Exploration concrétisée et pertinente de systèmes d'événements abstraits en vue de la génération automatique de tests

Guillaume Voiron

THÈSE DE DOCTORAT DE L'ÉTABLISSEMENT UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

PRÉPARÉE À L'UNIVERSITÉ DE FRANCHE-COMTÉ

École doctorale n°37

Sciences Pour l'Ingénieur et Microtechniques

Doctorat d'Informatique

par

GUILLAUME VOIRON

Exploration concrétisée et pertinente de systèmes d'événements abstraits en vue de la génération automatique de tests

Thèse présentée et soutenue à Besançon, le 27 Septembre 2019

Composition du Jury :

LE GALL PASCALE	Professeur à CentraleSupélec	Rapportrice
LEDRU YVES	Professeur à l'Université de Grenoble Alpes	Rapporteur
KOSMATOV NIKOLAÏ	Ingénieur chercheur au CEA LIST Saclay	Examinateur
KOUCHNARENKO OLGA	Professeur à l'Université de Bourgogne-Franche-Comté	Examinatrice
JULLIAND JACQUES	Professeur émérite	Directeur de thèse
MASSON PIERRE-ALAIN	Maître de conférences à l'Université de Bourgogne-Franche-Comté	Codirecteur de thèse

N° | X | X | X |

école doctorale **sciences pour l'ingénieur et microtechniques**

Titre : Exploration concrétisée et pertinente de systèmes d'événements abstraits en vue de la génération automatique de tests

Mots-clés : Systèmes réactifs, Abstraction de modèles, Test à partir de modèles, Génération de tests pertinents, Couverture structurelle

Résumé :

Les travaux présentés dans cette thèse constituent une contribution aux méthodes de génération automatique de sous-approximations en vue de la génération de tests à partir de modèles. Le test à partir de modèle a pour objectif de garantir la conformité d'une implémentation vis-à-vis d'un modèle, tous les deux conçus à partir des spécifications par deux équipes différentes.

Dans cette thèse, nous proposons l'utilisation des techniques connues d'abstraction à partir de prédicats de modèles comportementaux qui permettent de réduire à un ensemble fini et restreint l'espace d'états manipulé. Nous proposons d'extraire les prédicats d'abstraction à partir de l'objectif de test afin que les tests générés couvrent les comportements ciblés par ce dernier. Cependant, le calcul d'une abstraction entraîne une perte d'information de l'atteignabilité par rapport au modèle initial. Nos objectifs sont donc dans un premier temps de calculer efficacement une abstraction de modèle aussi représentative que possible d'un objectif de test. Dans un second temps, nous cherchons à extraire à partir de cette abstraction des exécutions instanciables sur le modèle avant abstraction, ciblant les comportements à tester, et visant la couverture des états et des transitions du modèle abstrait.

Nos contributions sont les suivantes. Nous définissons une méthode de génération de tests combinant plusieurs algorithmes qui permettent d'obtenir une bonne couverture structurelle d'une abstraction de modèle comportemental non déterministe. Nous proposons dans un premier temps un algorithme calculant une abstraction de modèle par prédicats issus d'un objectif de test exprimé sous la forme d'une propriété temporelle. Cet algorithme calcule une sous-approximation du modèle en couvrant les états et les transitions abstraits du modèle. Il applique plusieurs heuristiques et diverses techniques d'exploration ayant pour but d'augmenter le nombre d'instances effectivement atteintes. Dans un second temps, nous proposons d'améliorer par le biais de deux autres algorithmes la couverture structurelle obtenue par cette première sous-approximation. Le premier, entièrement automatisé, tire parti des modalités des transitions abstraites qui fournissent des propriétés d'atteignabilité. Le second algorithme d'extension de la sous-approximation fait appel à l'expertise du testeur qui doit, à partir des transitions non couvertes, énoncer un prédicat de pertinence qui guide et limite l'exploration et l'instanciation réalisées. Nous définissons un ensemble de règles permettant d'énoncer ce prédicat de pertinence et de calculer un variant garantissant la terminaison de l'algorithme d'exploration. Ces deux algorithmes complètent la sous-approximation obtenue auparavant par des exécutions instanciables. Enfin, nous mettons en œuvre une démarche expérimentale pour l'évaluation de la qualité de la méthode, portant sur cinq études de cas.

UBFC UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

Université Bourgogne Franche-Comté
32, avenue de l'Observatoire
25000 Besanon, France

SOMMAIRE

I

Introduction et état de l'art

1

INTRODUCTION

Cette thèse présente une contribution à la génération automatique de tests à partir de modèles (en anglais Model-Based Testing, souvent abrégé par « MBT ») pour des systèmes de grande taille ou infinis.

Les systèmes ciblés par les contributions présentées dans cette thèse sont les systèmes réactifs et en particulier les systèmes événementiels modélisés à l'aide du langage B [Abrial, 1996]. Ils diffèrent des programmes séquentiels classiques [Hoare, 1969] par leur flot de contrôle implicite : les événements d'un système événementiel sont déclenchés spontanément lorsque leur condition de déclenchement est atteinte. Par opposition, l'ordre dans lequel s'exécutent les instructions d'un programme séquentiel sera toujours le même avec les mêmes entrées et la simulation du programme permet donc de connaître cet ordre. Dans le cas des systèmes événementiels, les événements sont déclenchés en fonction des évolutions de l'environnement, et plusieurs événements pourraient être déclenchés dans un état donné du système : les systèmes événementiels sont non-déterministes.

La génération automatique de tests à partir de modèles consiste à étudier formellement un modèle de système afin d'en déduire des entrées et des sorties permettant d'observer le fonctionnement d'une implémentation du système dans des situations particulières. Ces situations sont généralement caractérisées par un objectif de test. Les tests (les entrées et les sorties du programme) doivent ainsi couvrir, selon un ou plusieurs critère(s) de couverture bien défini(s), cet objectif de test.

1.1/ OBSTACLES ET DÉFIS

L'automatisation du processus de test et l'étude formelle de programmes présente plusieurs avantages par rapport à l'écriture manuelle de tests qui motivent l'intérêt des chercheurs à innover dans ces domaines. Cependant, un certain nombre d'obstacles et de défis restent à surmonter.

La pertinence, vis-à-vis d'un objectif de test, de tests exprimés manuellement n'est pas garantie ou est en tout cas difficile à évaluer. L'écriture de tests vise à mettre en évidence l'existence d'erreurs, signe que les tests sont de bonne qualité, mais cet objectif est particulièrement compliqué à remplir, notamment si le système concerné est très grand. La fiabilité de l'implémentation du système est ainsi incertaine, même lorsque les tests sont écrits par des personnes expérimentées.

Les travaux de cette thèse constituent ainsi une contribution à la génération automatique de tests pertinents vis-à-vis d'un objectif de test particulier. Les différents algorithmes proposés dans cette optique sont cependant conçus pour être efficaces et donc utilisables avec des systèmes de grande taille, ou infinis.

1.1.1/ TEST EXHAUSTIF IMPOSSIBLE EN PRATIQUE

L'écriture de tests exhaustifs consisterait à observer un système dans tous les comportements qu'il est susceptible d'avoir, en tenant compte de toutes les configurations et de tous les paramètres environnementaux auxquels il peut être soumis. L'écriture manuelle de tests non exhaustifs est déjà très coûteuse en temps et l'écriture manuelle de tests exhaustifs se révèle donc impossible à mettre en œuvre en pratique. Il en va de même pour la génération automatique de tests : un système trop complexe serait impossible à tester de manière exhaustive par un procédé automatique à moins de disposer d'une puissance de calcul non accessible encore aujourd'hui. Dans le cadre d'un système infini d'ailleurs, le test exhaustif est littéralement impossible puisque le système peut effectuer une infinité d'exécutions.

1.1.2/ CONCEPTION DE TESTS COÛTEUSE

La génération automatique de tests permet de réduire la charge de travail du testeur et de limiter ainsi le coût en temps employé au test d'un système. Les méthodes formelles actuellement employées pour effectuer l'analyse, la vérification, la validation et le test d'un système restent cependant relativement coûteuses, si ce n'est en temps d'exécution, au moins en termes de ressources de calcul. Que ce soit pour un processus de production de tests automatique ou manuel, il reste d'ailleurs à la charge du testeur de spécifier ce qui doit être intensivement testé et ce qui doit l'être dans une moindre mesure.

La génération de tests à partir de modèles réduit en revanche considérablement les coûts liés à la maintenance d'un système. En effet, un changement dans le fonctionnement du système ne nécessite que de modifier le modèle en conséquence pour qu'il soit conforme à ce changement. Les tests sont ensuite une nouvelle fois générés automatiquement à partir du nouveau modèle. Cela évite l'écriture manuelle de nouveaux tests à chaque changement dans les exigences décrivant le système. De plus, si les tests étaient conformes aux précédentes exigences, il est possible et même probable que certains ne soient plus conformes aux nouvelles exigences. La génération automatique de tests à partir d'un modèle à jour vis-à-vis des exigences représente donc un gain de temps non négligeable pour l'ingénieur de test, ce qui représente l'un des intérêts de cette méthode.

1.1.3/ CONCEPTION DE TESTS PERTINENTS ET SUFFISANTS

Une méthode de génération automatique de tests s'efforcera de garantir la qualité des tests qu'elle calcule. La qualité d'une suite de tests (un ensemble de tests) pourrait être évaluée selon trois caractéristiques :

- sa capacité à mettre en évidence des erreurs d'implémentation d'un système, ce qui est difficile à évaluer car des tests qui ne mettraient pas en évidence d'erreurs ne sont pas nécessairement mauvais pour autant,

- sa pertinence vis-à-vis d'un objectif de test, c'est-à-dire la capacité des tests qui le composent à couvrir, selon les critères de couverture spécifiés, les situations ciblées par l'objectif de test,
- le coût d'exécution des tests qui la composent : pour deux suites de tests générant la même couverture de l'objectif de test, on préférera généralement celle dont l'exécution est la moins coûteuse.

En pratique, les méthodes de génération de tests cherchent à atteindre des objectifs différents et l'évaluation de la qualité de ces tests peut ne dépendre qu'en partie des critères énoncés précédemment, ou d'aucun d'entre eux.

1.2/ QUESTIONS DE RECHERCHE

De ces limitations et défis concernant la génération de tests peuvent être extraites plusieurs questions de recherche.

Q_1 *Comment générer des tests pour des systèmes infinis ?*

Certains systèmes peuvent posséder un espace d'états infini, c'est-à-dire que l'ensemble de situations (ou états) dans lesquels il peut se trouver est infini. L'application des méthodes formelles, notamment de génération de tests, à un tel système, ou à un système dont l'espace d'état est très grand, est plus difficile à mettre en œuvre. Afin de pouvoir appliquer les contributions de cette thèse à des systèmes infinis, on aura recours en particulier à la technique d'abstraction par prédicats [Graf et al., 1997].

Q_2 *Comment abstraire un système infini ou de grande taille en conservant ses comportements pertinents vis-à-vis d'un objectif de test ?*

L'abstraction d'un système permet de calculer un système fini à partir d'un système infini au prix généralement d'une perte d'information sur les états pouvant effectivement être observés sur ce système. Cette perte d'information pourrait masquer les comportements du système qu'on souhaiterait couvrir par les tests qui en seront issus. On cherchera donc à calculer des abstractions de systèmes qui conservent les comportements dont l'observation est pertinente vis-à-vis d'un objectif de test.

Q_3 *Comment générer efficacement des tests ?*

Malgré le recours aux techniques d'abstraction, il peut être impossible de considérer et manipuler l'ensemble des états, qui peut rester grand, d'un système abstrait. Les algorithmes de génération de tests constituant les contributions de cette thèse auront donc pour objectif de limiter les comportements du système abstrait couverts par les tests, tout en favorisant leur conformité avec les critères de couverture visés.

Q_4 *Comment générer des tests pertinents vis-à-vis d'un objectif de test ?*

Le fait de constituer une abstraction pertinente vis-à-vis d'un objectif particulier ne suffit pas toujours à garantir que les tests qui en sont issus seront eux-mêmes pertinents. La génération des tests à partir d'abstraction doit donc également être guidée dans l'optique de couvrir l'objectif de test ou des objectifs intermédiaires liés à cet objectif de test.

Q_5 *Qu'est-ce qu'une bonne couverture d'une abstraction ?*

Il existe de nombreux critères de couverture d'une abstraction de système : on peut par exemple vouloir générer des tests permettant d'observer tous les événements, ou l'espace d'état complet de ce système abstrait, ou les deux, etc. Le problème de déterminer la qualité de la couverture d'une abstraction et d'un objectif de test reste un problème ouvert auquel on tentera d'apporter des éléments de réponse.

1.3/ PLAN DE THÈSE

Ce mémoire est décomposé en trois parties.

La **première partie** décrit le sujet, le contexte scientifique des travaux présentés ainsi que les travaux et concepts existants constituant le cadre dans lequel cette thèse s'inscrit.
Le chapitre courant, le **chapitre 1**, est consacré à l'introduction du sujet ainsi qu'à l'énonciation des problématiques qui en découlent.
Le **chapitre 2** présente le contexte scientifique nécessaire à la compréhension des contributions de cette thèse. Les structures, les concepts importants et les notations manipulé(e)s tout au long de cette thèse y sont introduits.
Le **chapitre 3** présente le cadre dans lequel les travaux de cette thèse s'inscrit. Il constitue un état de l'art en détaillant les travaux existants dans la littérature et dont les objectifs s'apparentent aux nôtres.

La **deuxième partie** détaille les contributions de cette thèse à la génération de tests à partir d'abstraction de modèles.
Le **chapitre 4** présente une méthode (appelée CXP) efficace de calcul d'une sous-approximation d'un système événementiel. A partir de cette sous-approximation, on génère des tests dont l'objectif est de couvrir tous les états et toutes les transitions d'un système abstrait au moins une fois.
Le **chapitre 5** présente une méthode (appelée BCI) faisant usage d'informations d'atteignabilité des séquences de transitions qui constituent un système abstrait. A l'aide de ces informations, BCI complète la sous-approximation obtenue avec la méthode CXP.
Le **chapitre 6** propose un méthode (appelée RCXP) permettant de couvrir les états et transitions d'un système abstrait qui n'auraient pas été couvertes avec CXP. RCXP complète la sous-approximation calculée par CXP par des états et transitions jugés pertinents par le testeur.
Le **chapitre 7** propose une méthode à suivre pour savoir quel(s) algorithme(s) de calcul d'une sous-approximation appliquer selon la situation et comment l'appliquer.
Le **chapitre 8** présente l'outil de génération de tests développé pour appliquer les différents algorithmes constituant les contributions de cette thèse et évaluer expérimentalement ces méthodes. Ce chapitre présente également les résultats obtenus avec cet outil sur un ensemble de systèmes événementiels modélisés en B événementiel.

La **troisième partie** conclut ce mémoire et expose les perspectives des travaux présentés. Le **chapitre 9** présente une conclusion générale sur les contributions et leur capacité à répondre aux questions de recherche. Le **chapitre 10** suggère des pistes de recherche à explorer dans la continuité des travaux effectués lors de cette thèse.

2

CONTEXTE SCIENTIFIQUE

Ce chapitre décrit le contexte scientifique sur lequel les travaux présentés dans cette thèse s'appuient. Les systèmes événementiels sont le cadre formel pour modéliser des systèmes réactifs. Ce sont les principaux systèmes sur lesquels s'appliquent les contributions de cette thèse. Ils sont présentés en section 2.1 avec un exemple fil rouge qui permettra d'illustrer les définitions et concepts utilisés sur l'ensemble de cette thèse. La section 2.2 présente le concept d'abstraction par prédicats qui permet de concentrer l'étude sur une partie d'un système événementiel et non sur l'ensemble de ses comportements modélisés. L'abstraction par prédicats engendre des systèmes dit *abstraits*. Structurellement, il s'agit de systèmes de transitions modaux qui sont présentés en section 2.3. Les travaux effectués dans le cadre de cette thèse visent à couvrir les états et les transitions qui composent les systèmes de transitions modaux selon divers critères de couverture présentés en section 2.5. Enfin, la section 2.6 présente le langage de propriétés temporelles utilisé pour exprimer des objectifs de test sur les systèmes événementiels.

N.B. Dans cette thèse, tous les acronymes utilisés pour raccourcir et simplifier les notations proviennent de la version en anglais des concepts qu'ils décrivent. Par exemple, les systèmes événementiels seront notés ES pour « Event Systems ».

2.1/ SYSTÈMES ÉVÉNEMENTIELS : SYNTAXE, SÉMANTIQUE ET EXEMPLE

Les contributions de cette thèse sont prévues pour être appliquées à une classe de systèmes modélisés par des systèmes événementiels décrits dans le langage B [Abrial, 1996] et présentés par la définition 1. Cette classe est plus généralement celle des systèmes de transitions qui sont la sémantique des systèmes événementiels.

Définition 1 : Système Événementiel (ES)

Un système événementiel est un 4-uplet $\langle X, Inv, Init, Ev \rangle$ où :

- X est un ensemble de variables d'état,
- *Inv* est un invariant que chaque état du système doit respecter,
- *Init* est une action d'initialisation des variables d'état du système,
- Ev est un ensemble de définitions d'événements sous la forme $e \stackrel{def}{=} a$, où e est le nom de l'événement et a est sa définition (l'action qu'il effectue sur les variables d'état du système).

L'invariant permet a minima de typer les variables d'état mais peut également exprimer

des contraintes plus fortes sur les valeurs relatives les unes aux autres que peuvent prendre les différentes variables.

L'application de l'action d'initialisation attribue une valeur à chaque variable d'état du système qui se trouve dans un état dit *concret* (voir définition 2) et *initial* (puisqu'il résulte de l'application de l'action d'initialisation). Il peut exister plusieurs états initiaux pour un même système événementiel étant donné que les actions d'un tel système peuvent donner lieu à du non-déterminisme (en particulier l'action d'initialisation).

Définition 2 : État concret d'un système événementiel

Soient ES $\stackrel{def}{=} \langle X, Inv, Init, Ev \rangle$ un système événementiel.
Un état d'un système événementiel associe à chaque variable x de X une valeur dans son domaine de définition, noté $Dom(x)$. Cet état est également considéré comme la proposition définie comme la conjonction de ces valuations, dénotée $\bigwedge_{x \in X} x = v_x$ avec $v_x \in Dom(x)$.

Les actions d'un système événementiel, qu'il s'agisse de l'action d'initialisation ou de l'action opérée par un événement, modifient la valeur d'une ou plusieurs variables d'état du système et mènent donc le système dans un nouvel état concret. Elles appliquent donc une ou plusieurs substitutions sur les variables d'état du système, et ce de manière atomique, c'est-à-dire que toutes les substitutions qu'elles opèrent sont effectuées instantanément. Dans le langage B événementiel, les substitutions sont construites à partir de six substitutions dites primitives présentées par la définition 3.

Définition 3 : Substitutions primitives en B événementiel [Abrial, 1996, Abrial, 2010]

Soient S, S_1 et S_2 et S_{i_j} (avec $i, j \in \mathbb{N}$) des substitutions généralisées, les substitutions généralisées du langage de spécifications B sont de l'une des six formes suivantes :

1. La substitution neutre $skip$ qui n'affecte pas l'état auquel elle est appliquée.
2. La substitution simple $x := E$ qui modifie la valuation (ici par la valeur de l'expression E) d'une unique variable d'état (ici x) de l'état auquel elle est appliquée.
3. La substitution multiple $x, y, ... := E, F, ...$ qui modifie la valuation (ici par $E, F, ...$ respectivement) de plusieurs variables d'état (ici $x, y, ...$) de l'état auquel elle est appliquée. Cette substitution est également notée $x := E \parallel y := F \parallel ....$
4. La substitution gardée $P \Rightarrow S$ qui applique une substitution généralisée (ici S) à l'état auquel elle est appliquée à condition que la garde P soit satisfaite par cet état.
5. Le choix borné non déterministe $S_1 \; [] \; S_2 \; [] \; ... \; [] \; S_n$ avec $n \in \mathbb{N}$ qui applique, de manière non déterministe, l'une des substitutions généralisées qui le composent (ici S_1 ou S_2 ou ... ou S_n) dont la garde est satisfaite par l'état auquel le choix est appliqué.
6. Le choix non borné et non déterministe $@i.S$ qui correspond aux choix non déterministe $S_{i_0} \; [] \; S_{i_1} \; [] \; ...$ pour toutes les valeurs de $i \in \{i_0, i_1, ...\}$ satisfaisant la garde de la substitution S.

