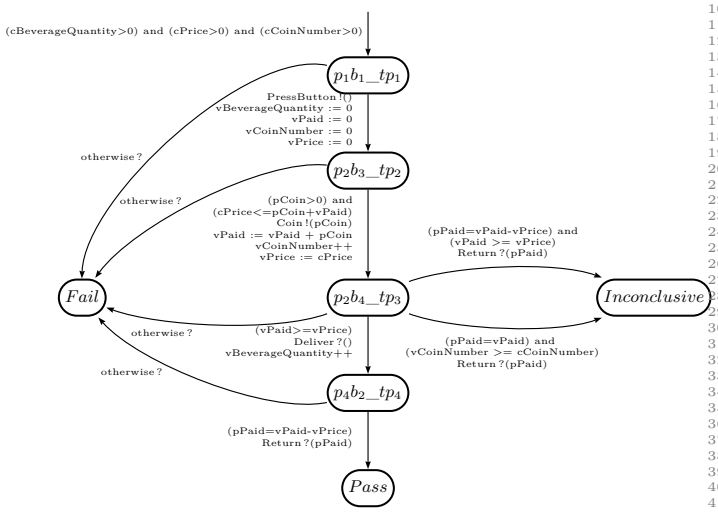

Figure 11: Le cas de test représenté par un IOSTS.

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 Figure 10, est montrée sur la Figure 11. A noter que, ce cas de test est spécifique à l'objectif de test expliqué dans la 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 Figure 11 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 Figure 11, 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.

4.6 Du cas de test abstrait au cas de test exécutable

```
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) }
}
```

Figure 12: L'extrait du cas de test π-ADL C&C.

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 Figure 11 et appelé TC_{IOSTS}, est traduit en le composant en π-ADL-C&C, montré sur la Figure 12 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} (*Deliver ?*, *Return ?*, and *Coin !*, *Cancel !*, *PressButton !*) jouent le rôle de connecteurs dans $TC_{\pi\text{-ADL-C\&C}}$ (voir les lignes 7-11).
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 14-36), qui possède quatre paramètres : *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 61). 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 21-28). 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 44-48, 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 en code π-ADL-Spec (voir Section 4.1), puis dans un programme de test exécutable concret exprimé dans la langage π-ADL.NET (voir Section 4.2).

4.7 Exécution du cas de test

La dernière étape de notre approche consiste à compiler et exécuter le cas de test π-ADL.NET obtenu à partir d'un cas de test abstrait, représenté par un IOSTS, comme cela a été expliqué dans la Section 4.6. 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 d'architecturale dans un comportement ciblé par l'objectif de test.

5. OUTILLAGE

Un avantage majeur offert par les langages formels utilisés pour la description architecturale de systèmes logiciels est que leur formalisme les rend aptes à être manipulés par des outils logiciels. L'utilité d'un ADL est ainsi lié directement aux outils qu'il fournit pour soutenir la description architecturale, l'analyse, le raffinement, la génération de code et l'évolution d'un système logiciel. En effet, nous avons développé un ensemble complet d'outils pour soutenir l'architecture axée sur le développement formel autour de π-ADL. Il est composé de :

- un compilateur et une machine virtuelle pour l'exécution des descriptions d'architecture fondées sur la sémantique opérationnelle de π-ADL (ils sont implémentés en C# sur la plateforme NET.) [15] ;
- trois transformateurs implémentés en C++ et permettant de traduire (1) un code π-ADL-C&C dans un code π-ADLSpec, (2) un code π-ADL-Spec dans un code π-ADL.NET, et (3) un code π-ADL-Spec dans un modèle IOSTS.
- Un vérificateur de syntaxe π-ADL-C&C implémenté en C++.