L'atomicité des substitutions réalisées par les actions est importante car les événements d'un système événementiel sont déclenchés spontanément et de manière non déterministe lorsque le système se trouve dans un état satisfaisant leur garde. Une fois qu'un événement est déclenché, aucun autre événement n'est donc déclenché avant que toutes les substitutions opérées par son action n'aient été réalisées. Les gardes des sub-

stitutions primitives, notées grd(S) où S est une substitution primitive, sont présentées par la définition 4.

Définition 4 : Gardes des substitutions primitives [Dijkstra et al., 1990]

Les gardes des substitutions primitives sont les suivantes :

1. $\texttt{grd}(skip) \stackrel{def}{=} vrai$,
2. $\texttt{grd}(x := E) \stackrel{def}{=} vrai$,
3. $\texttt{grd}(x, y, ... := E, F, ...) \stackrel{def}{=} vrai$,
4. $\texttt{grd}(P \Rightarrow S) \stackrel{def}{=} P \wedge \texttt{grd}(S)$,
5. $\texttt{grd}(S_1 \ [] \ S_2 \ [] \ ... \ [] \ S_n) \stackrel{def}{=} \texttt{grd}(S_1) \vee \texttt{grd}(S_2) \vee ... \vee \texttt{grd}(S_n)$,
6. $\texttt{grd}(@i.S) \stackrel{def}{=} \exists(i).(\texttt{grd}(S))$.

Certaines substitutions peuvent être effectuées sans contraintes. C'est le cas de la substitution $skip$, qui n'effectue aucune modification des valeurs des variables d'état du système et de la substitution simple et multiple. La garde de ces substitutions est donc toujours $vraie$. Comme son nom l'indique, la substitution gardée $P \Rightarrow S$ requiert en revanche que la garde P soit vérifiée par l'état dans lequel le système se trouve pour que la substitution S puisse être appliquée, à condition que la garde de cette dernière soit également vérifiée. Dans le cas du choix borné non-déterministe, il faut que la garde de l'une des substitutions qui la composent soit vérifiée pour que cette substitution soit appliquée. Si les gardes de deux substitutions ou plus sont satisfaites, l'une de ces substitutions est appliquée de manière non-déterministe. Enfin, le choix borné non déterministe n'est appliqué que s'il existe une variable liée satisfaisant la garde de la substitution qui le compose.

On associe à chaque substitution généralisée un *prédicat avant-après* (noté $prd_X(S)$ où S est une substitution généralisée). Ce prédicat s'exprime sur un couple d'états consécutifs : l'état (ou les états) dans lequel le système doit se trouver *avant* que la substitution puisse être appliquée et l'état (ou les états) dans lesquels le système se trouvera *après* son application. Le concept de prédicat avant-après est présenté par la définition 5.

2.1.1/ SATISFIABILITÉ MODULO THÉORIES

L'implémentation d'un algorithme utilisant la technique d'abstraction par prédicats repose généralement sur l'emploi de solveurs de contraintes dits SMT (pour Satisfiabilité Modulo Théories). De tels solveurs permettent de déterminer la satisfiabilité ou la non-satisfiabilité de formules exprimées en logique du premier ordre. Une formule est dite *satisfiable* s'il est possible de trouver une interprétation (on parle alors de *modèle*) qui la rend vraie.

Soit ϕ une formule logique du premier ordre. On utilise dans la suite de ce mémoire la notation $\texttt{SAT}(\phi)$ pour déterminer si la formule ϕ est *satisfiable*. $\texttt{SAT}(\phi)$ masque donc un appel à un solveur SMT qui retourne soit un modèle lorsque ϕ est satisfiable, soit $unsat$ lorsque ϕ n'est pas satisfiable, soit $unknown$ si le solveur n'est pas parvenu à déterminer si ϕ était satisfiable ou non.

Définition 5 : Prédicat avant-après

Un prédicat avant-après est un prédicat portant sur un ensemble de variables d'état d'un système. Il contraint les valuations possibles des variables *avant* et *après* application d'une substitution.
Soit X un ensemble de variables d'état d'un système événementiel. On utilise le nom de la variable pour désigner cette variable *avant* application d'une substitution et la notation primée pour désigner cette variable *après* application d'une substitution.
Les prédicats avant-après des substitutions généralisées sont alors définis comme suit :

- $prd_X(skip) \stackrel{def}{=} \bigwedge_{x \in X} x' = x$,
- $prd_X(x := E) \stackrel{def}{=} x' = E \wedge \bigwedge_{y \in X - \{x\}} y' = y$,
- $prd_X(x, y, ... := E, F, ...) \stackrel{def}{=} x' = E \wedge y' = F \wedge ... \wedge \bigwedge_{z \in X - \{x,y,...\}} z' = z$,
- $prd_X(P \Rightarrow S) \stackrel{def}{=} P \wedge prd_X(S)$,
- $prd_X(S_1 \,[]\, S_2) \stackrel{def}{=} prd_X(S_1) \vee prd_X(S_2)$,
- $prd_X(@i.S) \stackrel{def}{=} \exists(i).prd_X(S)$ si i n'est pas modifiée par S,
- $prd_X(@i.S) \stackrel{def}{=} \exists(i, i').prd_{X \cup \{i\}}(S)$ si i est modifiée par S et intervient donc dans la relation $prd_{X \cup \{i\}}(S)$.

Par exemple, soit $\langle X, Inv, Init, Ev \rangle$ un système événementiel où :

- $X \stackrel{def}{=} \{x, y\}$,
- $Inv \stackrel{def}{=} x \in \{0..1\} \wedge y \in \{0..1\}$,
- $Init \stackrel{def}{=} x := 0 \parallel y := 1$,
- $Ev \stackrel{def}{=} \{e \stackrel{def}{=} x = 0 \wedge y = 1 \Rightarrow x := y \parallel y := x\}$.

Le prédicat avant-après de l'unique événement de ce système est le suivant :

$$prd_X(e) \stackrel{def}{=} x = 0 \wedge y = 1 \wedge x' = y \wedge y' = x$$

La notation $(c, c') := \texttt{SAT}(Inv \wedge Inv[^{X'}/_X] \wedge prd_X(e))$ indique alors que le tuple (c, c') est le modèle retourné par l'appel au solveur SMT avec la formule $prd_X(e)$ tout en respectant l'invariant Inv et l'invariant dans lequel toutes les variables de X sont primées (noté $Inv[^{X'}/_X]$). En l'occurrence, le modèle retourne une valuation pour chaque variable non primée ($x = 0$ et $y = 1$) et pour chaque variable primée ($x' = y = 1$ et $y' = x = 0$).

L'état concret c est alors la conjonction des valuations des variables non primées du modèle, et c' est la conjonction des valuations des variables primées du modèle. On aura ainsi $c \stackrel{def}{=} x = 0 \wedge y = 1$ et $c' \stackrel{def}{=} x = 1 \wedge y = 0$. On note que même si c' correspond à la conjonction des valuations des variables primées dans le modèle, il s'agit quand même d'un état concret et donc les valuations portent sur les variables de X, et non pas sur les variables de X primées.

Si le solveur SMT n'est pas parvenu à déterminer la satisfiabilité de la formule $Inv \wedge Inv[^{X'}/_X] \wedge prd_X(e)$ ou qu'il a déterminé que la formule n'était pas satisfiable, on considère que (c, c') vaut $unknown$ ou $unsat$ (donc $(c, c') \in \{unknown, unsat\}$) afin de simplifier les notations.

N.B. Afin d'alléger les notations, tous les appels à SAT, on considère que l'invariant *Inv* et l'invariant primé $Inv[^{X'}/_X]$ font implicitement partie de la formule dont la satisfiabilité doit être vérifiée. En effet, tout état retourné dans le modèle doit se conformer à l'invariant. Ainsi, $\texttt{SAT}(prd_X(e))$ correspond à l'appel au solveur suivant : $\texttt{SAT}(Inv \wedge Inv[^{X'}/_X] \wedge prd_X(e))$.

2.1.2/ SÉMANTIQUE D'UN SYSTÈME ÉVÉNEMENTIEL

La sémantique d'un système événementiel est définie par un système de transitions étiquetées (noté LTS pour « Labelled Transition System »). La définition 6 présente le concept standard de LTS qui est un quadruplet $\langle C, C_0, L, R\rangle$ où C_0 est un ensemble d'états initiaux, C est un ensemble d'états contenant C_0, L est un ensemble d'étiquettes et R est une relation de transitions étiquetées qui est un sous-ensemble de $C \times L \times C$. Une transition est un triplet (c, e, c') (on note aussi $c \xrightarrow{e} c'$) où c est un état source, c' un état cible et e une étiquette indiquant le nom de l'événement appliqué pour effectuer le changement d'état de c à c'.

Définition 6 : Système de Transitions Étiquetées (LTS)

Un système de transitions étiquetées est un 4-uplet $\langle C, C_0, L, R\rangle$ où :

- C est un ensemble d'états
- $C_0 \subseteq C$ est un ensemble d'états initiaux
- L est un ensemble d'étiquettes
- $R \subseteq C \times L \times C$ est une relation de transitions

La sémantique d'un système événementiel est le LTS qui lui est associé comme indiqué par la définition 7 issue des travaux de [Bert et al., 2000]. Son ensemble d'états concrets est l'ensemble des n-uplets de valuations de X qui satisfont l'invariant du système événementiel. L'invariant définit le domaine de valeurs de chaque variable de X ainsi que des contraintes spécifiques au cas d'étude.

Définition 7 : Sémantique d'un système événementiel

Étant donné un système événementiel ES $\stackrel{def}{=} \langle X, Inv, Init, Ev\rangle$, sa sémantique est un LTS $\stackrel{def}{=} \langle C, C_0, L, R\rangle$ où :

- $C \stackrel{def}{=} \{ \bigwedge_{x\in X} x = v_x \mid v_x \in Dom(x) \wedge \mathtt{SAT}(\bigwedge_{x\in X} x = v_x \wedge Inv)\}$ est l'ensemble des états concrets qui satisfont l'invariant *Inv*,
- $C_0 \stackrel{def}{=} \{c \mid c \in C \wedge \mathtt{SAT}(prd_X(Init) \wedge c[{}^{X'}/_X]) = sat\}$ est l'ensemble des états concrets initiaux obtenus par application de la substitution d'initialisation *Init*,
- $L \stackrel{def}{=} \{e \mid e \stackrel{def}{=} a \in Ev\}$ est l'ensemble des noms d'événements de ES,
- $R \stackrel{def}{=} \{c \xrightarrow{e} c' \mid c \in C \wedge e \stackrel{def}{=} a \in Ev \wedge c' \in C \wedge \mathtt{SAT}(prd_X(a) \wedge c'[{}^{X'}/_X] \wedge c) = sat\}$ est la relation de transitions concrètes.

Définition 8 : Atteignabilité concrète

Un état concret c' est dit *atteignable* si une transition $c_0 \xrightarrow{e} c'$ avec $c_0 \in C_0$ appartient à la clôture transitive et réflexive de R. Une transition $c \xrightarrow{e} c' \in R$ est également dite *atteignable* si son état source c est lui-même atteignable.
On qualifie les états concrets et les transitions concrètes d'atteint(e)s lorsqu'il existe une séquence de transitions du CTS qui y mène en partant d'un état concret initial (un état de C_0).

Les états initiaux d'un système événementiel sont les états c atteints (voir définition 8) par application de la substitution d'initialisation *Init*. L'état c étant l'état cible, il doit être primé pour être consistant avec le prédicat avant-après de *Init*. La notation $c[{}^{X'}/_X]$ définit un état cible obtenu à partir de l'état c en primant chacune des variables. Par exemple, étant donné l'état $c \stackrel{def}{=} x_1 = v_1 \wedge ... \wedge x_n = v_n$ avec $n = |X|$, $c[{}^{X'}/_X]$ est l'état $x'_1 = v_1 \wedge ... \wedge x'_n = v_n$.

Les transitions du LTS sémantique d'un système événementiel sont tous les triplets (c, e, c') qui satisfont le prédicat avant-après pour chaque événement $e \stackrel{def}{=} a$. Autrement dit, c satisfait la garde de a ($\texttt{grd}(a)$) et l'application de a à c mène à l'état c'.

2.1.3/ EXEMPLE FIL ROUGE

Afin d'illustrer les définitions et les concepts déjà vus et à venir dans la suite de cette thèse, on présente ici un exemple fil rouge de système événementiel. Cet exemple, présenté par la figure 2.1, modélise le comportement d'un distributeur de café en langage B événementiel.

X	$\stackrel{def}{=}$	$\{Balance, Pot, Status, CofLeft, AskCof, AskChange\}$
Inv	$\stackrel{def}{=}$	$Pot \in 0..MAX\_POT + 50 \wedge Balance \in 0..MAX\_BAL \wedge$ $CofLeft \in 0..MAX\_COF \wedge Pot \ mod\ 50 = 0 \wedge Balance \ mod\ 50 = 0 \wedge$ $Status \in \{off, on, error\} \wedge AskCof \in \{false, true\} \wedge AskChange \in \{false, true\} \wedge$ $AskChange = true \Rightarrow (Balance > 0 \wedge AskCof = false) \wedge$ $AskCof = true \Rightarrow (Balance \geq 50 \wedge AskChange = false) \wedge$ $Balance = 0 \Rightarrow (AskCof = false \wedge AskChange = false)$
$Init$	$\stackrel{def}{=}$	$Balance := 0 \parallel Pot := 0 \parallel CofLeft := 10 \parallel$ $Status := off \parallel AskCof := false \parallel AskChange := false$
- $insert50$	$\stackrel{def}{=}$	$Status = on \wedge AskChange = false \wedge AskCof = false \wedge$ $Balance + 50 \leq MAX\_BAL \Rightarrow Balance := Balance + 50$
- $insert100$	$\stackrel{def}{=}$	$Status = on \wedge AskChange = false \wedge AskCof = false \wedge$ $Balance + 100 \leq MAX\_BAL \Rightarrow Balance := Balance + 100$
- $powerUp$	$\stackrel{def}{=}$	$Status = off \wedge CofLeft > 0 \wedge Pot \leq MAX\_POT \Rightarrow$ $Status := on \parallel Balance := 0 \parallel AskCof := false \parallel AskChange := false$
- $powerDown$	$\stackrel{def}{=}$	$(Status = on \wedge AskChange = false \wedge AskCof = false \wedge Balance = 0) \vee$ $Status = error \Rightarrow Status := off$
- $autoOut$	$\stackrel{def}{=}$	$Status = on \Rightarrow Status := error$
- $takePot$	$\stackrel{def}{=}$	$Status = off \wedge Pot \geq MAX\_POT - 50 \Rightarrow Pot := 0$
- $cofReq$	$\stackrel{def}{=}$	$Status = on \wedge Balance \geq 50 \wedge AskCof = false \wedge$ $AskChange = false \Rightarrow AskCof := true$
- $changeReq$	$\stackrel{def}{=}$	$Status = on \wedge Balance > 0 \wedge AskCof = false \wedge$ $AskChange = false \Rightarrow AskChange := true$
- $addCof$	$\stackrel{def}{=}$	$\exists x.(x \in 1..MAX\_COF \wedge CofLeft + x \leq MAX\_COF \wedge$ $Status = off \Rightarrow CofLeft := CofLeft + x)$
- $serveCof$	$\stackrel{def}{=}$	$Status = on \wedge Balance \geq 50 \wedge AskCof = true \wedge CofLeft > 0 \wedge Pot \leq MAX\_POT \Rightarrow$ $AskCof := false \parallel Balance := Balance - 50 \parallel$ $CofLeft := CofLeft - 1 \parallel Pot := Pot + 50 \parallel$ $(Pot \geq MAX\_POT \vee CofLeft = 1 \Rightarrow Status := error\ []$ $Pot + 50 \leq MAX\_POT \wedge CofLeft \neq 1 \Rightarrow skip) \parallel$ $(Balance > 50 \Rightarrow AskChange := true\ []\ Balance = 50 \Rightarrow skip)$
- $backBalance$	$\stackrel{def}{=}$	$Status = on \wedge Balance > 0 \wedge AskChange = true \Rightarrow Balance := 0 \parallel AskChange := false$

FIGURE 2.1 – Spécification d'un distributeur de café en B événementiel

Les constantes utilisées par ce système, dont la valeur est fixée arbitrairement, ont la signification suivante :

- *MAX_BAL* : il s'agit de la quantité maximale d'argent que l'utilisateur peut insérer dans le distributeur avant de demander un café ou de récupérer son argent,
- *MAX_POT* : il s'agit de la quantité maximale d'argent que la machine peut contenir,
- *MAX_COF* : il s'agit du nombre maximal de cafés pouvant être stockés (et vendus) par la machine.

Les variables d'état de ce système sont celles de l'ensemble nommé X :

- *Balance* $\in 0..$*MAX_BAL* : une variable indiquant la somme d'argent insérée par l'utilisateur dans la machine,
- *Pot* $\in 0..$*MAX_POT*$+50$: une variable indiquant la somme d'argent récoltée par la vente de cafés,
- *Status* $\in \{$*off*, *on*, *error*$\}$: une variable indiquant l'état de fonctionnement de la machine (*off* signifiant qu'elle est arrêtée, *on* signifiant qu'elle fonctionne, et *error* signifiant qu'une erreur s'est produite),
- *CofLeft* $\in 0..$*MAX_COF* : une variable indiquant le nombre de cafés restant dans le distributeur,
- *AskCof* $\in \{$*false*, *true*$\}$: une variable indiquant si l'utilisateur a demandé du café (*true*) ou non (*false*),
- *AskChange* $\in \{$*false*, *true*$\}$: une variable indiquant si l'utilisateur a demandé à récupérer la somme insérée dans la machine sans effectuer d'achat (*true*) ou non (*false*).

L'invariant *Inv* sert à typer les variables d'état et ajoute notamment les contraintes suivantes :

- L'utilisateur ne peut introduire que des pièces multiples de 50 unités et le pot ne peut contenir qu'une somme multiple de 50. Il s'agit d'une simplification de la machine pour ne pas modéliser la gestion de divers types de pièces.
- L'utilisateur ne peut pas demander à récupérer son argent tant qu'il n'en a pas inséré dans le distributeur ou s'il vient de demander un café et que le distributeur ne lui a pas encore délivré.
- L'utilisateur ne peut pas demander de café tant qu'il n'a pas inséré suffisamment d'argent dans le distributeur ou s'il vient de demander à récupérer l'argent qu'il a inséré.

La substitution d'initialisation *Init* initialise la *Balance* et le *Pot* à 0 (*Balance* $:= 0 \parallel$ *Pot* $:= 0$), le nombre de cafés dans le distributeur à *MAX_COF* (*CofLeft* $:=$ *MAX_COF*), met le distributeur à l'arrêt (*Status* $:=$ *off*) et considère qu'aucun café n'a encore été demandé (*AskCof* $:=$ *false*) et que l'utilisateur n'a pas demandé à récupérer son argent (*AskChange* $:=$ *false*).

Les événements modélisés simulent à la fois les actions réalisées par le distributeur lui-même, par le technicien chargé de la maintenance de la machine, ou encore par un utilisateur du distributeur.

L'utilisateur peut insérer de l'argent dans le distributeur (50 ou 100 unités d'argent par le biais des événements *insert50* et *insert100* respectivement) puis demander un café (par activation de l'événement *cofReq*) dont le prix est fixé à 50 unités. Il peut également demander à récupérer l'argent qu'il a inséré dans le distributeur (par le biais de l'événement *changeReq*) dans le cas où il ne souhaite plus de café.

Le distributeur délivre un café (*serveCof*) à l'utilisateur si et seulement si il a demandé un café, s'il reste des cafés en stock et si au moins 50 unités d'argent ont été insérées par l'utilisateur. Le distributeur rend l'argent inséré à l'utilisateur (*backBalance*) si celui-ci l'a réclamé (par l'événement *changeReq*) et rend la monnaie si l'utilisateur a demandé un café et introduit une somme supérieure à 50 unités.

Le technicien peut mettre la machine en état de marche (*powerUp*) ou l'arrêter (*powerDown*) pour ajouter du café (*addCof*), récupérer l'argent gagné grâce aux ventes de cafés (*takePot*), ou intervenir si la machine a rencontré une erreur (simulée par *autoOut* ou produite

lorsque le pot est plein ou que le dernier café a été délivré).

En cas d'erreur, le distributeur n'accepte plus d'argent de la part de l'utilisateur tant que le technicien ne l'a pas remise en état de marche.

Exemple 1 : Illustration des concepts liés aux systèmes événementiels

Dans cet exemple, les concepts liés aux systèmes événementiels sont illustrés sur l'exemple fil rouge du distributeur de café.

La garde de l'événement *powerUp* se détermine ainsi :

$$\begin{array}{lcl} \texttt{grd}(\textit{powerUp}) & \stackrel{def}{=} & \textit{Status} = \textit{off} \wedge \textit{CofLeft} > 0 \wedge \textit{Pot} \leq \textit{MAX\_POT} \wedge \\ & & \texttt{grd}(\textit{Status} := \textit{on} \parallel \textit{Balance} := 0 \parallel \textit{AskCof} := \textit{false} \parallel \textit{AskChange} := \textit{false}) \\ & = & \textit{Status} = \textit{off} \wedge \textit{CofLeft} > 0 \wedge \textit{Pot} \leq \textit{MAX\_POT} \wedge \textit{true} \\ & = & \textit{Status} = \textit{off} \wedge \textit{CofLeft} > 0 \wedge \textit{Pot} \leq \textit{MAX\_POT} \end{array}$$

L'événement *powerUp* applique une substitution gardée dont la garde est $\texttt{grd}(\textit{powerUp})$ et la substitution appliquée lorsque cette garde est satisfaite est la substitution multiple suivante : $\textit{Status} := \textit{on} \parallel \textit{Balance} := 0 \parallel \textit{AskCof} := \textit{false} \parallel \textit{AskChange} := \textit{false}$.

Le prédicat avant-après correspondant à la substitution gardée appliquée par l'événement *powerUp* se détermine donc ainsi :

$$\begin{array}{lcl} prd_X(\textit{powerUp}) & \stackrel{def}{=} & \textit{Status} = \textit{off} \wedge \textit{CofLeft} > 0 \wedge \textit{Pot} \leq \textit{MAX\_POT} \wedge \\ & & prd_X(\textit{Status} := \textit{on} \parallel \textit{Balance} := 0 \parallel \textit{AskCof} := \textit{false} \parallel \textit{AskChange} := \textit{false}) \\ & \stackrel{def}{=} & \textit{Status} = \textit{off} \wedge \textit{CofLeft} > 0 \wedge \textit{Pot} \leq \textit{MAX\_POT} \wedge \\ & & \textit{Status'} = \textit{on} \wedge \textit{Balance'} = 0 \wedge \textit{AskCof'} = \textit{false} \wedge \textit{AskChange'} = \textit{false} \wedge \\ & & \textit{Pot'} = \textit{Pot} \wedge \textit{CofLeft'} = \textit{CofLeft} \end{array}$$

N.B. Seules les variables *Status*, *Balance*, *AskCof* et *AskChange* sont modifiées par la substitution multiple appliquée par *powerUp*. Les variables restantes *Pot* et *CofLeft* restent en revanche inchangées, ce qu'exprime la condition $\textit{Pot'} = \textit{Pot} \wedge \textit{CofLeft'} = \textit{CofLeft}$.

Soient les états concrets suivants :

- $c_0 \stackrel{def}{=} \textit{Balance} = 0 \wedge \textit{Pot} = 0 \wedge \textit{Status} = \textit{off} \wedge \textit{CofLeft} = 10 \wedge \textit{AskCof} = \textit{false} \wedge \textit{AskChange} = \textit{false}$
- $c_1 \stackrel{def}{=} \textit{Balance} = 0 \wedge \textit{Pot} = 0 \wedge \textit{Status} = \textit{on} \wedge \textit{CofLeft} = 10 \wedge \textit{AskCof} = \textit{false} \wedge \textit{AskChange} = \textit{false}$

On note que c_0 est un état initial du système car il correspond exactement à l'application de la substitution d'initialisation $\textit{Init} \stackrel{def}{=} \textit{Balance} := 0 \parallel \textit{Pot} := 0 \parallel \textit{Status} := \textit{off} \parallel \textit{CofLeft} := 10 \parallel \textit{AskCof} := \textit{false} \parallel \textit{AskChange} := \textit{false}$. Cette substitution d'initialisation étant déterministe et portant sur toutes les variables d'état du système, c_0 est en réalité le seul état initial du système.

La substitution gardée correspondant à l'événement *powerUp* peut être appliquée à c_0 car il satisfait sa garde $\texttt{grd}(\textit{powerUp})$ (le *Status* est *off*, *CofLeft* vaut 10 ce qui est supérieur à 0, et le *Pot* vaut 0 ce qui est inférieur à *MAX_POT*).

L'application de cette substitution à c_0 ne modifie pas la valeur du *Pot* ni de *CofLeft* qui restent donc respectivement à 0 et 10. En revanche, elle modifie les valeurs des autres variables (comme exprimé par $prd_X(\textit{powerUp})$) en mettant le *Status* à *on*, la *Balance* à 0, *AskCof* à *false* et *AskChange* à *false*. L'état concret atteint par l'application de *powerUp* à c_0 est donc c_1 car les valeurs de *Pot* et *CofLeft* ne changent pas de c_0 à c_1, et les valeurs

respectives du *Status*, de la *Balance*, de *AskCof* et de *AskChange* sont fixées à *on*, 0, *false* et *false* dans c_1.

L'application de *powerUp* à c_1 est en revanche impossible car la garde de *powerUp* (grd(*powerUp*)) requiert que le *Status* soit *off*, ce qui n'est pas le cas dans c_1 où le *Status* est *on*.

La figure 2.2 montre une partie du LTS sémantique du distributeur de café sous la forme d'un graphe.

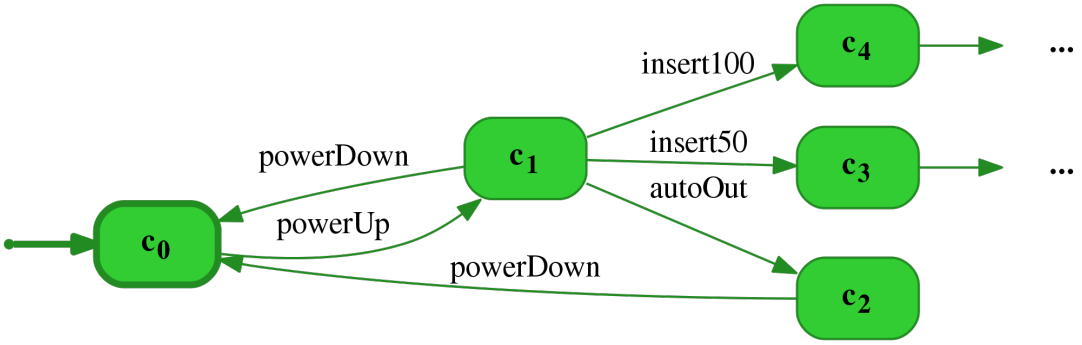


FIGURE 2.2 – LTS sémantique partiel du distributeur de café

Les états concrets (de c_0 à c_4) sont représentés par des rectangles aux coins arrondis. L'état initial c_0 est mis en évidence par une bordure plus épaisse que les autres états concrets et par la flèche (tout à gauche) ayant pour état cible c_0 et n'ayant aucun état source. Les transitions sont représentées par des flèches allant d'un état à un autre et portant le nom de l'événement qu'elles appliquent.

On obtient ainsi le LTS partiel $\langle C, C_0, L, R\rangle$ où :

- $C = \{c_0, c_1, c_2, c_3, c_4\}$,
- $C_0 = \{c_0\}$,
- $L = \{$*insert50*, *insert100*, *powerUp*, *powerDown*, *autoOut*, *takePot*, *cofReq*, *changeReq*, *addCof*, *serveCof*, *backBalance*$\}$,
- $R = \{\ c_0 \xrightarrow{powerUp} c_1, c_1 \xrightarrow{powerDown} c_0, c_1 \xrightarrow{autoOut} c_2, c_2 \xrightarrow{powerDown} c_0, c_1 \xrightarrow{insert100} c_4, c_1 \xrightarrow{insert50} c_3\ \}$

Dans l'objectif de générer des tests, calculer le LTS sémantique complet d'un système n'est pas envisageable en raison de son très grand nombre d'états et de transitions potentiels. Afin de limiter une explosion de l'espace d'états considéré, on propose l'utilisation de la technique d'abstraction par prédicats présentée par la section 2.2.

2.2/ ABSTRACTION PAR PRÉDICATS

L'abstraction par prédicats [Graf et al., 1997] est une technique qui permet de réduire l'espace d'états à traiter et de le ramener à un ensemble fini (même dans le cas où l'ensemble d'états concrets est infini). Avec des prédicats d'abstraction bien choisis, le système abs-

trait permet de se focaliser sur les états et transitions ayant un intérêt particulier vis-à-vis d'un objectif de test ou d'une propriété du modèle à vérifier par exemple.

L'abstraction par prédicat nécessite un ensemble non vide $\mathcal{P} \stackrel{def}{=} \{p_1, ..., p_n\}$ de $n \in \mathbb{N}$ prédicats d'abstraction présentés par la définition 9.

Définition 9 : Prédicat d'abstraction

Un prédicat d'abstraction est un prédicat exprimé en logique du premier ordre et portant sur les variables d'état du système à abstraire.

Les prédicats d'abstraction permettent de construire des classes d'équivalence pour tous les états concrets d'un système événementiel. Ces classes d'équivalence constituent des états dits *abstraits* présentés par la définition 10.

Définition 10 : Etat abstrait

Soit $\mathcal{P} \stackrel{def}{=} \{p_1, ..., p_n\}$ un ensemble de prédicats d'abstraction.
Un état abstrait est un n-uplet $q \stackrel{def}{=} (q_1, q_2, ..., q_n)$ avec q_i égal à $\neg p_i$ ou à p_i pour $i \in 1..n$.
Un état abstrait est également considéré comme le prédicat $\bigwedge_{i=1}^{n} q_i$.

Par la suite, on note A l'ensemble des états abstraits. Par définition d'un état abstrait, cet ensemble contient 2^n états, où n est le nombre de prédicats d'abstraction.

On associe à un état concret d'un système l'état abstrait auquel il correspond (i.e. la classe d'équivalence à laquelle il appartient) par le biais d'une fonction d'abstraction présentée par la définition 11.

Définition 11 : Fonction d'abstraction

Soient C un ensemble d'états concrets et A un ensemble d'états abstraits.
Une fonction d'abstraction est une fonction totale $\alpha : C \to A$ qui à un état concret c de C associe l'état abstrait q de A qu'il satisfait. Ainsi, $\alpha(c)$ est l'état abstrait $q \stackrel{def}{=} (q_1, q_2, ..., q_n)$ tel que $c \wedge q_i$ est une formule valide pour tout $i \in 1..n$.

Par abus de langage, on dit que q contient c, qu'un état concret c est *dans* l'état abstrait q, ou encore que c est un état concret de q, lorsque $\alpha(c) = q$.

La définition 12 présente la notion de système d'états et de transitions abstraits

Définition 12 : Système abstrait

Soit $\mathcal{P}$ un ensemble de prédicats d'abstraction.
Le système abstrait issu de $\mathcal{P}$ est un 4-uplet $\langle A, A_0, L, R \rangle$ où :

- A est l'ensemble des états abstraits constitués à partir de $\mathcal{P}$,
- $A_0 \subseteq A$ est un ensemble d'états abstraits initiaux,
- L est un ensemble d'étiquettes,
- $R \subseteq A \times L \times A$ est une relation de transitions abstraites.

Une transition $(q, e, q') \in R$ est dite abstraite (car q et q' sont des états abstraits). Elle est notée $q \stackrel{e}{\to} q'$.

L'exemple 2 illustre, à l'aide du distributeur de café, les principaux concepts liés à l'abstraction par prédicats. La notion de système abstrait n'est pas illustrée ici mais sera exemplifiée dans la section 2.3.

Exemple 2 : Illustration des concepts d'abstraction par prédicats

On utilise l'ensemble de prédicats d'abstraction $\mathcal{P} \stackrel{def}{=} \{p_0, p_1, p_2\}$ où :

- $p_0 \stackrel{def}{=}$ *Status* = *off* $\wedge$ *Pot* $\geq$ *MAX_POT*-50 (garde de *takePot*),
- $p_1 \stackrel{def}{=}$ *Status* = *on* (garde de *autoOut*),
- $p_2 \stackrel{def}{=}$ (*Status* = *on* $\wedge$ *AskChange* = *false* $\wedge$ *AskCof* = *false* $\wedge$ *Balance* = 0) $\vee$ *Status* = *error* (garde de *powerDown*).

L'ensemble A des états abstraits contient ainsi $2^{|\mathcal{P}|} = 2^3 = 8$ états, notés q_i avec $i \in 0..7$ où :

- $q_0 \stackrel{def}{=} \neg p_0 \wedge \neg p_1 \wedge \neg p_2$
- $q_1 \stackrel{def}{=} \neg p_0 \wedge \neg p_1 \wedge p_2$
- $q_2 \stackrel{def}{=} \neg p_0 \wedge p_1 \wedge \neg p_2$
- $q_3 \stackrel{def}{=} \neg p_0 \wedge p_1 \wedge p_2$
- $q_4 \stackrel{def}{=} p_0 \wedge \neg p_1 \wedge \neg p_2$
- $q_5 \stackrel{def}{=} p_0 \wedge \neg p_1 \wedge p_2$
- $q_6 \stackrel{def}{=} p_0 \wedge p_1 \wedge \neg p_2$
- $q_7 \stackrel{def}{=} p_0 \wedge p_1 \wedge p_2$

N.B. Tous les états abstraits doivent également satisfaire l'invariant pour appartenir au système abstrait. De plus, les états abstraits dont la formule logique qui leur correspond ne peut pas être satisfaite sont dits *vides*. Dans cet exemple, les états abstraits suivants sont vides :

- q_5 car le *Status* ne peut pas être à la fois égal à *off* comme requis par p_0 et égal à *on* ou *error* comme requis par p_2
- q_6 et q_7 car le *Status* ne peut pas être à la fois égal à *off* comme requis par p_0 et égal à *on* comme requis par p_1

On considère généralement que A ne contient que les états abstraits non vides : $A = \{q_0, q_1, q_2, q_3, q_4\}$.

Soient les états concrets suivants :

- $c_5 \stackrel{def}{=}$ *Status* = *error* $\wedge$ *AskChange* = *false* $\wedge$ *AskCof* = *false* $\wedge$ *Balance* = 0 $\wedge$ *Pot* = 100 $\wedge$ *CofLeft* = 0
- $c_6 \stackrel{def}{=}$ *Status* = *on* $\wedge$ *AskChange* = *true* $\wedge$ *AskCof* = *false* $\wedge$ *Balance* = 0 $\wedge$ *Pot* = 100 $\wedge$ *CofLeft* = 10
- $c_7 \stackrel{def}{=}$ *Status* = *on* $\wedge$ *AskChange* = *false* $\wedge$ *AskCof* = *false* $\wedge$ *Balance* = 0 $\wedge$ *Pot* = 100 $\wedge$ *CofLeft* = 10
- $c_8 \stackrel{def}{=}$ *Status* = *on* $\wedge$ *AskChange* = *false* $\wedge$ *AskCof* = *false* $\wedge$ *Balance* = 0 $\wedge$ *Pot* = 500 $\wedge$ *CofLeft* = 0
- $c_9 \stackrel{def}{=}$ *Status* = *off* $\wedge$ *AskChange* = *false* $\wedge$ *AskCof* = *false* $\wedge$ *Balance* = 0 $\wedge$ *Pot* = 50 $\wedge$ *CofLeft* = 0
- $c_{10} \stackrel{def}{=}$ *Status* = *off* $\wedge$ *AskChange* = *false* $\wedge$ *AskCof* = *false* $\wedge$ *Balance* = 0 $\wedge$ *Pot* = *MAX_POT* $\wedge$ *CofLeft* = 0

L'association réalisée par la fonction d'abstraction α sur ces états sera alors la suivante :

- $\alpha(c_5) = q_1$ car, dans c_5, le *Status* n'est ni *off* ni *on* comme requis par p_0 et p_1 respectivement mais vaut *error*, ce qui suffit à satisfaire le prédicat p_2.
- $\alpha(c_6) = q_2$ car, dans c_6, le *Status* n'est pas *off* comme requis par p_0, vaut *on* comme requis par p_1 et le booléen *AskChange* vaut *true*, ce qui rend le prédicat p_2 non satisfiable puisqu'il requiert soit que *AskChange* soit *true*, soit que le *Status* soit *error*.
- $\alpha(c_7) = \alpha(c_8) = q_3$ car, dans c_7 et c_8, le *Status* n'est pas *off* comme requis par p_0, vaut *on* comme requis par p_1 et toutes les conditions imposées par la première partie de la disjonction de p_2 (*Status* = *on* $\wedge$ *AskChange* = *false* $\wedge$ *AskCof* = *false* $\wedge$ *Balance* = 0) sont vraies.
- $\alpha(c_9) = q_0$ car, dans c_9, le *Status* est *off* mais le *Pot* est inférieur à *MAX_POT - 50*, ce

qui ne satisfait pas p_0, le *Status* n'est pas *on* comme requis par p_1 et le *Status* n'est ni *on*, ni *error* comme requis par p_2.

- $\alpha(c_{10}) = q_4$ car, dans c_{10}, le *Status* est *off* et le *Pot* est égal à *MAX_POT* comme requis par p_0, le *Status* n'est pas *on* comme requis par p_1, et le *Status* n'est ni *on* ni *error* comme requis par p_2.

Les systèmes abstraits permettent donc de concentrer une étude sur une ou plusieurs propriétés d'un système événementiel en sélectionnant des prédicats d'abstraction adaptés et de réduire l'espace d'états à un ensemble fini et réduit d'états *abstraits*. L'abstraction par prédicats entraîne en revanche une perte d'information sur l'atteignabilité des états et transitions abstraits : il peut ne pas exister d'état concret (respectivement de transition concrète) *atteignable* instanciant un état abstrait (respectivement une transition abstraite).

2.3/ SYSTÈMES DE TRANSITIONS MODAUX

On associe des modalités aux transitions abstraites afin de les classer en fonction de leur atteignabilité et de l'atteignabilité des états abstraits qui les composent. La notion de modalité est présentée par la définition 13.