Le travail présenté dans cet article ajoute une nouvelle méthode et un outil pour la validation de l'implementation d'un système en utilisant sa spécification architecturale, qui est basé sur le test de conformité. En effet, afin de valider la conformité du système exécutable par rapport à sa spécification architecturale, nous appliquons la technique des tests de conformité, à savoir que les tests sont générés automatiquement, en utilisant l'outil STG [21, 14, 7] et ils sont ensuite exécutés sur le système sous test. Pour être en mesure de générer des tests à partir d'une spécification architecturale π-ADL-Spec avec STG, la spécification doit être traduite en

un modèle IOSTS de bas niveau. Cette étape est presque s'automatisée. L'outil STG génère des cas de test abstraits exprimés par des IOSTS, donc nous devons aussi les transformer dans le langage π-ADL-C&C (pour l'instant, cette étape est réalisée manuellement).

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 de 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 test comme un processus symbolique, jusqu'à et incluant 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.

7. REFERENCES

[1] B. Beizer. *Software Testing Techniques*. New York : Van Nostrand Reinhold, 1990.

[2] A. Bertolino, F. Corradini, P. Inverardi, and H. Muccini. Deriving test plans from architectural descriptions. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 220–229, New York, NY, USA, 2000. ACM.

[3] A. Bertolino and P. Inverardi. Architecture-based software testing. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, ISAW '96, pages 62–64, New York, NY, USA, 1996. ACM.

[4] A. Bertolino, P. Inverardi, and H. Muccini. Deriving tests from software architectures. In *Twelfth IEEE International Symposium on Software Reliability Engineering*, ICSE'01, pages 308–313, 2001.

[5] A. Bertolino, P. Inverardi, and H. Muccini. An explorative journey from architectural tests definition downto code tets execution. In *Twelfth IEEE International Symposium on Software Reliability Engineering*, ICSE'01, pages 211–220, 2001.

[6] A. Bertolino, P. Inverardi, and H. Muccini. Formal methods in testing software architectures. In *In SFM*, pages 122–147. Springer, 2003.

[7] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG : A Symbolic Test Generation tool. In *The 8th International Conference on Tools and Algorithms for the Construction and Analysis of System (TACAS'02)*, volume 2280 of *LNCS*, pages 470–475, Grenoble, France, April 2002. Springer-Verlag.

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

[9] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1) :70–93, 2000.

[10] H. Muccini, A. Bertolino, and P. Inverardi. Using software architecture for code testing. *IEEE Transactions on Software Engineering (TSE)*, pages 160–171, March 2004.

[11] H. Muccini, M. S. Dias, and D. J. Richardson. Reasoning about software architecture-based regression testing through a case study. In *Computer Software and Applications Conference*, COMPSAC'05, pages 189–195, 2005.

[12] G. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[13] F. Oquendo. Pi-adl : an architecture description language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures. *SIGSOFT Softw. Eng. Notes*, pages 1–14, 2004.

[14] F. Ployette and F.-X. Ponscarme. The STG tool page. Available at http ://www.irisa.fr/prive/ployette/stg-doc/stg-web.html, October 18, 2007.

[15] Z. Qayyum and F. Oquendo. The pi-adl.net project : an inclusive approach to adl compiler design. *WSEAS Transactions on Computers*, 7(5) :414–423, May 2008.

[16] D. J. Richardson and A. L. Wolf. Software testing at the architectural level. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, ISAW '96, pages 68–71, New York, NY, USA, 1996. ACM.

[17] D. Sangiorgi. *Expressing Mobility in Process Algebras : First-Order and Higher-Order Paradigms*. PhD thesis, University Edinburgh, UK, February 1992.

[18] S. Shaw and D. Garlan. *Software Architecture : Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[19] J. A. Stafford, D. J. Richardson, and A. L. Wolf. Chaining : A software architecture dependence analysis technique, 1997.

[20] G. J. Tretmans. *A Formal Approach to Conformance*

Testing. PhD thesis, University of Twente, the Netherlands, December 1992.

[21] E. Zinovieva-Leroux. *Symbolic methods in test generation for reactive systems with data*. PhD thesis, University of Rennes 1, France, November 22, 2004.