Définition 13 : Modalité d'une transition abstraite

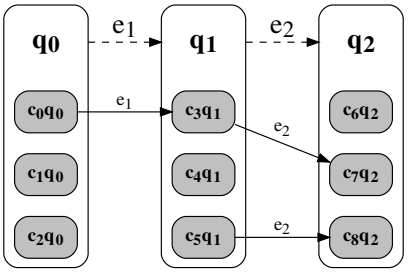
Soient un système événementiel ES $\stackrel{def}{=} \langle X, Inv, Init, Ev\rangle$, $\mathcal{P}$ un ensemble de prédicats d'abstraction et AS $\stackrel{def}{=} \langle A, A_0, L, R\rangle$ un système abstrait.
On considère quatre modalités possibles pour une transition abstraite $q \xrightarrow{e} q' \in R$ définies comme suit :

1. La modalité may est attribuée à $q \xrightarrow{e} q'$ lorsqu'il existe une transition concrète $c \xrightarrow{e} c'$ du LTS sémantique de ES telle que $\alpha(c) = q$, $\alpha(c') = q'$.
2. La modalité $must^-$ est attribuée à $q \xrightarrow{e} q'$ lorsqu'elle possède la modalité may et que pour tout c' tel que $\alpha(c') = q'$, il existe un état concret c tel que $\alpha(c) = q$ et $c \xrightarrow{e} c'$ est une transition du LTS sémantique de ES.
3. La modalité $must^+$ est attribuée à $q \xrightarrow{e} q'$ lorsqu'elle possède la modalité may et que pour tout c tel que $\alpha(c) = q$, il existe un état concret c' tel que $\alpha(c') = q'$ et $c \xrightarrow{e} c'$ est une transition du LTS sémantique de ES.
4. La modalité $must^\#$ est attribuée à $q \xrightarrow{e} q'$ lorsqu'elle possède à la fois la modalité $must^-$ et $must^+$.

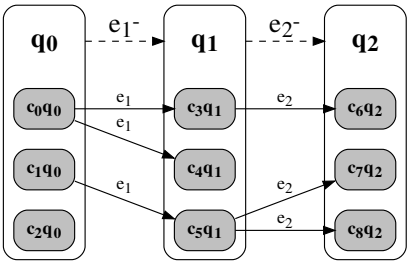
La figure 2.3 illustre les quatre modalités possibles pour une transition. La modalité may est attribuée à toute transition abstraite pour laquelle il existe au moins une instance concrète correspondante. La modalité $must^-$ est attribuée à toute transition abstraite pour laquelle tous les états concrets de l'état abstrait cible possèdent un prédécesseur dans l'état abstrait source par cette transition. La modalité $must^+$ est attribuée à toute transition abstraite pour laquelle tous les états concrets de l'état abstrait source possèdent un successeur dans l'état abstrait cible par cette transition.

Sur la figure 2.3a, il existe une instance ($c_0q_0 \xrightarrow{e_1} c_3q_1$) de la transition abstraite $q_0 \xrightarrow{e_1} q_1$ et deux instances ($c_3q_1 \xrightarrow{e_1} c_7q_2$ et $c_5q_1 \xrightarrow{e_1} c_8q_2$) de la transition $q_1 \xrightarrow{e_2} q_2$. Les transitions

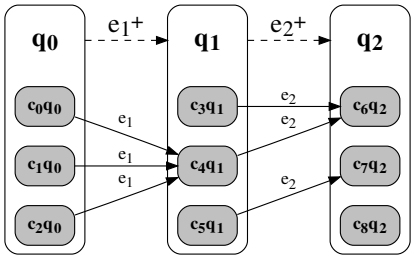
$q_0 \xrightarrow{e_1} q_1$ et $q_1 \xrightarrow{e_2} q_2$ sont donc des *may*-transitions.
Sur la figure 2.3b, tous les états concrets (c_3q_1, c_3q_1 et c_3q_1) de l'état abstrait q_1 sont la cible d'une instance de la transition abstraite $q_0 \xrightarrow{e_1} q_1$. De même, tous les états concrets (c_6q_2, c_7q_2 et c_8q_2) de l'état abstrait q_2 sont la cible d'une instance de la transition abstraite $q_1 \xrightarrow{e_2} q_2$. Les transitions $q_0 \xrightarrow{e_1} q_1$ et $q_1 \xrightarrow{e_2} q_2$ sont donc des $must^-$-transitions qu'on notera ainsi respectivement $q_0 \xrightarrow{e_1^-} q_1$ et $q_1 \xrightarrow{e_2^-} q_2$.
Sur la figure 2.3c, tous les états concrets (c_0q_0, c_1q_0 et c_2q_0) de l'état abstrait q_0 sont la source d'une instance de la transition abstraite $q_0 \xrightarrow{e_1} q_1$. De même, tous les états concrets (c_3q_1, c_4q_1 et c_5q_1) de l'état abstrait q_1 sont la source d'une instance de la transition abstraite $q_1 \xrightarrow{e_2} q_2$. Les transitions $q_0 \xrightarrow{e_1} q_1$ et $q_1 \xrightarrow{e_2} q_2$ sont donc des $must^+$-transitions qu'on notera respectivement $q_0 \xrightarrow{e_1^+} q_1$ et $q_1 \xrightarrow{e_2^+} q_2$.
Enfin, sur la figure 2.3d, les transitions abstraites $q_0 \xrightarrow{e_1} q_1$ et $q_1 \xrightarrow{e_2} q_2$ possèdent à la fois les propriétés d'une $must^-$-transition et d'une $must^+$-transition. Il s'agit donc de deux $must^\#$-transitions qu'on notera respectivement $q_0 \xrightarrow{e_1^\#} q_1$ et $q_1 \xrightarrow{e_2^\#} q_2$.



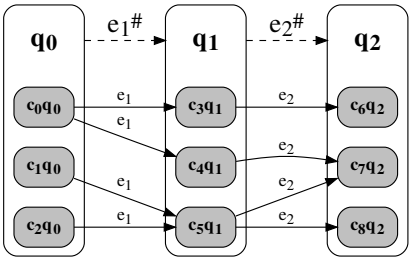
(a) Illustration d'une séquence de *may*-transitions composée des transitions $q_0 \xrightarrow{e_1} q_1$ et $q_1 \xrightarrow{e_2} q_2$



(b) Illustration d'une séquence de $must^-$-transitions composée des transitions $q_0 \xrightarrow{e_1^-} q_1$ et $q_1 \xrightarrow{e_2^-} q_2$



(c) Illustration d'une séquence de $must^+$-transitions composée des transitions $q_0 \xrightarrow{e_1^+} q_1$ et $q_1 \xrightarrow{e_2^+} q_2$



(d) Illustration d'une séquence de $must^\#$-transitions composée des transitions $q_0 \xrightarrow{e_1^\#} q_1$ et $q_1 \xrightarrow{e_2^\#} q_2$

FIGURE 2.3 – Illustration des quatre différentes modalités des transitions abstraites

On parle de may-transition, $must^-$-transition, $must^+$-transition et $must^\#$-transition et on note $q \xrightarrow{e} q'$, $q \xrightarrow{e^-} q'$, $q \xrightarrow{e^+} q'$ et $q \xrightarrow{e^\#} q'$ lorsqu'une transition a la modalité may, $must^-$, $must^+$ ou $must^\#$ respectivement.

Soient $q \xrightarrow{e} q'$ une transition abstraite et a la substitution appliquée par l'événement e. La modalité de $q \xrightarrow{e} q'$ est calculée par des appels à un solveur SMT. Elle a la modalité :

- may si et seulement si $\texttt{SAT}(q \wedge prd_X(a) \wedge q'[^{X'}/_X]) = sat$.
- $must^-$ si et seulement si $\texttt{SAT}(\neg\exists(X).(q \wedge prd_X(a)) \wedge q'[^{X'}/_X]) = unsat$.
- $must^+$ si et seulement si $\texttt{SAT}(q \wedge \neg\exists(X').(prd_X(a) \wedge q'[^{X'}/_X])) = unsat$.

La définition 14 présente les systèmes de transitions tri-modaux qui s'apparentent aux systèmes abstraits et attribuent des modalités à chaque transition qui les composent.

Définition 14 : Système de Transitions Tri-Modal (3MTS) associé à un ES et un ensemble de prédicats d'abstraction

Soient un système événementiel ES $\stackrel{def}{=} \langle X, Inv, Init, Ev\rangle$ à abstraire, $\mathcal{P}$ un ensemble de prédicats d'abstraction sur X et A l'ensemble des états abstraits issus de $\mathcal{P}$.
Le système de transitions tri-modal (noté 3MTS) associé à ES et $\mathcal{P}$ est un 5-uplet $\langle Q_0, Q, \Delta, \Delta^-, \Delta^+\rangle$ où :

- $Q_0 \stackrel{def}{=} \{q \mid q \in Q \wedge \texttt{SAT}(q[^{X'}/_X] \wedge prd_X(Init))\}$ est un ensemble fini d'états initiaux du système abstrait
- $Q \stackrel{def}{=} \{q \mid q \in A \wedge \exists(q').(q \xrightarrow{e} q' \in \Delta)\}$ est un ensemble fini d'états abstraits
- $\Delta \subseteq Q \times \{e \mid e \stackrel{def}{=} a \in Ev\} \times Q$ est la relation de may-transitions.
  Puisque les $must^-$-transitions, $must^+$-transitions et $must^\#$-transitions sont également des may-transitions, elles sont également dans Δ.
- $\Delta^- \subseteq \Delta$ est l'ensemble des $must^-$-transitions.
- $\Delta^+ \subseteq \Delta$ est l'ensemble des $must^+$-transitions.

L'ensemble des $must^\#$-transitions peut être construit par l'intersection entre Δ^- et Δ^+, raison pour laquelle il n'apparaît pas dans un 3MTS. De plus, l'ensemble des may-transitions n'étant ni des $must^-$-transitions ni des $must^+$-transitions peut être obtenu en retirant les éléments de Δ étant également contenus dans Δ^- et/ou Δ^+.

Dans le cas d'une étude où on ne s'intéresse qu'à la modalité may, les ensembles Δ^- et Δ^+ sont supprimés. On parle alors de système de may-transitions (noté MTS pour « May-Transitions System ») présenté par la définition 15.

Définition 15 : Système de May-Transitions (MTS) associé à un système événementiel et un ensemble de prédicats d'abstraction

Un système de may-transitions correspond au triplet $\langle Q_0, Q, \Delta\rangle$ d'un 3MTS $\langle Q_0, Q, \Delta, \Delta^-, \Delta^+\rangle$.

La notion de may-atteignabilité fait l'objet de la définition 16.

Définition 16 : May-atteignabilité

Un état may-atteignable d'un MTS $\langle Q_0, Q, \Delta\rangle$ est un état atteignable depuis un état de Q_0 par la relation de transitions Δ. Par extension, une transition d'un MTS est dite may-atteignable si son état source est may-atteignable.

On note qu'une transition peut-être may-atteignable mais ne pas être atteignable de manière concrète au sens de la définition 8. En effet, l'atteignabilité concrète d'une tran-

sition *may*-atteignable impose qu'au moins un état concret atteignable soit source d'une instance de cette transition. La relation de transition Δ ne garantit pas que ce soit le cas.

Indécidabilité du calcul du MTS atteignable : Le MTS atteignable à partir d'un système d'événements et d'un ensemble de prédicats d'abstraction est le MTS qui contient toutes les *may*-transitions atteignables concrètement et seulement celles-là. Il s'agit de la même notion que celle de « *true FSM* » dans le contexte des machines à états abstraits (FSM, Finite State Machine, voir section 2.1.2 dans [Grieskamp et al., 2002]). En conséquence, et comme il est prouvé dans [Grieskamp et al., 2002] (section 3.3), le calcul du MTS atteignable à partir d'un système d'événements et d'un ensemble de prédicats d'abstraction est un problème indécidable dans le cas général.

Soient ES un système événementiel, AS son système abstrait correspondant et α la fonction d'abstraction. On dit d'une transition concrète $c \xrightarrow{e} c'$ qu'elle *instancie* (ou qu'elle *correspond à*) une transition abstraite $q \xrightarrow{e} q'$ de AS lorsque $\alpha(c) = q$, $\alpha(c') = q'$ et $c \xrightarrow{e} c'$ appartient au LTS sémantique de ES. De même, on dit d'un état concret c qu'il est une *instance* d'un état abstrait q de AS si $\alpha(c) = q$ et c appartient au LTS sémantique de ES.

Les instances concrètes correspondant aux états abstraits et aux transitions abstraites d'un MTS constituent un *système concret de transitions*, noté CTS pour « Concrete Transitions System » et présenté par la définition 17. On dit alors qu'un CTS est la concrétisation du MTS.

Définition 17 : Système Concret de Transitions (CTS) associé à un MTS

Soit $\mathrm{M} \stackrel{def}{=} \langle Q_0, Q, \Delta \rangle$ un système de may-transitions.
Un système concret de transitions associé à M est un 3-uplet $\langle C_0, C, \Delta^c \rangle$ où :

- C est un ensemble d'instances des états de Q
- $C_0 \subseteq C$ est un ensemble d'instances des états de Q_0
- Δ^c est un ensemble d'instances des transitions de Δ

Les notions de sous-approximation et de sur-approximations sont primordiales dans le cadre de la vérification de systèmes et de la génération de tests. Ces concepts sont introduits par la définition 18.

Définition 18 : Sous-approximation et sur-approximation d'un modèle

Un MTS est une sur-approximation d'un modèle lorsque toute séquence de transitions concrètes constituant une exécution du modèle instancie une séquence de transitions *may*-atteignables du MTS.
Un MTS est une sous-approximation d'un modèle lorsque toute séquence de transitions *may*-atteignables du MTS peut être instanciée par une séquence de transitions concrètes constituant une exécution du modèle.
Par extension, un CTS $\langle C_0, C, \Delta^c \rangle$ issu d'un MTS est une sous-approximation d'un modèle si l'ensemble des états et des transitions qui le composent sont atteignables sur le modèle.

Dans le cadre de la vérification de propriétés sur un modèle, on cherchera généralement à calculer efficacement une sur-approximation du modèle et à démontrer que la propriété n'est pas violée dans cette sur-approximation. Cela permet de garantir que l'implémentation, si elle est conforme au modèle, ne violera pas non plus cette propriété. Dans le cadre du test en revanche, on cherche à générer des tests instanciables

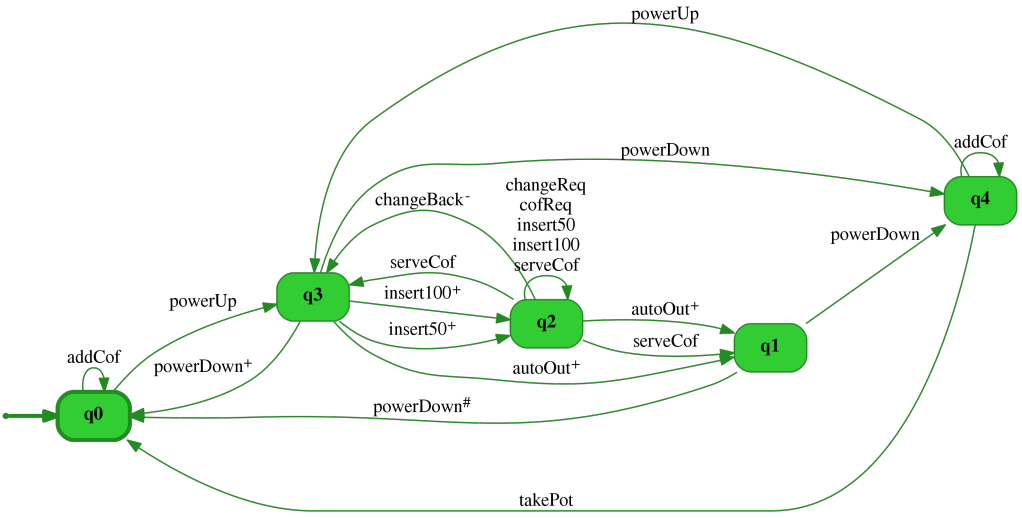


FIGURE 2.4 – Système de transitions tri-modal associé au distributeur de café et $\mathcal{P}$

sur l'implémentation du modèle, ce qui implique de calculer une sous-approximation du modèle plutôt qu'une sur-approximation.

Exemple 3 : Exemple de système de transitions tri-modal

Cet exemple décrit le système de transitions tri-modal associé au distributeur de café et à l'ensemble de prédicats d'abstraction $\mathcal{P} \stackrel{def}{=} \{p_0, p_1, p_2\}$ vu dans l'exemple 2. Le système de transitions tri-modal associé est représenté par le graphe de la figure 2.4.

On y retrouve les 5 états abstraits non vides de l'exemple 2 (de q_0 à q_4) sous la forme de rectangles aux coins arrondis. L'état abstrait initial est l'état q_0 dont les bordures sont plus épaisses que les autres états abstraits. Les transitions sont une nouvelle fois représentées par une flèche d'un état abstrait à un autre. Les symboles $^-$, $^+$ et $^\#$ apparaissant sur le nom de l'événement appliqué par une transition indiquent respectivement qu'elle a la modalité $must^-$, $must^+$ ou $must^\#$. Dans le cas ou aucun de ces symboles n'est présent sur un nom d'événement, la transition a la modalité may.

On obtient ainsi le 3MTS $\stackrel{def}{=} \langle Q_0, Q, \Delta, \Delta^-, \Delta^+ \rangle$ où :

- $Q_0 = \{q_0\}$,
- $Q = \{q_0, q_1, q_2, q_3, q_4\}$,
- $\Delta = \Delta^- \cup \Delta^+ \cup \{q_0 \xrightarrow{addCof} q_0, q_0 \xrightarrow{powerUp} q_3, q_1 \xrightarrow{powerDown} q_4, q_2 \xrightarrow{serveCof} q_1, q_2 \xrightarrow{serveCof} q_2, q_2 \xrightarrow{insert50} q_2, q_2 \xrightarrow{insert100} q_2, q_2 \xrightarrow{cofReq} q_2, q_2 \xrightarrow{changeReq} q_2, q_2 \xrightarrow{serveCof} q_3, q_3 \xrightarrow{powerDown} q_4, q_4 \xrightarrow{takePot} q_0, q_4 \xrightarrow{powerUp} q_3, q_4 \xrightarrow{addCof} q_4\}$,
- $\Delta^- \stackrel{def}{=} \{q_1 \xrightarrow{powerDown} q_0, q_2 \xrightarrow{changeBack} q_3\}$,
- $\Delta^+ \stackrel{def}{=} \{q_1 \xrightarrow{powerDown} q_0, q_2 \xrightarrow{autoOut} q_1, q_3 \xrightarrow{powerDown} q_0, q_3 \xrightarrow{autoOut} q_1, q_3 \xrightarrow{insert50} q_2, q_3 \xrightarrow{insert100} q_2\}$.

2.4/ SYSTÈMES DE TRANSITIONS APPROXIMÉS

La réunion d'un MTS et d'un CTS associé forme un *système de transitions approximé*, noté ATS pour « Approximated Transition System », présenté par la définition 19.

Définition 19 : Système de Transitions Approximé (ATS) associé à un ES et un ensemble de prédicats d'abstraction

Un système de transitions approximé est un 8-uplet $\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa \rangle$ où :

- $\langle Q_0, Q, \Delta \rangle$ est un système de may-transitions (MTS)
- $\langle C_0, C, \Delta^c \rangle$ est un système concret de transitions (CTS) qui est une concrétisation du MTS $\langle Q_0, Q, \Delta \rangle$
- $\alpha : C \to Q$ est une fonction d'abstraction
- $\kappa : C \to \{bleu, vert\}$ est une fonction de coloration des états concrets

La fonction κ associe une couleur à chaque état de C. Elle est utilisée par des heuristiques dans tous les algorithmes constituant les contributions de cette thèse pour améliorer la couverture par les tests des états abstraits et des transitions abstraites. Elle permet également de déterminer si un état ou une transition du CTS de l'ATS est atteignable sur le modèle. Cette heuristique est présentée en section 4.2 et s'inspire des travaux de [Veanes et al., 2003].

2.5/ CRITÈRES DE COUVERTURE DES SYSTÈMES À ÉTATS ET TRANSITIONS

Il est généralement souhaitable de générer des tests permettant d'obtenir une couverture du système étudié aussi exhaustive que possible. Une couverture exhaustive impliquerait que les tests correspondent à l'ensemble des exécutions possibles depuis les états initiaux de ce système. Pour raisonner en termes de systèmes à états et transitions, les tests seraient composés de l'ensemble des séquences de transitions existant pour ce graphe à partir de ses états initiaux. Clairement, dès qu'une boucle apparaît dans ce graphe, la taille de ces séquences et leur nombre peut être infini ; il en va de même si le système modélisé sous forme de graphe est infini.