ANNEXE

A. SEMANTIQUE OPERATIONNELLE DE π-ADL

Un système de transition est défini par des règles de transition. Les règles de transition utilisent aussi la structure des règles de preuve. Dans une règle de transition (voir Figure 13), les prémisses et les conclusions sont des relations de transition. Ainsi, si les relations de transition étiquetées par $\alpha_1, \ldots, \alpha_n$ peuvent être empruntées, alors la relation de transition étiquetée par α peut être empruntée, *c'est-à-dire*, si P_1 peut exécuter α_1 et devenir P_1', ... , et P_n peut exécuter α_n et devenir P_n', alors, C peut exécuter α et devenir C'. Les conditions de bord et les effets de bord peuvent également être exprimés. Les conditions de bord peuvent être considérées comme des pré-conditions sur les termes, exprimées dans les Prémisses, et les effets de bord – comme des post-conditions sur les termes exprimés dans la conclusion.

$$\text{Transition rule} = \frac{P_1 \xrightarrow{\alpha_1} P_1' \ldots P_n \xrightarrow{\alpha_n} P_n' \text{ side conditions}}{C \xrightarrow{\alpha} C', \textit{where} \text{ side effects}}$$

Figure 13: Règle de transition

Cette structure sémantique opérationnelle représente le comportement de π-ADL au moyen d'un système déductif, exprimé par le système de transition. Par conséquent, la sémantique formelle de π-ADL est définie par un système de transition, avec des règles de transition formalisant la sémantique opérationnelle des constructions du langage, en ligne avec son système de type, formalisés par les règles de typage. Le "type soundness" affirme que les termes bien typés ne donnent pas lieu à des erreurs d'exécution dans le cadre du système de transition.

Selon cette approche, la sémantique π-ADL est complètement formalisée.

Chaque construction de π-ADL a sa sémantique opérationnelle structurelle spécifiées par un ensemble de règles de transition comme le montrent les Figures 15 et 14. Chaque règle spécifie un comportement possible associé à une construction qui correspond directement à l'explication donnée dans la syntaxe abstraite.

Dans la Figure 15, les quatre premières règles de transition, *c'est-à-dire,* **T-PrefixOutput**, **T-PrefixInput**, **T-PrefixTau** et **T-PrefixMatching**, définissent la sémantique de la construction π-ADL exprimant respectivement le préfixe de sortie : `via connection send value`, le préfixe d'entrée : `via connection receive value`, le préfixe silencieux : `unobservable` et le préfixe match : `if boolean do prefix` (avec la condition de bord que l'expression booléenne est vraie et que α est un préfixe du comportement correspondant). Les règles de transition symétriques **T-Summation-l** et **T-Summation-r** définissent la sémantique de la construction de choix : `choose {` $behavior_0$ `or` ... `or` $behavior_n$ `}` (avec la condition de bord que α est un préfixe du comportement correspondant) et les règles symétriques **T-Composition-l** et **T-Composition-r** définissent la sémantique de la construction de la composition : `compose {` $behavior_0$ **and** ... **and** $behavior_n$ `}` (avec la condition de bord que α n'est pas un nom libre).

Dans la Figure 14, les deux premières règles de transition symétriques, *c'est-à-dire,* **T-Communication-l** et **T-Communication-r**, définissent la sémantique de communication entre les comportements, où l'un est prêt à envoyer et l'autre à recevoir via la même connexion. Les deux prochaines règles de transition, *c'est-à-dire,* **T-Restriction** et **T-Open**, définissent la sémantique des noms concernant la portée : déclarer un nom comme étant interne (privé) à un comportement n'a aucune incidence sur les autres actions portées par le comportement. Les deux dernières règles symétriques, *c'est-à-dire,* **T-Replication-action** et **T-Rep-com**, définissent la sémantique de réplication d'un comportement dans le cas d'une action interne d'un comportement répliqué ou d'une action de communication entre deux occurrences d'un comportement répliqué, respectivement. Dans le premier cas, le nom restreint est passé à un autre comportement qui a déjà ce nom dans sa portée et dans l'autre cas, son champ d'application est étendu à un nouveau comportement.