Il est donc nécessaire de définir des critères de couverture différents du critère exhaustif. Les critères de couverture de systèmes à états et transitions les plus couramment utilisés sont les suivants :

- la couverture de tous les états : les tests doivent permettre de visiter l'ensemble des états du système,
- la couverture de toutes les transitions : les tests doivent permettre de visiter l'ensemble des transitions du système ; ce critère couvre en même temps tous les états, sauf s'il existe des états initiaux n'étant la source d'aucune transition,
- la couverture des paires de transitions : les tests doivent permettre de visiter l'ensemble des paires constituées de deux transitions successives (la cible de la première étant la source de la seconde) sur le système ; ce critère de couverture englobe le critère de couverture de toutes les transitions et permet en plus d'observer la causalité entre ces transitions,
- la couverture des k-chemins : les tests doivent permettre de visiter l'ensemble des séquences de transitions de taille $k \in \mathbb{N}$ ou moins ; clairement, la couverture des 0-chemins est identique à la couverture de tous les états, la couverture des 1-chemins est identique à la couverture de toutes les transitions et la couverture des 2-chemins est identique à la couverture des paires de transitions.

Dans le cas de la couverture des k-chemins, plus k est grand, plus on s'approche du critère de couverture exhaustif. Choisir un k très grand pour forcer les tests à couvrir une séquence de transitions particulière ou des états particuliers n'est pas forcément judicieux. Cela générera en effet des tests plus nombreux et dont une grande partie n'aura pas d'intérêt vis-à-vis de la séquence ou des états ciblés. Dans ce cas, il est donc préférable de déterminer un sous-ensemble *pertinent* de chemins à couvrir par les tests et de guider cette génération afin qu'elle couvre ce sous-ensemble.

2.6/ PROPRIÉTÉS TEMPORELLES

Les propriétés temporelles permettent d'exprimer des propriétés portant sur les exécutions d'un système. Ces propriétés sont très utilisées en vérification de modèle car elles expriment une spécification du système à laquelle le modèle doit se conformer. Dans le cadre de cette thèse, on utilisera les propriétés temporelles comme des objectifs de test : si le modèle doit se conformer à une propriété, l'implémentation du système doit également s'y conformer. Un exemple de propriété temporelle portant sur les exécutions du distributeur de café pourrait être la suivante : « aucun café ne peut être servi entre le moment où l'utilisateur demande à récupérer sa monnaie et en insère de nouveau dans la machine ». Autrement dit, l'événement *serveCof* n'est jamais appliqué entre l'application de *changeBack* et de *insert50* ou *insert100*. S'il existe une exécution déclenchant dans l'ordre les événements *powerUp*, *insert50*, *changeReq*, *changeBack*, *serveCof*, *insert50*, la propriété est violée car *serveCof* est appliqué entre *changeBack* est *insert50*.

Habituellement, les propriétés temporelles à vérifier sont décrites à l'aide des formalismes LTL, CTL, CTL* ou plus généralement du μ-calcul. Il peut cependant être très compliqué d'exprimer précisément la propriété souhaitée dans ces formalismes, même pour un expert du domaine. Généralement d'ailleurs, seule une partie du pouvoir d'expression des logiques temporelles est utilisé en pratique. Certaines constructions exprimant une dynamique particulière entre les états ou les événements reviennent régulièrement dans la littérature [Chan et al., 1998, Dillon et al., 1994, Dwyer et al., 1997, N. Naumovich et al., 1996, Wing et al., 1997]. Par conséquent, Matthew Dwyer propose dans [Dwyer et al., 1999] un langage de spécification permettant de s'affranchir des différents formalismes en proposant des patrons de spécification.

Les patrons de spécification permettent d'exprimer les propriétés temporelles les plus couramment rencontrées dans un large échantillon d'exemples par le biais de constructions simples et courantes qui possèdent leur équivalent dans tous les formalismes. La propriété temporelle est donc exprimée à l'aide d'un patron de spécification combiné avec une *portée*. La portée indique sur quelle portion des exécutions du système la formule exprimée par le patron doit être vérifiée.

2.6.1/ PATRONS DE SPÉCIFICATION

La classification, selon leur sémantique, des différentes propriétés temporelles pouvant être exprimées par les patrons est donnée par la figure 2.5 et tirée de [Dwyer et al., 1999].

Les patrons suivants sont proposés pour exprimer les conditions d'occurrence sur les exécutions du système :

- **Jamais** P : la propriété P ne doit être vraie pour aucun état (*absence*),
- **Toujours** P : la propriété P doit être vraie pour tous les états (*universalité*),
- **Éventuellement** P : la propriété P doit être vraie pour au moins un état (*existence*),
- **Éventuellement** P **au plus** n **fois** : la propriété P doit être vraie pour *au plus* $n \in \mathbb{N}$ états (*existence bornée*).

Les patrons exprimant une notion d'*ordre* d'apparition des propriétés dans une exécution sont les suivants :

- P **avant** P' : la propriété P doit être vraie durant l'exécution avant que P' soit vraie (*précédence*),
- P' **après** P : la propriété P' doit être vraie durant l'exécution après que P soit vraie (*réponse*).

Les patrons composés peuvent également utiliser des *combinaisons* ou des *séquences* de propriétés comme suit :

- $P_1, \ldots, P_n$ **avant** P' : toutes les propriétés $P_1, \ldots, P_n$ doivent être vraies, dans cet ordre, durant l'exécution avant que P' soit vraie,
- P **avant** $P'_1, \ldots, P'_n$: la propriété P doit être vraie avant que la séquence de propriétés $P'_1, \ldots, P'_n$ soit vraie,
- $P_1, \ldots, P_n$ **après** P' : toutes les propriétés $P_1, \ldots, P_n$ doivent être vraies, dans cet ordre, durant l'exécution après que P' soit vraie,
- P' **après** $P_1, \ldots, P_n$: la propriété P' doit être vraie après que la séquence de propriétés $P_1, \ldots, P_n$ soit vraie,
- les opérations booléennes classiques telles que la négation, la conjonction et la disjonction peuvent également être appliquées aux propriétés pour exprimer des conditions temporelles plus complexes.

Les propriétés P, P', $P_1, \ldots, P_n$ sont supposées être des formules booléennes portant sur les variables d'état d'un modèle. Dans le cadre de cette thèse, nous les utiliserons également comme des événements. Dans ce cas il convient de remplacer la notion de *vérité* d'une propriété P dans les définitions des patrons par la notion de *déclenchement* de l'événement P.

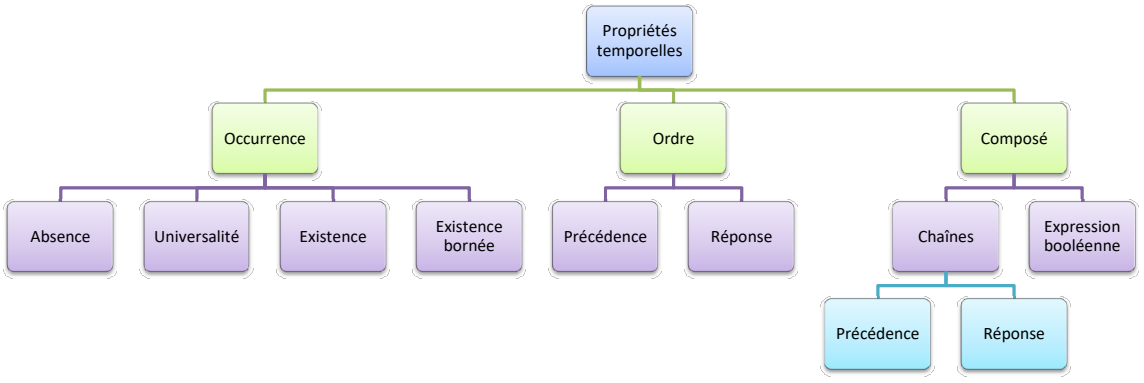


FIGURE 2.5 – Hiérarchie de la sémantique des propriétés temporelles

2.6.2/ PORTÉE DES PATRONS

Il est possible de restreindre la portion d'exécution considérée par le patron grâce à la notion de portée. Plus précisément, la portée d'un patron définit la partie des exécutions pour laquelle le patron doit être vérifié.

Soit un patron de spécification P. Les différentes portées possibles pour P, telles que proposées dans [Dwyer et al., 1999], sont illustrées par la figure 2.6 et sont décrites ci-dessous :

- **Globalement** P : P doit être vérifié sur toute l'exécution,
- P **avant** Q : P doit être vérifié avant que l'état (respectivement l'événement) Q ne soit atteint (respectivement déclenché),
- P **après** Q : P doit être vérifié après que l'état (respectivement l'événement) Q ait été atteint (respectivement déclenché),
- P **entre** Q **et** R : P doit être vérifié entre le moment où l'état (respectivement l'événement) Q est atteint (respectivement déclenché) et le moment où l'état (respectivement l'événement) R est atteint (respectivement déclenché),
- P **après** Q **jusqu'à** R : cette portée à la même signification que P **entre** Q **et** R à part que P doit être vérifié même si l'état (respectivement l'événement) R n'est pas atteint (respectivement déclenché).

On note que les portées des patrons peuvent être exprimées sur les états atteints lors d'une exécution ainsi que sur les événements déclenchés.

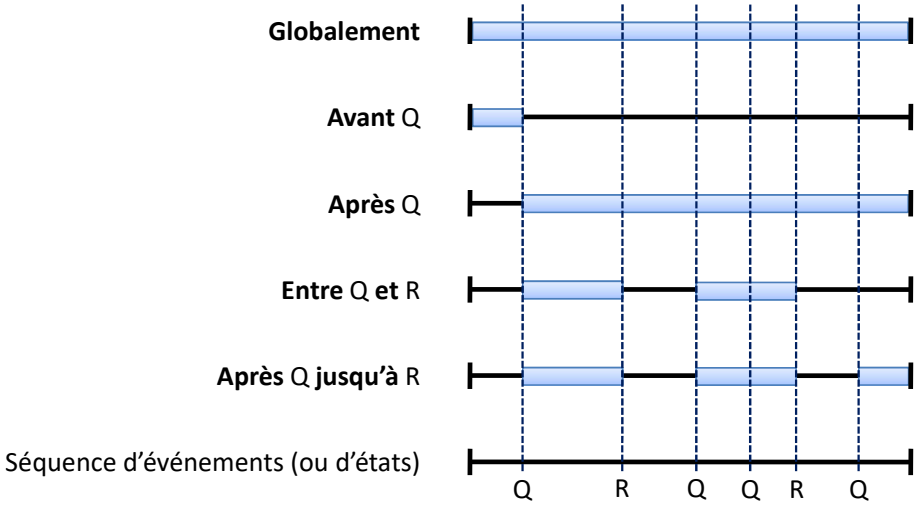


FIGURE 2.6 – Portées des propriétés temporelles

Exemple 4 : Expression d'une propriété temporelle sur l'exemple fil rouge

La propriété « aucun café ne peut être servi entre le moment où l'utilisateur demande à récupérer sa monnaie et en insère de nouveau dans la machine » peut s'exprimer par le patron de spécification P suivant :

$$P \stackrel{def}{=} \textbf{Jamais } \textit{serveCof} \textbf{ entre } \textit{askCof} \textbf{ et } (\textit{insert50} \textbf{ ou } \textit{insert100})$$

3

ETAT DE L'ART

3.1/ GÉNÉRATION DE TESTS À PARTIR DE MODÈLES

La génération de tests à partir de modèles (en anglais « Model Based Testing » souvent abrégé par « MBT ») est une méthode formelle permettant la vérification de logiciels et de systèmes critiques [El-Far et al., 2002, Blackburn et al., 1996, Legeard et al., 2004]. Cette méthode consiste à confronter les implémentations à un modèle. Le principe est d'extraire des tests d'un modèle et d'exécuter ces tests sur les implémentations afin de détecter les non conformités entre le modèle et ces implémentations. Il y a non conformité quand un test exécuté sur une implémentation produit des résultats non conformes à ceux prédits par le test issu du modèle. Cette méthode est aujourd'hui très utilisée dans l'industrie [Blackburn et al., 2001] et son intérêt a déjà été démontrée sur de gros projets du domaine ferroviaire, aéronautique et dans les télécommunications notamment [Lougee, 2001, Fantechi et al., 2012, Bernard et al., 2004].

3.1.1/ TEST DE SYSTÈMES

L'objectif du test de systèmes est de détecter des erreurs dans l'implémentation de ce système afin de pouvoir les corriger avant la mise en service de celle-ci. Plus le système est critique, c'est-à-dire plus les conséquences d'un défaut du système peuvent être graves (parfois jusqu'à mettre des vies humaines en jeu) [Knight, 2002], plus il est crucial de le tester afin d'avoir suffisamment confiance en sa qualité [Pressman, 2005] avant de l'utiliser.

Un test, ou cas de test, composé d'un couple données/résultat attendu, consiste à soumettre l'ensemble des données à l'implémentation du système sous test (en anglais « system under test » souvent abrégé par « SUT ») afin d'observer comment il réagit et de comparer le résultat obtenu avec les résultats attendus du système (on parle d'oracle). Si une différence entre le résultat obtenu et celui attendu est détectée, le test a permis de mettre en évidence une erreur pouvant se situer au niveau de :

- l'implémentation, auquel cas il convient de corriger l'erreur et d'appliquer à nouveau l'ensemble des tests à cette implémentation modifiée, et non pas seulement le test ayant mis en évidence l'erreur car la correction peut avoir engendré de nouvelles erreurs (voir tests de régression [Huizinga et al., 2007]),
- l'oracle, lorsque le résultat obtenu se révèle correct mais que le résultat attendu était erroné, auquel cas il convient de revoir ou préciser les résultats attendus pour

ce cas de test précis,
- l'implémentation et l'oracle, lorsque ni le résultat obtenu, ni l'oracle ne sont corrects.

Si le test d'un système peut mettre en évidence la présence d'erreurs, il ne peut cependant pas mettre en évidence leur absence. C'est-à-dire que si les tests ne révèlent aucune erreur, rien ne prouve qu'aucune erreur n'existe : les tests ne permettent simplement pas de les détecter.

Les oracles pour les différents cas de test d'une suite de tests (un ensemble de cas de test) sont issus de la description d'un modèle du système. Cette description est effectuée par l'ingénieur de test à partir des spécifications décrites conjointement par le client et l'ingénieur d'explicitation des besoins.

3.1.2/ SPÉCIFICATIONS DE SYSTÈMES

Les spécifications d'un système consistent généralement en un document énonçant, de manière aussi compréhensible que possible par le client et l'ingénieur d'explicitation des besoins, les fonctionnalités (on parle d'exigences fonctionnelles) et les comportements attendus du système. Bien qu'il existe de nombreux langages permettant de formaliser les spécifications (Z, B, Spec#, VDM, etc.) [Spivey, 1989, Abrial, 1996, Barnett et al., 2005, Fitzgerald et al., 2007, Dwyer et al., 1999], leur adoption par les non-initiés (souvent le client) est généralement compliquée. Pour cette raison, les spécifications sont généralement décrites de manière informelle.

Ce manque de formalisme est cependant source de problèmes car il en résulte des spécifications parfois ambiguës voire incomplètes. Si le résultat observé sur l'implémentation après un test est celui attendu par l'ingénieur, mais pas par le client (ou l'inverse), cela met en évidence une ambiguïté dans les spécifications. Des spécifications incomplètes correspondent par exemple au cas où le client souhaite que le système réponde d'une certaine manière dans certaines situations particulières (typiquement avec un ensemble d'entrées spécifiques), mais que certaines de ces situations ne sont pas spécifiées. Si les tests mettent en évidence cet oubli, c'est-à-dire lorsque les entrées soumises au système correspondent à un ensemble d'entrées spécifiques pour lesquelles le client s'attend (sans le spécifier) à une réponse particulière de la part du système, le problème sera détecté. Si en revanche les tests n'utilisent pas cet ensemble particulier d'entrées, la situation est plus grave car les spécifications seront considérées comme respectées alors que les exigences (les attentes) du client ne le seront pas. Dans les cas énoncés précédemment, lorsque le problème est détecté, il convient évidemment de reformuler, préciser ou compléter les points de spécification concernés par les tests afin de renforcer la confiance en la qualité du système vis-à-vis des exigences et en même temps renforcer les exigences.

Si toutes les spécifications ne sont pas testées, des problèmes dans les spécifications ou l'implémentation pourraient bien ne jamais être détectées. Idéalement, il serait donc souhaitable de concevoir un prototype du système reprenant chaque fonctionnalité demandée du système et s'accordant avec l'ensemble des spécifications et de s'appuyer sur ce prototype pour implémenter le système. En pratique en revanche, en raison notamment du nombre souvent élevé des spécifications, la conception d'un tel prototype serait trop coûteuse et contre-productive à mettre en œuvre. En effet, cela s'apparenterait fortement à implémenter directement le système et vérifier que l'implémentation

satisfait toutes les spécifications, ce qui n'est généralement pas réalisable. Pour pallier ce problème, on utilise généralement les spécifications informelles et on établit un modèle reprenant seulement une partie des comportements et des fonctionnalités décrites dans ces spécifications.

3.1.3/ MODÈLES DE TEST

Un modèle, dans le cadre de la génération de tests à partir de modèles, est une abstraction (une simplification) du système à tester : il ne modélise qu'une partie des spécifications et des fonctionnalités demandées par le client. Le modèle est également abstrait en ce sens qu'il ne modélise pas toutes les données telles qu'elles seront utilisées sur l'implémentation : elles sont simplifiées.

Par exemple, dans [Philipps et al., 2003], un système de carte à puce est étudié. Généralement, le terminal communiquant avec une carte à puce lui transmet les opérations et la carte répond par un code hexadécimal indiquant le résultat de la requête. Il peut s'agir d'un code exprimant le succès de l'opération, d'un code d'erreur de lecture des données, d'un code d'erreur d'écriture des données, d'un code d'erreur signalant que la commande émise par le terminal est inconnue, etc. Typiquement, le modèle ne s'intéressera pas forcément à tous les codes pouvant être renvoyés par la carte et distinguera par exemple plutôt le cas ou il s'agit d'un code d'erreur et le cas ou il s'agit d'un code de succès.

Cette abstraction des données implique donc que les entrées des tests qui seront générés à partir d'un modèle devront être adaptées pour pouvoir être appliquées au système. De même, le résultat attendu par un test et le résultat obtenu sur l'implémentation n'ont pas le même niveau d'abstraction et une couche d'adaptation est donc également nécessaire pour pouvoir comparer ces résultats.

Les comportements prévus du système ne seront généralement pas couverts par les tests si ils ne sont pas modélisés. Aussi, il convient de modéliser tous les comportements jugés critiques du système. Il n'existe pas, à priori, de méthode systématique pour établir quels comportements et fonctionnalités demandés dans les spécifications sont critiques. L'estimation du niveau de criticité est donc laissée à l'appréciation du client et de l'ingénieur. En pratique, il arrive que plusieurs modèles soient conçus, modélisant chacun une partie particulière des spécifications [Utting, 2005].

Un modèle doit évidemment être conforme aux spécifications qu'il modélise. Cela permet notamment de s'assurer, d'une part, que les tests qui seront issus du modèle pourront effectivement être applicables sur l'implémentation du système et, d'autre part, que les résultats attendus par les tests soient conformes aux spécifications. Cette validation du modèle est généralement réalisée à l'aide d'outils formels de vérification de propriétés tels que CUTE [Sen et al., 2006], SPIN [Holzmann, 2003] ou encore Java Pathfinder [Visser et al., 2005] traduisant les exigences (par exemple à l'aide de formules temporelles telles que la LTL [Pnueli, 1981] ou la CTL [Clarke et al., 1982, Emerson et al., 1985]). Afin de pouvoir mettre en œuvre ces techniques de *model-checking* [Clarke et al., 1999], le modèle doit bien sûr être plus simple et plus rapide à vérifier que l'implémentation du système elle-même.