Transition rules

$$\textbf{T-Communication-l}: \frac{Behavior_1 \xrightarrow{\textbf{restrict}\ \text{names} : \text{T}.\textbf{via}\ \text{name}\ \textbf{send}\ \text{value}} Behavior_1' \quad Behavior_2 \xrightarrow{\textbf{via}\ \text{name}\ \textbf{receive}\ \text{value}} Behavior_2'}{\textbf{compose}\ \{Behavior_1\ \textbf{and}\ Behavior_2\} \xrightarrow{\tau} \textbf{restrict}\ \text{names} : \text{T}.\textbf{compose}\{Behavior_1'\ \textbf{and}\ Behavior_2'\}}\ \text{names} \cap \text{fn}(Behavior_2) = \emptyset$$

$$\textbf{T-Communication-r}: \frac{Behavior_1 \xrightarrow{\textbf{via}\ \text{name}\ \textbf{receive}\ \text{value}} Behavior_1' \quad Behavior_2 \xrightarrow{\textbf{restrict}\ \text{names} : \text{T}.\textbf{via}\ \text{name}\ \textbf{send}\ \text{value}} Behavior_2'}{\textbf{compose}\ \{Behavior_1\ \textbf{and}\ Behavior_2\} \xrightarrow{\tau} \textbf{restrict}\ \text{names} : \text{T}.\textbf{compose}\{Behavior_1'\ \textbf{and}\ Behavior_2'\}}\ \text{names} \cap \text{fn}(Behavior_1) = \emptyset$$

$$\textbf{T-Restriction}: \frac{Behavior \xrightarrow{\alpha} Behavior'}{\textbf{restrict}\ \text{name} : \text{T}.Behavior \xrightarrow{\alpha} \textbf{restrict}\ \text{name} : \text{T}.Behavior'}\ \text{name} \notin \text{n}(\alpha)$$

$$\textbf{T-Open}: \frac{Behavior \xrightarrow{\textbf{restrict}\ n_1:T_1,\ldots,n_n:T_n.\textbf{via}\ \text{name}\ \textbf{send}\ \text{value}} Behavior'}{\textbf{restrict}\ \text{n} : \text{T}.Behavior \xrightarrow{\textbf{restrict}\, n_1:T_1,\ldots,n_n:T_n,n:T.\textbf{via}\ \text{name}\ \textbf{send}\ \text{value}} Behavior'}\ \text{n} \in \text{fn(value)},\ \text{n} \notin \{n_1, \ldots, n_n, \text{name}\}$$

$$\textbf{T-Replication-action}: \frac{Behavior \xrightarrow{\alpha} Behavior'}{\textbf{replicate}\ Behavior \xrightarrow{\alpha} \textbf{compose}\{Behavior' \textbf{and replicate}\ Behaviour\}}$$

$$\textbf{T-Repl-com}: \frac{Behavior \xrightarrow{\textbf{restrict}\ n_1:T_1,\ldots,n_n:T_n.\textbf{via}\ \text{name}\ \textbf{send}\ \text{value}} Behavior' \quad Behavior \xrightarrow{\textbf{via}\ \text{name}\ \textbf{receive}\ \text{value}} Behavior''}{\textbf{replicate}\ Behavior \xrightarrow{\tau} \textbf{compose}\{\textbf{restrict} n_1 : T_1, \ldots, n_n : T_n.\textbf{compose}\{Behavior'\ \textbf{and}\ Behavior''\}\ \textbf{and replicate}\ Behavior\}}\{n_1, \ldots, n_n\} \cap \text{fn}(Behavior) = \emptyset$$

Figure 14: Sémantique formelle des comportements : communication, restriction et réplication.