3.1.4/ PROCESSUS DE GÉNÉRATION DE TESTS À PARTIR DE MODÈLES

Le processus de génération de tests à partir de modèles consiste généralement en 4 étapes principales [Utting, 2005, Utting et al., 2012]. La figure 3.1, constituant une version simplifiée du processus initalement présenté dans [Utting et al., 2012], illustre ces différentes étapes.

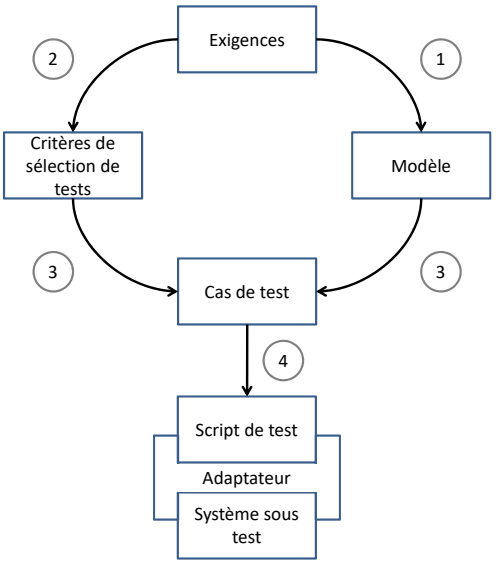


FIGURE 3.1 – Illustration simplifiée du processus de génération de tests à partir de modèles (voir [Utting et al., 2012])

ÉTAPE 1 : CONCEPTION DU MODÈLE DU SYSTÈME SOUS TEST

Un modèle abstrait du système est conçu à partir des exigences, habituellement exprimées sous la forme de spécifications informelles ou, plus rarement, de spécifications formelles (voir l'étape 1 sur la figure 3.1).

Le modèle décrit les comportements attendus du système sous test et modélise également son environnement : le contexte dans lequel le système sera utilisé et les contraintes qui en résultent. Un "*bon*" modèle de test ne modélise cependant qu'une partie des comportements attendus et de son environnement : il vise à tester une partie du système sous certaines contraintes, et non pas à tester le système complet dans son environnement réel d'utilisation. Sans cette abstraction du système et de son environnement, le modèle posséderait la même complexité que le système réel, ce qui annulerait l'intérêt de la modélisation.

ÉTAPE 2 : DÉFINITION DE CRITÈRES DE SÉLECTION DE TESTS

La définition de critères de sélection de tests (voir l'étape 2 sur la figure 3.1) permet de décrire les comportements fonctionnels ou structurels devant être couverts par les tests.

Les critères de sélection définissent donc ce qui doit être couvert par les tests, c'est-à-dire un comportement ou une propriété particulière du système, et comment cette couverture pourra être mesurée.

Dans le cadre de cette thèse, on s'intéresse particulièrement à la génération de tests réalisant la couverture de tous les états abstraits et de toutes les transitions abstraites : il s'agit d'un critère structurel de sélection des tests. Le système abstrait étant calculé à partir de prédicats d'abstraction constituant les gardes des événements apparaissant dans un objectif de test, les tests couvriront également partiellement cet objectif de test, donc un comportement fonctionnel exigé du système.

ÉTAPE 3 : GÉNÉRATION DES CAS DE TEST

La génération des cas de test est ensuite réalisée à l'aide du modèle et en fonction des critères de sélection de tests (voir l'étape 3 sur la figure 3.1). Typiquement, un algorithme ayant pour objectif de répondre aux critères de sélection de tests est appliqué au modèle. Il existe de nombreuses méthodes de génération de tests dans la littérature [Broy et al., 2005, Jaffuel et al., 2006], visant des critères de couvertures variés. Généralement, les cas de test issus d'un modèle sont représentés par des traces d'exécution indiquant les entrées à appliquer au système, les opérations à effectuer et les états dans lesquels le système doit se trouver à l'issue de ces opérations.

ÉTAPE 4 : EXÉCUTION DES TESTS SUR L'IMPLÉMENTATION

Une fois les cas de test générés, il convient de les exécuter sur le système afin de comparer les résultats attendus (tels que décrits par les tests) et les résultats effectivement obtenus (voir l'étape 4 sur la figure 3.1).

Puisque les tests sont issus d'un modèle abstrait du système, ils ne sont généralement pas directement exécutables sur ce dernier. Une phase d'adaptation des tests est donc souvent nécessaire : elle consiste à *traduire* les tests abstraits en tests concrets dont les entrées peuvent être appliquées au système implémenté et dont les sorties attendues sont directement comparables avec ceux obtenus sur le système.

Un test est dit *passant* lorsque le résultat attendu dans les tests est conforme à celui obtenu sur le système et *non-passant* lorsqu'il ne l'est pas. Un test passant ne met aucun problème en évidence mais permet de renforcer la confiance dans le système implémenté, à supposer que le test lui-même et donc le modèle dont il est issu correspondent bien aux spécifications. Un test non-passant met en évidence un problème au niveau de l'implémentation ou de la conception du modèle lorsque celui-ci ne décrit pas le système de manière suffisamment précise (d'où l'intérêt de valider le modèle vis-à-vis des spécifications).

3.2/ GÉNÉRATION DE TESTS À PARTIR D'ABSTRACTIONS

Malgré la première abstraction que le modèle constitue par rapport au SUT, ce modèle peut malgré tout définir un espace d'états de grande taille ou infini, rendant difficile voire impossible la génération automatique de tests pour certains critères de

couverture du modèle. On procédera alors à une abstraction du modèle lui-même, dans le but de maîtriser la taille de son espace d'états. Sauf mention contraire, le terme abstraction employé dans ce mémoire désigne cette phase d'abstraction du modèle lui-même. Afin d'éviter de générer des tests non applicables sur un système, la méthode généralement employée consiste à calculer une sous-approximation de ce modèle. La sous-approximation prend généralement la forme d'un système de transitions représentant de manière partielle la sémantique du modèle considéré.

La modélisation d'un système permet d'abstraire certains comportements afin de focaliser les tests sur des comportements précis ou des situations particulières. Cependant, l'espace d'états observable à partir d'un tel modèle, typiquement à l'aide d'un algorithme d'exploration, peut demeurer trop large et même être infini. Pour cette raison, de nombreuses techniques de génération d'abstractions d'un modèle (qui est lui-même une abstraction du système à tester) ont été mises au point [Grieskamp et al., 2002, Clarke et al., 2003, Veanes et al., 2003, Ball et al., 2005, Pasareanu et al., 2007]. Généralement, ces techniques permettent de calculer un système de transitions de taille réduite et finie, au prix d'une perte d'informations sur les situations (les états) pouvant être atteints par le système sous test ainsi que sur les opérations (les transitions) effectivement applicables sur ce dernier.

Dans le cadre de la vérification de propriétés sur un modèle (« model checking »), cette perte d'information ne représente pas forcément une barrière car l'abstraction peut malgré tout préserver ces propriétés [Henzinger et al., 2002, Clarke et al., 2003, Chaki et al., 2003, Ball et al., 2005]. En effet, en model-checking on cherche en général à prouver qu'aucune exécution du modèle ne viole une propriété donnée. Ainsi, si aucune exécution *may*-atteignable ne viole une propriété, alors a fortiori aucune exécution concrètement atteignable ne le fait. Dans ce cas, l'étude de l'abstraction (généralement de petite taille) est donc avantageuse car elle permet de vérifier la propriété plus simplement et plus rapidement qu'en étudiant l'espace d'états observables à partir du modèle. Si l'abstraction ne respecte pas une propriété, un contre-exemple produit à partir de cette abstraction exhibant une trace violant la propriété pourrait être un faux positif, c'est-à-dire que le contre-exemple pourrait ne pas être applicable au système à tester. Dans ce cas, il est d'usage de raffiner l'abstraction jusqu'à obtenir la garantie que les propriétés soient préservées, par exemple à l'aide de techniques de raffinement à partir de contre-exemples (en anglais « counter-example guided abstraction refinement », souvent abrégé par « CEGAR ») [Clarke et al., 2003].

Dans le cadre de la génération de tests à partir de modèles, il est également désirable d'obtenir des tests effectivement applicables au système. Lorsque l'exploration complète des situations et des comportements possibles à partir du modèle n'est pas envisageable, l'abstraction du modèle peut malgré tout être envisagée. Naturellement, puisque le système de transitions obtenu par abstraction constitue une sur-approximation du modèle et donc du système à tester, l'algorithme de génération de tests doit s'assurer d'en extraire la partie pouvant être instanciée sur ce dernier. Un tel algorithme de génération de tests à partir d'une abstraction de modèle calcule donc habituellement une sous-approximation de la sémantique décrite par ce modèle, à partir d'un système de transitions abstrait constituant une sur-approximation de ce dernier.

3.2.1/ SOUS-APPROXIMATION DE PROGRAMMES UTILISANT LES MODALITÉS DES TRANSITIONS

Dans [Ball, 2005], Thomas Ball propose une méthode de génération de tests (appelée « PCT » pour « Predicate-Complete Testing ») à partir d'une abstraction de programmes. Un programme est classiquement composé d'un compteur de programme, d'un ensemble d'instructions d'affectation de variables, de structures conditionnelles et de boucles (voir la logique de Hoare [Hoare, 1969]).

Les conditions apparaissant dans les structures conditionnelles et les boucles constituent l'ensemble de prédicats d'abstraction utilisé pour calculer l'abstraction du programme. Les tests générés avec la méthode PCT visent à couvrir tous les états atteignables d'un programme (au plus $m \times 2^n$ avec m le nombre d'instructions dans le programme et n le nombre de prédicats d'abstraction). Un tel critère de couverture permet de considérer les dépendances entre les conditions d'un programme et les instructions dont elles contrôlent l'exécution.

La figure 3.2 montre un exemple de programme avec :

- une instruction `skip` ne modifiant pas l'état du programme au niveau du compteur de programme L2,
- des instructions d'affectation au niveau du compteur de programme L0, L3, L4 et L6,
- des instructions conditionnelles au niveau du compteur de programme L1 et L5.

On note que la variable x n'est pas initialisée et on considère qu'elle prend une valeur aléatoire parmi l'ensemble des entiers relatifs $\mathbb{Z}$.

FIGURE 3.2 – Un exemple de programme

```
// Code              | // Etats atteignables avant exécution de l'instruction
                     |
L0 : y := 0;         | (L0, x < 0), (L0, ¬(x < 0)) // x non initialisée
L1 : if (x < 0) {    | (L1, x < 0), (L1, ¬(x < 0)) // x non initialisée
L2 :   skip;         | (L2, x < 0) // condition x < 0 en L1
     } else {        |
L3 :   x := -2;      |              (L3, ¬(x < 0)) // négation de x < 0 en L1
     }               |
L4 : x := x + 1;     | (L4, x < 0) // x < 0 par L2 ou x = -2 par L3
L5 : if (x < 0) {    | (L5, x < 0), (L5, ¬(x < 0)) // x ≤ 0 ou x = -1 par L4
L6 :   y := 1;       | (L6, x < 0) // condition x < 0 en L5
     }               |
```

Les instructions conditionnelles (L1 et L5) utilisent la même condition $x < 0$ qui constitue donc l'unique prédicat d'abstraction pour ce programme. Le compteur de programme n'étant jamais abstrait, les états abstraits manipulés sont une paire constituée du compteur de programme et de la formule correspondante. Les transitions abstraites ne sont pas nommées car les programmes classiques ne possèdent pas d'événements.

A partir du programme, on peut calculer le 3MTS $\langle Q_0, Q, \Delta, \Delta^-, \Delta^+ \rangle$ correspondant (voir [Godefroid et al., 2001] pour un algorithme de calcul du 3MTS). Le 3MTS calculé pour le programme de la figure 3.2 est le suivant :

- $Q_0 \stackrel{def}{=} \{(L0, x < 0), (L0, \neg(x < 0))\}$,
- $Q \stackrel{def}{=} \{(L0, x < 0), (L0, \neg(x < 0)), (L1, x < 0), (L1, \neg(x < 0)), (L2, x < 0), (L3, \neg(x < 0)), (L4, x < 0), (L5, x <$

$0), (L5, \neg(x < 0)), (L6, x < 0)\}$,

- $\Delta \stackrel{def}{=} \{(L0, x < 0) \xrightarrow{must^{\#}} (L1, x < 0), (L0, \neg(x < 0)) \xrightarrow{must^{\#}} (L1, \neg(x < 0)), (L1, x < 0) \xrightarrow{must^{\#}} L2, x < 0, (L1, \neg(x < 0)) \xrightarrow{must^{\#}} L3, \neg(x < 0), (L2, x < 0) \xrightarrow{must^{\#}} (L4, x < 0), (L3, \neg(x < 0)) \xrightarrow{must^{+}} (L4, x < 0), (L4, x < 0) \xrightarrow{may} (L5, x < 0), (L4, x < 0) \xrightarrow{may} (L5, \neg(x < 0)), (L5, x < 0) \xrightarrow{must^{\#}} (L6, x < 0)\}$,
- $\Delta^{-} \stackrel{def}{=} \{(L0, x < 0) \xrightarrow{must^{\#}} (L1, x < 0), (L0, \neg(x < 0)) \xrightarrow{must^{\#}} (L1, \neg(x < 0)), (L1, x < 0) \xrightarrow{must^{\#}} L2, x < 0, (L1, \neg(x < 0)) \xrightarrow{must^{\#}} L3, \neg(x < 0), (L2, x < 0) \xrightarrow{must^{\#}} (L4, x < 0), (L4, x < 0) \xrightarrow{may} (L5, \neg(x < 0)), (L5, x < 0) \xrightarrow{must^{\#}} (L6, x < 0)\}$,
- $\Delta^{+} \stackrel{def}{=} \{(L0, x < 0) \xrightarrow{must^{\#}} (L1, x < 0), (L0, \neg(x < 0)) \xrightarrow{must^{\#}} (L1, \neg(x < 0)), (L1, x < 0) \xrightarrow{must^{\#}} L2, x < 0, (L1, \neg(x < 0)) \xrightarrow{must^{\#}} L3, \neg(x < 0), (L2, x < 0) \xrightarrow{must^{\#}} (L4, x < 0), (L3, \neg(x < 0)) \xrightarrow{must^{+}} (L4, x < 0), (L5, x < 0) \xrightarrow{must^{\#}} (L6, x < 0)\}$.

Si la variable x vaut initialement -1, les instructions exécutées sont L0, L1, L2, L4, L5. En effet, x vaut -1 après L0. La condition en L1 $(x < 0)$ est donc vérifiée et L2 est donc exécutée mais n'affecte aucune variable. L'instruction en L4 incrémente ensuite x qui vaut donc 0. La condition en L5 n'est pas vérifiée et L6 n'est donc pas exécutée.

Tous les états abstraits présents dans Q sont des états may-atteignables (atteignables par une séquence de may-transitions à partir d'un état abstrait source). Tout état abstrait n'appartenant pas à Q (par exemple $(L6, \neg(x < 0))$ pour le programme de la figure 3.2) ne peut donc être couvert par des tests. Le calcul du 3MTS constitue donc une sur-approximation possiblement non-déterministe à partir d'un programme déterministe (à partir de $(L4, x < 0)$ par exemple, il existe une transition vers $(L5, x < 0)$ et $(L5, \neg(x < 0))$). Cette sur-approximation permet d'obtenir une borne supérieure au nombre d'états atteignables et constitue donc une meilleure approximation que la première limite $m \times 2^n$.

Une fois le 3MTS calculé, il convient de calculer une sous-approximation du programme, c'est-à-dire un sous-ensemble des états effectivement atteignables par une exécution du programme. Pour cela, Thomas Ball utilise la connaissance des modalités des transitions pour calculer des états dont l'atteignabilité est garantie. Il démontre dans [Ball, 2005] que des séquences de transitions composées d'une séquence de longueur quelconque de $must^{-}$-transitions suivie d'au plus une may-transition suivie d'une séquence de longueur quelconque de $must^{+}$-transitions peuvent nécessairement être instanciées et être atteintes dans le programme. La seule condition est que ces séquences soient atteignables depuis un état initial. Une telle séquence sera appelée « chaîne de Ball » dans la suite de cette thèse. Tous les états et transitions d'une séquence de transitions instanciant une chaîne de Ball et dont l'état source est un état initial sont donc atteignables sur le programme.

A partir du 3MTS du programme de la figure 3.2, on peut ainsi identifier de nombreuses chaînes de Ball à partir des états initiaux. Les plus longues d'entre-elles sont les suivantes (on rappelle que les transitions dont la modalité est $must^{\#}$ possèdent la modalité $must^{-}$ et la modalité $must^{+}$:

- $(L0, x < 0) \xrightarrow{must^{\#}} (L1, x < 0), (L1, x < 0) \xrightarrow{must^{\#}} (L2, x < 0), (L2, x < 0) \xrightarrow{must^{\#}} (L4, x < 0), (L4, x < 0) \xrightarrow{must^{-}} (L5, x < 0), (L5, x < 0) \xrightarrow{must^{\#}} (L6, x < 0)$
- $(L0, x < 0) \xrightarrow{must^{\#}} (L1, x < 0), (L1, x < 0) \xrightarrow{must^{\#}} (L2, x < 0), (L2, x < 0) \xrightarrow{must^{\#}} (L4, x < 0), (L4, x < 0) \xrightarrow{may} (L5, \neg(x < 0))$
- $(L0, \neg(x < 0)) \xrightarrow{must^{\#}} (L1, \neg(x < 0)), (L1, \neg(x < 0)) \xrightarrow{must^{\#}} (L3, \neg(x < 0)), (L3, x < 0) \xrightarrow{must^{+}} (L4, x < 0)$

Dans cet exemple, tous les états du 3MTS sont atteignables par au moins une chaîne de Ball dont l'état abstrait source est initial. Comme cela sera montré par la suite, tous

les états du 3MTS ne sont cependant pas toujours atteints par une chaîne de Ball (voir l'algorithme BCI du chapitre 5 page 77 par exemple). Cela ne signifie cependant pas que ces états ne sont pas atteignables par le programme : leur atteignabilité est inconnue. Ainsi, le calcul de toutes les chaînes de Ball ne fournit qu'une sous-approximation des états atteignables.

Le principe de la méthode PCT consiste alors à réduire la différence entre le nombre d'états abstraits may-atteignables ($|Q|$) et le nombre d'états abstraits atteints par des chaînes de Ball. Lorsque ces deux valeurs sont égales, la sous-approximation contient l'ensemble des états atteignables par le programme. Pour réduire cet écart, on introduit généralement de nouveaux prédicats à l'ensemble de prédicats d'abstraction comme dans [Pasareanu et al., 2007] par exemples (article discuté plus loin).

L'objectif final étant de générer des tests à partir de cette sous-approximation, il convient de calculer des instances concrètes de chaînes de Ball. En principe, plusieurs chaînes de Ball peuvent couvrir les mêmes états abstraits. Par exemple, les deux premières chaînes de Ball énumérées précédemment couvrent exactement les mêmes états abstraits jusqu'à $(L4, x < 0)$. A partir de là, la première couvre $(L5, x < 0)$ et $(L6, x < 0)$, tandis que la seconde couvre $(L5, \neg(x < 0))$. Aucune de ces deux chaînes ne permet de couvrir par exemple les états abstraits $(L0, \neg(x < 0))$, $(L1, \neg(x < 0))$ et $(L2, \neg(x < 0))$. La troisième chaîne énumérée, couvre cependant ces états. Ainsi, on pourra sélectionner ces trois chaînes de Ball pour couvrir tous les états abstraits du programme.

En trouvant (à l'aide d'un solveur SMT par exemple) des états instanciant les chaînes de Ball sélectionnées, on obtient des séquences de transitions concrètes pouvant servir de tests permettant de couvrir tous les états de la sous-approximation calculée. On qualifie ce procédé d'instanciation de chaînes de transitions abstraites d'« exécution symbolique » (voir la section 3.3 pour une explication plus précise).

Typiquement, dans le cas d'un système événementiel, à la différence des programmes considérés par Thomas Ball, le compteur de programme n'existe pas et les événements sont déclenchés de manière non-déterministe lorsque leur garde est satisfaite. Ainsi, une transition may-atteignable peut ne posséder aucune instance susceptible d'être atteinte depuis un état concret initial et le 3MTS constitue donc clairement une sur-approximation des états et transitions effectivement atteignables.

3.2.2/ CALCUL D'UNE SOUS-APPROXIMATION D'UN SYSTÈME POUR LA VÉRIFICATION DE PROPRIÉTÉS : ALGORITHMES αSearch ET RefinementSearch

Dans [Pasareanu et al., 2007], l'objectif est de vérifier une propriété sur un système en raffinant automatiquement son abstraction (par prédicats) tant qu'un contre-exemple violant la propriété n'est pas trouvé et que l'abstraction peut entraîner une perte de précision vis-à-vis de cette propriété.

Les algorithmes proposés dans cet article sont appliqués à des systèmes déterministes et composés d'un unique état concret initial. Afin de simplifier les explications, on considère donc par la suite qu'ils sont appliqués à des systèmes événementiels particuliers dont les opérations sont déterministes et qui possèdent un unique état concret initial.

ALGORITHME D'EXPLORATION VÉRIFIANT LA PRÉSENCE D'UNE PERTE DE PRÉCISION

L'algorithme αSearch présenté dans [Pasareanu et al., 2007] prend en paramètres un système événementiel déterministe et un ensemble de prédicats d'abstraction. Il réalise une exploration partielle de l'espace d'états concrets du système et enregistre les états abstraits auxquels ils appartiennent, ainsi que les transitions abstraites concrétisées lors de l'exploration. La structure ainsi calculée par l'algorithme correspond ainsi globalement à un ATS $\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa \rangle$ (voir définition 19 page 33) dans lequel la fonction κ n'est pas utilisée.

La phase d'exploration commence par calculer et enregistrer dans Q_0 et Q l'état abstrait auquel correspond l'état initial c_0 du système ($\alpha(c_0)$ où α est la fonction d'abstraction). L'algorithme teste ensuite, pour chaque événement e, s'il peut être appliqué à partir de c_0. Si c'est le cas, l'état abstrait correspondant à l'état concret cible c' obtenu (donc l'état $\alpha(c')$) est enregistré dans Q. De même, la *may*-transition $\alpha(c_0) \xrightarrow{e} \alpha(c')$ est enregistrée dans Δ. Les états concrets et les transitions concrètes calculés sont également ajoutés à leurs ensembles respectifs C (et C_0 pour l'état c_0) et Δ^c.

Si l'état abstrait correspondant à l'état c' n'existait pas dans Q, donc si un nouvel état abstrait est découvert, l'algorithme d'exploration recommence à partir de c' (et non pas c_0 cette fois) après que tous les événements possibles ont été appliqués à c_0. Si plusieurs états concrets correspondant à de nouveaux états abstraits sont obtenus par application des événements, tous ces états concrets serviront de source à l'exploration.

Afin de raffiner l'abstraction par la suite (voir l'algorithme RefinementSearch plus loin), un nouvel ensemble de prédicats d'abstraction (contenant initialement les prédicats d'abstraction fournis en entrée de l'algorithme αSearch) est créé et complété lors de l'exploration. Chaque fois que l'algorithme détermine si un événement est applicable à un état concret, il teste également si l'abstraction risque d'entraîner une perte de précision. Cette situation apparaît typiquement lorsque la transition abstraite correspondant à une transition concrète n'a pas la modalité $must^+$ [Ball, 2005, Dams et al., 2004, de Alfaro et al., 2004, Shoham et al., 2007]. Dans ce cas, la garde de l'événement appliqué par la transition est ajoutée au nouvel ensemble de prédicats d'abstraction. Intuitivement, l'ajout des gardes des événements de ces transitions à ce nouvel ensemble de prédicats d'abstraction permettra, lors de la phase de raffinement de l'abstraction, de faire apparaître la modalité $must^+$ recherchée.

Finalement, l'algorithme αSearch termine si aucun nouvel état abstrait n'est découvert lors de l'exploration. Il renvoie alors une paire composée du MTS calculé et du nouvel ensemble de prédicats d'abstraction complété.

On note que les *may*-transitions rencontrées lors de l'exploration sont ainsi toujours concrétisées par des transitions atteintes : l'algorithme calcule une sous-approximation du système. L'algorithme se révèle donc intéressant dans le cadre de la génération de tests car il garantit que les traces constituant les tests seront instanciables sur le système. Cependant, le caractère non déterministe des événements d'un système événement implique que certaines *may*-transitions peuvent ne jamais être concrétisées, typiquement si leur état abstrait source n'est jamais atteint lors de l'exploration. Ainsi, la couverture par les tests de l'ensemble des états abstraits et des transitions abstraites pour un système non déterministe n'est pas garantie avec l'algorithme αSearch.

ALGORITHME DE RAFFINEMENT D'UNE ABSTRACTION VÉRIFIANT UNE PROPRIÉTÉ

L'algorithme RefinementSearch présenté dans [Pasareanu et al., 2007] prend en paramètres un système événementiel déterministe et une formule booléenne ϕ représentant un état d'erreur (exprimé par combinaison des prédicats apparaissant dans un ensemble de prédicats d'abstraction $\mathcal{P}$). Il a pour objectif de déterminer si l'état d'erreur ϕ peut être atteint sur le système.

L'algorithme fait d'abord appel à αSearch avec le système événementiel et $\mathcal{P}$ comme paramètres. Une paire composée de l'ATS calculé et du nouvel ensemble de prédicats d'abstraction $\mathcal{P}'$ est ainsi obtenue.

Puisque le CTS de l'ATS constitue une sous-approximation du système, alors toute séquence de transitions issue du CTS permettant d'atteindre ϕ constitue une preuve que cet état d'erreur et atteignable sur le système. Si le nouvel ensemble de prédicats d'abstraction est identique au précédent ($\mathcal{P}' = \mathcal{P}$), cela signifie que l'abstraction constitue une bisimulation [Lee et al., 1992] du système événementiel, et donc qu'aucune imprécision n'est générée par l'abstraction. Par conséquent, si $\mathcal{P}' = \mathcal{P}$ et si l'état ϕ ne peut pas être atteint sur le CTS calculé avec αSearch, il ne peut pas l'être sur le système événementiel.

Si les deux ensembles de prédicats d'abstraction sont différents et que ϕ n'est pas atteint sur le CTS en revanche, l'algorithme ne peut pas conclure immédiatement sur l'atteignabilité de ϕ sur le système. Il fait donc de nouveau appel à αSearch avec cette fois-ci le nouvel ensemble de prédicats d'abstraction $\mathcal{P}'$ en paramètres afin de raffiner l'abstraction et donc l'ATS obtenus. L'algorithme RefinementSearch continue tant qu'il ne peut pas conclure, c'est-à-dire tant qu'aucune séquence de transitions permettant d'atteindre ϕ n'est trouvée et que le nouvel ensemble de prédicats obtenus par application de αSearch est différent du précédent.

La terminaison de l'algorithme n'est pas garantie. En effet, il est nécessaire qu'il parvienne à une bisimulation pour s'arrêter. Cependant, il est aussi possible que l'algorithme parvienne à une bisimulation du système et ne puisse pas le savoir : le fait que les deux ensembles de prédicats d'abstraction soient différents n'implique pas forcément que l'abstraction entraîne une imprécision. De ce fait, l'algorithme RefinementSearch peut ne pas terminer, même s'il est garanti d'obtenir une bisimulation.

3.2.3/ GÉNÉRATION D'UN SYSTÈME FINI D'ÉTATS ET DE TRANSITIONS : ALGORITHME GenFSM

Dans [Grieskamp et al., 2002], les auteurs proposent un algorithme permettant de générer un système fini d'états et de transitions (en anglais « Finite State Machine ») à partir d'un modèle dont l'espace d'état est très grand ou infini. Les modèles utilisés sont décrits en langage AsmL (pour « Abstract State Machine Language ») mais l'algorithme de calcul du système fini peut en pratique tout à fait s'appliquer aux systèmes réactifs (notamment les systèmes événementiels). L'objectif final de l'algorithme est de pouvoir utiliser les outils de génération de tests déjà existants qui s'appliquent généralement à des sous-approximations exprimées sous la forme de systèmes finis.

L'exploration réalisée par l'algorithme proposé (GenFSM) est similaire à celle de l'algorithme αSearch de [Pasareanu et al., 2007] : elle calcule un ATS $\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa\rangle$ (sans la fonction de coloration κ) dont le CTS $\langle C_0, C, \Delta^c\rangle$ constitue une sous-approximation

du modèle. Les différences principales avec αSearch résident dans les faits suivants :

- GenFSM peut être appliqué à des systèmes non-déterministes,
- l'exploration s'arrête lorsqu'aucune transition concrète calculée n'est jugée pertinente (voir l'explication plus bas) plutôt que lorsqu'aucun nouvel état abstrait n'est découvert,
- GenFSM ne prépare pas d'étape de raffinement puisque l'objectif n'est pas de vérifier une propriété sur le modèle mais de générer des tests.

L'algorithme utilise une fonction d'abstraction, issue d'un ensemble de prédicats d'abstraction choisis arbitrairement, permettant de calculer l'état abstrait q correspondant à un état concret c. On considère par la suite que cette fonction est la fonction d'abstraction α de l'ATS et qu'un appel à $\alpha(c)$ permet de calculer l'état abstrait q auquel appartient c.

EXPLORATION D'UNE SOUS-APPROXIMATION DU SYSTÈME

L'algorithme tient à jour un ensemble d'états concrets à explorer appelé « frontière ». Initialement, la frontière contient l'ensemble des états concrets initiaux du système (on considère que cet ensemble est fini). Ces états concrets initiaux sont évidemment ajoutés aux ensembles C et C_0 de l'ATS calculé par l'algorithme.

Un état c est sélectionné parmi ceux contenus dans la frontière puis retiré de cette dernière. Tous les événements du système pouvant être appliqués à l'état c lui sont appliqués. Les états cibles c' obtenus par application de ces événements sont stockés dans un ensemble appelé « nextStates ». Dans le cas où un événement e est non-déterministe, l'ensemble (qu'on considère comme fini) des états concrets pouvant être atteints par application de e à l'état c sont ajoutés à nextStates.

Pour chaque état c' atteint (chaque état c' de nextStates), c et c' sont ajoutés à C, la transition concrète $c \xrightarrow{e} c'$ est ajoutée à l'ensemble des transitions concrètes Δ^c de l'ATS et la transition abstraite $\alpha(c) \xrightarrow{e} \alpha(c')$ qui lui correspond est ajoutée à Δ. Naturellement, l'ensemble d'états abstraits Q est complété par les états abstraits $\alpha(c)$ et $\alpha(c')$, de même que Q_0 dans le cas où c ou c' est un état initial. L'algorithme GenFSM garantit donc que toutes les transitions abstraites sont concrétisées par des transitions concrètes atteintes dans la sous-approximation calculée (c'est-à-dire dans le CTS).

Finalement, les états concrets c' atteints (ceux contenus dans nextStates) sont ajoutés à la frontière si la transition $c \xrightarrow{e} c'$ est considérée comme pertinente par rapport à un objectif particulier (voir l'explication ci-dessous). Sans cette condition de pertinence des transitions, l'algorithme ne terminerait pas dans le cas où le système contient une infinité d'états et calculerait l'espace d'état complet sinon. En effet, l'algorithme GenFSM recommence l'exploration à partir des états présents dans la frontière. Il ne s'arrête donc que lorsque celle-ci devient vide, ce qui implique de limiter les ajouts faits à la frontière.

ÉVALUATION DE LA PERTINENCE DES TRANSITIONS VIS-À-VIS D'UN OBJECTIF PARTICULIER

Afin de garantir la terminaison de l'algorithme GenFSM, ou en tout cas de réduire l'espace d'états exploré, les auteurs de [Grieskamp et al., 2002] proposent d'évaluer la pertinence des transitions instanciées par rapport à un objectif particulier. Un état concret atteint par

une transition instanciée lors de l'exploration n'est alors ajouté à la frontière que si cette transition est effectivement pertinente.

Dans [Grieskamp et al., 2002], il est proposé de considérer une transition comme pertinente uniquement si l'état abstrait atteint par cette transition n'avait pas déjà été découvert. Une telle évaluation de la pertinence d'une transition permet de garantir la terminaison de l'algorithme, puisque l'ensemble des états abstraits pouvant être explorés est fini. Elle tend également à guider l'exploration pour minimiser la taille de la sous-approximation tout en couvrant autant d'états abstraits et de transitions abstraites que possible.

Une autre méthode proposée consiste à considérer une transition comme pertinente si l'état concret atteint par application de la transition n'a pas encore été rencontré et si elle fait évoluer le système vers une situation d'intérêt pour le testeur. Cette méthode ne garantit cependant la terminaison que dans le cas où l'espace d'états est fini ou si la fonction évaluant la pertinence des transitions est garantie de renvoyer un résultat négatif (indiquant qu'une transition n'est pas pertinente) lors de l'exploration.

La solution finalement proposée consiste ainsi à définir un objectif particulier à atteindre. Dans le cas de l'exemple de la machine à café, on pourrait par exemple souhaiter observer le système lorsqu'il ne reste plus de café disponible. Une évaluation de la pertinence d'une transition pourrait alors consister à vérifier si elle mène dans un nouvel état abstrait, ou si elle permet de diminuer le nombre de cafés restants dans la machine.

Cet outil d'évaluation de pertinence permet donc de guider l'exploration afin d'obtenir une sous-approximation qui permette d'observer le système dans une (ou plusieurs) situation(s) précise(s) tout en augmentant la couverture des états abstraits et des transitions abstraites.

3.2.4/ GÉNÉRATION D'UNE SOUS-APPROXIMATION D'UN SYSTÈME AVEC ESTIMATION DE LA QUALITÉ DE CETTE DERNIÈRE

La méthode de génération d'une sous-approximation d'un système proposée dans [Veanes et al., 2003] reprend l'exploration effectuée par l'algorithme GenFSM de [Grieskamp et al., 2002]. La principale différence dans l'approche présentée dans [Veanes et al., 2003] (qu'on appellera GenColoredFSM) réside dans la connaissance obtenue de l'atteignabilité des transitions abstraites du MTS calculé.

Avec l'algorithme GenFSM on calcule un sous-ensemble des états abstraits et des transitions abstraites atteignables, c'est-à-dire pouvant être instanciées par une transition concrète qui peut être atteinte depuis un état initial du système. Seuls les états abstraits et les transitions abstraites atteints lors de l'exploration sont connus. Avec l'algorithme GenColoredFSM, le MTS complet est d'abord calculé : tous les états abstraits et toutes les transitions entre ces états abstraits sont enregistrés dans Q (et Q_0 si l'état abstrait contient un état concret initial) et Δ. On ne se soucie pas, pour l'instant, de savoir si les états abstraits et les transitions abstraites peuvent être concrétisés par des instances atteignables depuis un état initial.

C'est ici qu'intervient la fonction de coloration κ de l'ATS. Cette fonction attribue, d'après la définition 19, une couleur à chaque état abstrait du MTS. Dans le cas de GenColoredFSM, elle attribue plutôt une couleur à chaque transition abstraite du MTS en fonction de son atteignabilité sur le système réel, vérifiée à l'aide d'un solveur SMT. L'attribution d'une

couleur à une transition abstraite $t \stackrel{def}{=} q \xrightarrow{e} q'$ s'effectue alors selon les règles suivantes :

- t est *verte* ($\kappa(t) = vert$) si une instance de t atteignable par une séquence de transitions depuis un état initial existe,
- t est *rouge* ($\kappa(t) = rouge$) si aucune instance de t n'existe (typiquement si aucun état concret de q ne peut mener à un état concret de q' par e),
- t est *bleue* ($\kappa(t) = bleu$) si une instance de t existe mais que l'atteignabilité de cette instance depuis l'état initial n'est pas garantie,
- t est *grise* ($\kappa(t) = gris$) si elle n'est ni *rouge*, ni *verte*, ni *bleue* (autrement dit si sa couleur n'a pas encore été calculée ou si le solveur SMT n'est pas parvenu à conclure).

Initialement, toutes les transitions abstraites du MTS sont donc coloriées en $gris$. L'algorithme GenColoredFSM procède alors à l'exploration des états et des transitions concrètes par le même procédé que décrit par GenFSM. Toute transition atteinte durant l'exploration de GenFSM garantit que la transition abstraite qu'elle instancie est atteignable sur le système, la transition abstraite correspondante est ainsi coloriée en $vert$. Pour toute transition abstraite t n'ayant pas été coloriées en $vert$ à l'issue de cette exploration, GenColoredFSM vérifie s'il existe une instance $c \xrightarrow{e} c'$ de t. Si c'est le cas, t est coloriée en $vert$ si c est déjà connu comme étant la source ou la cible d'une instance de transition $verte$, en $bleu$ sinon. Si ce n'est pas le cas, t est coloriée en $rouge$ puisque aucune instance de t n'existe. Si le solveur SMT n'est pas parvenu à conclure sur l'existence d'une instance de t (autrement dit si le solveur renvoie $unknown$), la transition t reste $grise$.

L'algorithme GenColoredFSM permet donc d'obtenir une sous-approximation du système similaire à celle obtenue avec GenFSM, avec en plus une information d'atteignabilité sur les transitions abstraites du MTS. On sait en effet, pour toutes les transitions abstraites pour lequel le solveur SMT est parvenu à conclure, si cette transition abstraite peut être couverte par des tests (si elle est $verte$), et si elle ne peut pas être couverte par des tests instanciables (si elle est $rouge$). Si la transition est $bleue$ et qu'un algorithme permet d'atteindre, à partir d'un état initial du système, l'instance ayant permis de déterminer sa couleur, alors elle peut également être couverte par des tests. Si un algorithme permet de démontrer que cette instance ne peut être atteinte depuis un état initial, on ne peut en revanche pas conclure que la transition abstraite ne puisse pas être couverte par des tests : il peut exister d'autres instances de cette transition qui soient atteignables.

3.3/ EXPLORATION SYMBOLIQUE DYNAMIQUE

L'exploration symbolique [Păsăreanu et al., 2009, Baldoni et al., 2018] est une technique permettant de déterminer des valeurs pour les variables d'entrée d'un programme afin d'exécuter une portion spécifique de ce programme.

Afin d'illustrer les principes évoqués dans cette section, on propose d'étudier le programme de la figure 3.3. La fonction « `function` » prend en paramètre la variable x qui peut prendre n'importe quelle valeur dans l'ensemble des entiers relatifs $\mathbb{Z}$. L'objectif de l'exploration symbolique est alors de déterminer une valeur de x permettant d'amener le programme à exécuter la ligne L4 qui correspond à une situation d'erreur que l'on souhaite couvrir par des tests.

On note que si x est inférieure ou égale à 0, la variable y sera toujours inférieure ou égale

FIGURE 3.3 – Un exemple de programme. L4 n'est exécutée que si la variable x est égale à 1.

```
L0 : function(x) {
L1 :   y := 2 * x;
L2 :   if (y > 0) {
L3 :     if (x - 2 < 0) {
L4 :       // ERROR
         }
       }
     }
```

à 0 après exécution de l'instruction $y := 2 * x$ en L1. Par conséquent, la condition $y > 0$ de la ligne L2 sera toujours fausse et le programme ne passera ni par la ligne L3 ni par la ligne L4. Si x vaut 1, la variable y sera égale à 2 après exécution de l'instruction en L1. Les conditions en L2 et L3 sont toutes deux vraies car $2 > 0$ et $1 - 2 < 0$. L'instruction en L4 sera donc exécutée, ce qui est l'objectif. Si x est supérieure à 1 (donc supérieure ou égale à 2), la variable y sera supérieure ou égale à 4 après exécution de l'instruction L1. La condition de la ligne L2 sera vraie car $y \geq 4 \implies y > 0$. La condition de la ligne L3 sera en revanche fausse car $x \geq 2 \implies x - 2 \geq 0$, autrement dit $x \geq 2 \implies \neg(x - 2 < 0)$. L'exécution symbolique de la fonction `function` afin d'amener le programme à exécuter la ligne L4 doit donc permettre de déterminer la valeur 1 pour la variable x.