Labeled transition rules

$$\textbf{Output}: \frac{}{\textbf{via}\ \text{connection}\ \textbf{send}\ \text{value}.behavior \xrightarrow{\textbf{via}\ \text{connection}\ \textbf{send}\ \text{value}} behavior}$$

$$\textbf{Input}: \frac{}{\textbf{via}\ \text{connection}\ \textbf{receive}\ \text{value}.behavior \xrightarrow{\textbf{via}\ \text{connection}\ \textbf{receive}\ \text{value}} behavior\ \textbf{where}\ \text{value} := \text{value'}}$$

$$\textbf{Unobservable}: \frac{}{\textbf{unobservable}.behavior \xrightarrow{\tau} behavior}$$

$$\textbf{Conditional-Then}: \frac{behavior_1 \xrightarrow{action} behavior_1' \quad \text{boolean} \equiv \text{true}}{\textbf{if}\ \text{boolean}\ \textbf{then}\ behavior_1\ \textbf{else}\ behavior_2 \xrightarrow{action} behavior_1'}$$

$$\textbf{Conditional-Else}: \frac{behavior_2 \xrightarrow{action} behavior_2' \quad \text{boolean} \equiv \text{false}}{\textbf{if}\ \text{boolean}\ \textbf{then}\ behavior_1\ \textbf{else}\ behavior_2 \xrightarrow{action} behavior_2'}$$

$$\textbf{Choice-Left}: \frac{behavior_1 \xrightarrow{action} behavior_1'}{\textbf{choose}\{behavior_1\ \textbf{or}\ behavior_2\} \xrightarrow{action} behavior_1'}$$

$$\textbf{Choice-Right}: \frac{behavior_2 \xrightarrow{action} behavior_2'}{\textbf{choose}\{behavior_1\ \textbf{or}\ behavior_2\} \xrightarrow{action} behavior_2'}$$

$$\textbf{Concurrent-Left}: \frac{behavior_1 \xrightarrow{action} behavior_1' \quad \text{bound(action)} \cap \text{free}(behavior_2) = \emptyset}{\textbf{compose}\{behavior_1\ \textbf{and}\ behavior_2\} \xrightarrow{action} \textbf{compose}\{behavior_1'\ \textbf{and}\ behavior_2\}}$$

$$\textbf{Concurrent-Right}: \frac{behavior_1 \xrightarrow{action} behavior_1' \quad \text{bound(action)} \cap \text{free}(behavior_1) = \emptyset}{\textbf{compose}\{behavior_1\ \textbf{and}\ behavior_2\} \xrightarrow{action} \textbf{compose}\{behavior_1\ \textbf{and}\ behavior_2'\}}$$

$$\textbf{Communication-Left}: \frac{behavior_1 \xrightarrow{\textbf{via}\ \text{name}\ \textbf{send}\ \text{value'}} behavior_1' \quad behavior_2 \xrightarrow{\textbf{via}\ \text{name}\ \textbf{receive}\ \text{value}} behavior_2'}{\textbf{compose}\ \{behavior_1\ \textbf{and}\ behavior_2\} \xrightarrow{\tau} \textbf{compose}\{behavior_1'\ \textbf{and}\ behavior_2'\ \textbf{where}\ \text{value} := \text{value'}\}}$$

$$\textbf{Communication-Right}: \frac{behavior_1 \xrightarrow{\textbf{via}\ \text{name}\ \textbf{send}\ \text{value'}} behavior_1' \quad behavior_2 \xrightarrow{\textbf{via}\ \text{name}\ \textbf{receive}\ \text{value}} behavior_2'}{\textbf{compose}\ \{behavior_1\ \textbf{and}\ behavior_2\} \xrightarrow{\tau} \textbf{compose}\{behavior_1'\ \textbf{where}\ \text{value} := \text{value'}\ \textbf{and}\ behavior_2'\}}$$

$$\textbf{Communication-Close-Left}: \frac{behavior_1 \xrightarrow{\textbf{restricted}\ \text{value'}.\textbf{via}\ \text{connection}\ \textbf{send}\ \text{value'}} behavior_1' \quad behavior_2 \xrightarrow{\textbf{via}\ \text{connection}\ \textbf{receive}\ \text{value}} behavior_2' \quad \text{value} \notin \text{free}(behavior_2)}{\textbf{compose}\ \{behavior_1\ \textbf{and}\ behavior_2\} \xrightarrow{\tau} \textbf{restricted}\ \text{value}.\textbf{compose}\{behavior_1'\ \textbf{and}\ behavior_2'\ \textbf{where}\ \text{value} := \text{value'}\}}$$

$$\textbf{Communication-Close-Right}: \frac{behavior_1 \xrightarrow{\textbf{via}\ \text{connection}\ \textbf{receive}\ \text{value}} behavior_1' \quad behavior_2 \xrightarrow{\textbf{restricted}\ \text{value'}.\textbf{via}\ \text{connection}\ \textbf{send}\ \text{value'}} behavior_2' \quad \text{value} \notin \text{free}(behavior_1)}{\textbf{compose}\ \{behavior_1\ \textbf{and}\ behavior_2\} \xrightarrow{\tau} \textbf{restricted}\ \text{value}.\textbf{compose}\{behavior_1'\ \textbf{where}\ \text{value} := \text{value'}\ \textbf{and}\ behavior_2'\}}$$

$$\textbf{Restriction}: \frac{behavior \xrightarrow{action} behavior' \quad \text{value} \notin \text{names}(action)}{\textbf{restricted}\ \text{value}.behavior \xrightarrow{\alpha} \textbf{restricted}\ \text{value}.behavior'}$$

$$\textbf{Restriction-Open}: \frac{behavior \xrightarrow{\textbf{via}\ \text{connection}\ \textbf{send}\ \text{value}} behavior' \quad \text{connection} \neq \text{value}}{\textbf{restricted}\ \text{value}.behavior \xrightarrow{\textbf{restricted}\ \text{value}.\textbf{via}\ \text{connection}\ \textbf{send}\ \text{value}} behavior'}$$

$$\textbf{Replication-Action}: \frac{behavior \xrightarrow{action} behavior'}{\textbf{replicate}\ behavior \xrightarrow{action} \textbf{compose}\{behavior'\textbf{and replicate}\ behaviour\}}$$

$$\textbf{Replication-Communication}: \frac{behavior \xrightarrow{\textbf{via}\ \text{connection}\ \textbf{send}\ \text{value'}} behavior' \quad behavior \xrightarrow{\textbf{via}\ \text{connection}\ \textbf{receive}\ \text{value}} behavior''}{\textbf{replicate}\ behavior \xrightarrow{\tau} \textbf{compose}\{\textbf{compose}\{behavior'\ \textbf{and}\ behavior''\ \textbf{where}\ \text{value} := \text{value'}\}\ \textbf{and replicate}\ behavior\}}$$

$$\textbf{Replication-Close}: \frac{behavior \xrightarrow{\textbf{restricted}\ \text{value}.\textbf{via}\ \text{connection}\ \textbf{send}\ \text{value}} behavior' \quad behavior \xrightarrow{\textbf{via}\ \text{connection}\ \textbf{receive}\ \text{value}} behavior'' \quad \text{value} \notin \text{free}(behavior)}{\textbf{replicate}\ behavior \xrightarrow{\tau} \textbf{compose}\{\textbf{restricted}\ \text{value}.\textbf{compose}\{behavior'\ \textbf{and}\ behavior''\ \textbf{where}\ \text{value} := \text{value'}\}\ \textbf{and replicate}\ behavior\}}$$

Figure 15: Sémantique formelle des comportements : actions basiques, choix et composition.