3.3.1/ EXPLORATION SYMBOLIQUE

Cette section présente le principe d'exploration symbolique [S. Boyer et al., 1975, Howden, 1977, King, 1975, King, 1976] en 3.3.1.1 et les limitations de cette méthode en 3.3.1.2.

3.3.1.1/ PRINCIPE GÉNÉRAL

Le principe de l'exploration symbolique pour le programme de la figure 3.3 consiste à associer à x une valeur *symbolique*, c'est-à-dire une valeur à laquelle des contraintes sont associées au cours de l'exécution du programme sans pour autant qu'une valeur exacte ne lui soit affectée. On note ϕ la valeur symbolique associée à x.

La seule contrainte initialement associée à ϕ est $\phi \in \mathbb{Z}$ car x appartient à l'ensemble des entiers relatifs. La ligne L1 affecte à y la valeur $2 * \phi$, sous-entendu « deux fois une valeur de l'ensemble $\mathbb{Z}$ ». L'évaluation de la condition de la ligne L2 revient alors à évaluer $2 * \phi > 0$ qui est vraie pour toute valeur de ϕ supérieure à 0, et fausse pour toute valeur de ϕ inférieure ou égale à 0. L'exécution symbolique suit donc les deux chemins possibles en mémorisant les contraintes nécessaires pour suivre chacun des chemins : $2 * \phi > 0$ pour passer à L3 et $2 * \phi \leq 0$ pour ne pas entrer dans la condition.

Lorsque la condition en L2 est évaluée à faux, l'exécution de la fonction se termine car il n'existe plus aucune instruction à exécuter et L4 n'est donc pas déclenchée. Ce chemin n'est donc pas celui qui nous intéresse. Si on souhaite malgré tout obtenir une valeur de x qui permette *de ne pas passer* par L4, on peut utiliser un solveur de contraintes (SMT par exemple) avec la contrainte $\phi \in \mathbb{Z} \wedge \neg(2 * \phi > 0)$. Un modèle satisfaisant ces contraintes

pourrait être $\phi = 0$ par exemple et l'entrée $x = 0$ permet donc de tester le cas ou la ligne L4 n'est pas exécutée.

Lorsque la condition en L2 est évaluée à vrai en revanche, donc lorsqu'il existe une valeur de ϕ telle que $\phi \in \mathbb{Z} \wedge 2 * \phi > 0$, la condition de la ligne L3 est évaluée. L'évaluation effectuée est alors $\phi - 2 < 0$, et on construit à nouveau deux chemins : l'un où $\phi - 2 < 0$ et l'autre où $\neg(\phi - 2 < 0)$, ces deux conditions pouvant en effet être satisfaites. Si la condition en L3 est fausse, la fonction termine et ce chemin ne nous intéresse donc pas.

Si la condition en L3 est vraie, la contrainte pour la variable ϕ est la suivante : $\phi \in \mathbb{Z} \wedge 2 * \phi > 0 \wedge \phi - 2 < 0$. L'instruction en L4 est ensuite exécutée et ce dernier chemin va donc nous intéresser. A l'aide d'un solveur SMT, on peut donc demander un modèle pour la contrainte $\phi \in \mathbb{Z} \wedge 2 * \phi > 0 \wedge \phi - 2 < 0$. Le seul modèle satisfaisant cette contrainte est $\phi = 1$. Ainsi, $x = 1$ est une entrée qui amènera la fonction à exécuter la ligne L4.

3.3.1.2/ LIMITATIONS

L'exécution symbolique possède un certain nombre de limitations qui empêchent parfois son application. La principale limitation concerne son efficacité qui dépend du nombre de branchements à considérer (c'est-à-dire du nombre de chemins étudiés). En effet, les appels aux solveurs sont généralement coûteux en temps et leur nombre est proportionnel au nombre de chemins empruntés lors de l'exploration symbolique. Le passage à l'échelle sur des programmes de taille conséquente est donc compromis. Une deuxième limitation importante vient de l'incapacité des solveurs à résoudre certaines contraintes [Bradley et al., 2010]. Typiquement, certains solveurs sont incapables de gérer des contraintes non linéaires. La ligne L1 du programme de la figure 3.3 ($y := 2 * x$) génère une contrainte linéaire facilement résolue par la plupart des solveurs classiques. Si on remplace cette ligne par $y := x * x$ en revanche, la contrainte n'est plus linéaire et certains solveurs seront incapables d'assigner une valeur aux différentes variables du programme.

3.3.2/ EXPLORATION CONCOLIQUE

L'exploration dite concolique vise à résoudre les limitations de l'exploration symbolique. Elle combine l'exploration concrète et l'exploration dynamique (le terme « concolique » vient de la concaténation de « concrète » et « symbolique »). Considérons le programme de la figure 3.4 qui reprend celui de la figure 3.3 avec cette fois une contrainte non linéaire due à la ligne L1.

L'exploration concolique considère que x est donnée initialement (supposons qu'elle vaut 0). L'exécution concrète associe à y la valeur $0 * 0$ donc 0 et l'exploration symbolique considérera y comme le calcul $\phi * \phi$, où ϕ est la variable symbolique associée à x. Au niveau concret, la condition de la ligne L2 devient $0 > 0$, ce qui est faux et le programme n'exécutera donc ni L3, ni L4. Afin de passer par un autre chemin (un susceptible d'exécuter L4), il faut donc que x soit différent de 0. Imaginons que le solveur résolve la contrainte $x \neq 0$ avec $x = -1$. On a alors $y = -1 * -1 = 1$ et toujours le calcul $\phi * \phi$ pour remplacer y au niveau symbolique.

Au niveau symbolique, la contrainte ayant précédemment permis d'emprunter le chemin L0, L1, L2 et de terminer le programme sans passer par L3 ni L4 était $\neg(\phi * \phi > 0)$ ou

FIGURE 3.4 – Un exemple de programme produisant des contraintes non linéaires. L4 n'est exécutée que si la variable x est inférieure à 0 ou égale à 1.

```
L0 : function(x) {
L1 :   y := x * x;
L2 :   if (y > 0) {
L3 :     if (x - 2 < 0) {
L4 :       // ERROR
         }
       }
     }
```

écrit autrement $\phi * \phi \leq 0$. Afin de passer par L3, il faut donc symboliquement résoudre la négation de $\phi * \phi \leq 0$, soit $\phi * \phi > 0$. Il s'agit cependant d'une contrainte non linéaire pour laquelle un solveur ne pourrait pas déterminer de valeur pour ϕ. Pour éviter ce problème, l'exploration concolique simplifie la contrainte en utilisant la valeur concrète de y dans la condition L2, c'est-à-dire 1. La condition $1 > 0$ étant évaluée à vrai, la condition L3 $(x - 2 < 0)$ est évaluée. Puisque x vaut -1, la condition est vraie et L4 est finalement exécutée. Le chemin symbolique ayant permis d'atteindre L4 est alors contraint par la formule $\phi * \phi > 0 \wedge \phi - 2 < 0$.

Si on souhaite obtenir une valeur de x permettant de passer par L3 sans passer par L4, il faudrait poursuivre le procédé sans utiliser les contraintes de chemin non linéaires. Par exemple, il faudrait demander au solveur de résoudre $x \neq 0 \wedge x - 2 \geq 0$ afin de déterminer une nouvelle valeur de x. Dans le cas du programme de la figure 3.4, si le solveur propose une valeur de x supérieure à 1, par exemple 2, le programme exécutera effectivement L0, L1, L2, L3 sans exécuter L4 (car $2 * 2 > 0$ et $\neg(2 - 2 < 0)$).

3.3.3/ EXPLORATION SYMBOLIQUE PARTIELLE POUR CALCULER DES INSTANCES ATTEINTES DE CHAÎNES DE BALL

Dans [Bride et al., 2016a], la technique proposée permet d'exploiter les chaînes de Ball pour calculer une sous-approximation à partir d'une abstraction par prédicats de systèmes événementiels. Pour rappel, les chaînes de Ball sont des séquences de transitions de la forme $(must^-)^* - may^{0/1} - (must^+)^*$. L'abstraction par prédicats entraîne une abstraction de la structure de contrôle du système événementiel. Les instances des chaînes de Ball identifiées sur cette abstraction peuvent donc ne pas être atteintes depuis un état initial du système modélisé.

Pour augmenter le nombre d'instances de chaînes de Ball atteintes, la méthode développée consiste à calculer, de manière symbolique, l'ensemble des états pouvant être atteints après application d'un nombre n fini d'événements depuis les états initiaux du système. Cet ensemble d'états constitue alors un état symbolique qui peut être utilisé comme un état initial car il est composé strictement d'états pouvant être atteints depuis un état initial du système. Le calcul de cet état peut être réalisé par un parcours en avant des séquences de transitions de longueur n applicables depuis un état initial. Tous les états rencontrés lors de cette exploration appartiennent à l'état symbolique qu'on appellera $S(n)$.

Les chemins ayant permis d'atteindre un état concret particulier appartenant à $S(n)$ sont

également mémorisés. En effet, tout état concret de $S(n)$ peut être utilisé comme source des instances de chaînes de Ball qui seront calculées par la suite. Il convient donc de mémoriser au moins un chemin concret ayant mené le système dans cet état source.

Le 3MTS $\stackrel{def}{=} \langle Q_0, Q, \Delta, \Delta^-, \Delta^+ \rangle$ correspondant au système événementiel à partir d'un ensemble de prédicats d'abstraction donné est calculé dans un premier temps. Puis $S(n)$ est lui-même considéré comme un état abstrait, caractérisé par la disjonction des états concrets qui le composent. Il est donc ajouté à l'ensemble Q des états abstraits du 3MTS et est considéré comme l'unique état abstrait initial de ce dernier. Toute *may*-transitions dont la source est $S(n)$ est ensuite ajoutée à l'ensemble des *may*-transitions Δ et, si ces *may*-transitions possèdent une modalité *must*, les ensembles Δ^- et Δ^+ sont complétés en conséquence.

Le 3MTS $\stackrel{def}{=} \langle Q_0, Q, \Delta, \Delta^-, \Delta^+ \rangle$ ainsi modifié est alors constitué des ensembles suivants :

- $Q_0 \stackrel{def}{=} \{S(n)\}$,
- $Q \stackrel{def}{=} A \cup \{S(n)\}$,
- $\Delta \stackrel{def}{=} \Delta \cup \{S(n) \xrightarrow{e} q' \mid S(n) \xrightarrow{e} q'$ est une *may*-transition $\}$,
- $\Delta^- \stackrel{def}{=} \Delta^- \cup \{S(n) \xrightarrow{e} q' \mid S(n) \xrightarrow{e} q'$ est une $must^-$-transition $\}$,
- $\Delta^+ \stackrel{def}{=} \Delta^+ \cup \{S(n) \xrightarrow{e} q' \mid S(n) \xrightarrow{e} q'$ est une $must^+$-transition $\}$.

Les chaînes de Ball peuvent ensuite être instanciées à partir de l'état abstrait initial $S(n)$. Toutes les chaînes de Ball ainsi instanciées sont garanties d'être atteintes car $S(n)$ n'est composé que d'états concrets pouvant être atteints depuis un état initial. De plus, puisque les chemins menant aux états de $S(n)$ sont mémorisés, les chemins permettant d'atteindre les instances des chaînes de Ball depuis l'un des états initiaux du système sont connus.

La méthode garantit donc de calculer une sous-approximation du système. En effet, les chaînes de Ball sont nécessairement instanciables (voir [Ball, 2005]) par des transitions atteintes si l'état de départ de l'instance est lui-même atteint. Naturellement, plus la valeur de n est grande, plus l'état $S(n)$ contient d'état concrets atteints et plus les chances de pouvoir concrétiser des chaînes de Ball par des instances atteintes sont grandes. Cependant, le coût du calcul de $S(n)$ est exponentiel par rapport à n et est limité par la capacité du solveur à manipuler la formule qui lui correspond.

II

CONTRIBUTIONS

4

GÉNÉRATION D'UNE SOUS-APPROXIMATION COUVRANT AU MOINS UNE FOIS TOUS LES ÉTATS ET TOUTES LES TRANSITIONS DE L'ABSTRACTION : ALGORITHME CXP

Ce chapitre présente une méthode de génération d'une sous-approximation d'un système événementiel visant à couvrir au moins une fois tous les états et toutes les transitions d'un système abstrait correspondant à un système événementiel. La section 4.1 introduit les problématiques auxquelles la méthode tente d'apporter une solution. La section 4.2 présente les différents objectifs de la contribution développée dans ce chapitre et évoque ses principes détaillés de fonctionnement. La section 4.3 applique l'algorithme CXP à l'exemple du distributeur de café. La section 4.4 présente l'algorithme CXP_{ASO}, semblable à CXP en termes d'entrées, de sorties et d'objectifs, mais dont la méthode d'exploration est différente. La section 4.5 discute la complexité, la terminaison, la complétude et la correction de l'algorithme. Enfin, la section 4.6 énonce un bilan sur la contribution et conclut ce chapitre.

4.1/ PROBLÉMATIQUES

Le test exhaustif étant en pratique infaisable sur un système de grande taille ou infini, on travaille ici sur une sous-approximation d'un tel système, c'est-à-dire une partie de son LTS sémantique. Si l'abstraction par prédicat permet de réduire l'espace d'états considéré, elle entraîne également une perte d'informations sur l'atteignabilité de ces états et sur la relation de transitions qui les lient.

Une instance concrète d'un état abstrait ou d'une *may*-transition peut ne pas être atteignable sur le système réel alors que l'état abstrait ou la may-transition est atteignable. Typiquement, il est possible que des états concrets soient dans un état abstrait sans qu'il soit possible de mettre le système dans l'un de ces états. De même, il est possible qu'il existe des instances de *may*-transitions pour lesquelles il n'existe aucune séquence de transitions concrètes susceptible de les déclencher depuis l'un des états initiaux du système. Afin de générer une sous-approximation et non pas une sur-

approximation d'un système événementiel, ces états et transitions non atteignables (et donc non-instanciables sur le système réel) sont à éviter.

Par ailleurs, si les instances calculées sont bel et bien instanciables sur le système réel, il est préférable qu'il existe une séquence de transitions dans la sous-approximation qui permette de les atteindre depuis l'un des états concrets initiaux. Il est donc préférable de ne pas calculer une instance de may-transition aléatoirement mais plutôt d'en calculer une dont l'état concret source est atteint dans la sous-approximation.

4.2/ OBJECTIFS, PRINCIPES ET PRÉSENTATION DE L'ALGORITHME CXP

Cette section décrit l'algorithme CXP (pour « Concrete eXPloration ») qui calcule à la volée la modalité d'une transition et instancie chaque état abstrait may-atteignable et chaque transition abstraite may-atteignable au moins une fois, et jusqu'à trois fois. Ainsi, chaque fois qu'une nouvelle transition may-atteignable est découverte, elle sera instanciée par CXP. Cet algorithme vise essentiellement à calculer un ATS (voir définition 19 page 33) à partir duquel les tests générés couvriront autant d'états et de transitions que possible.

CXP utilise trois heuristiques pour améliorer la connectivité des instances calculées et maximiser le nombre de ces transitions qui seront atteintes. Améliorer la connectivité permet une meilleure couverture des états et transitions abstraites par les tests issus de l'ATS et permet de calculer plus d'instances atteintes. Maximiser le nombre de transitions atteintes permet de s'assurer que les tests résultants de l'ATS pourront être instanciés sur le système.

Le fonctionnement des trois heuristiques est décrit ci-dessous :

1. Heuristique d'ordonnancement des événements : cette heuristique permet au testeur d'indiquer à l'algorithme, par le biais d'une liste d'événements notée *oEv*, l'ordre dans lequel les événements doivent être traités. L'algorithme instanciera ainsi les transitions en suivant l'ordre dans lequel les événements sont donnés dans *oEv*. En choisissant judicieusement l'ordre de traitement des événements, la connectivité des transitions dans l'ATS est généralement augmentée.
2. Heuristique d'ordonnancement des états abstraits : cette heuristique, similaire à la précédente, permet au testeur d'indiquer à l'algorithme l'ordre dans lequel il doit tenter de rejoindre les états abstraits à partir d'un état abstrait et d'un événement donnés. La fonction *orderStates* : $A \times 2^A \rightarrow L_A$ (L_A étant une liste sans redondance contenant tous les éléments de A) prend en paramètre l'état abstrait à partir duquel une transition doit être concrétisée, ainsi que l'ensemble des états abstraits du système. Elle retourne une liste d'états abstraits. Les états abstraits cibles seront ainsi traités par CXP dans le même ordre qu'ils apparaissent dans la liste. En choisissant judicieusement l'ordre de traitement des états abstraits cibles, la connectivité des transitions dans l'ATS est généralement augmentée.
3. Coloration des états concrets : cette heuristique est appliquée par défaut par l'algorithme CXP. Elle attribue une couleur à chaque état concret instancié au cours de l'exploration. Un état est colorié en *vert* si son atteinte est attestée par l'existence d'une séquence de transitions concrètes de l'ATS partant d'un état initial (on parle d'état concret atteint). Un état est colorié en *bleu* s'il a été instancié sans

qu'on dispose d'une séquence de transitions concrètes de l'ATS le reliant à un état initial. Cette heuristique est utilisée pour améliorer la connectivité des transitions dans l'ATS en incitant l'algorithme à calculer des instances de transitions dont l'état concret source est atteint (*vert*).

4.2.1/ Entrées, sorties et variables de l'algorithme CXP

Cette section décrit les paramètres d'entrée et de sortie ainsi que les variables utilisées par CXP.

4.2.1.1/ Entrées

Les entrées de l'algorithme sont décrites ci-dessous :

- $\langle X, Inv, Init, Ev \rangle$: il s'agit du système événementiel pour lequel l'ATS doit être calculé.
- A : il s'agit de l'ensemble des états abstraits calculés à partir des prédicats d'abstraction choisis par le testeur. Cet ensemble permet de calculer l'ensemble $A \times Ev \times A$ des transitions abstraites qui seront instanciées par CXP si elles sont may-atteignables.
- *oEv* : il s'agit de la fonction d'ordonnancement des événements décrite au début de cette section. Cette fonction est donnée par le testeur et spécifie l'ordre dans lequel CXP doit traiter les événements. Généralement, le testeur devra utiliser l'ordre causal entre les événements s'il existe. La liste d'événements résultante ne doit contenir que les événements contenus dans Ev et doit les contenir tous exactement une fois.
- *orderStates* : il s'agit de la fonction d'ordonnancement des états abstraits décrite au début de cette section. Cette fonction est également donnée par le testeur et spécifie l'ordre dans lequel CXP doit traiter les états abstraits cibles des transitions à instancier. Cette fonction prend en paramètre l'état abstrait source de la transition à instancier ainsi que l'ensemble A des états abstraits. La liste d'états abstraits résultante doit contenir tous les états abstraits de A exactement une fois.

4.2.1.2/ Sorties

Les sorties de l'algorithme sont décrites ci-dessous :

- $\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa \rangle$: il s'agit de l'ATS calculé par CXP.

4.2.1.3/ Variables

Les variables utilisées par l'algorithme sont décrites ci-dessous :

- RQ : cet ensemble contient tous les états abstraits may-atteignables rencontrés au cours de l'exploration. Chacun de ces états abstraits sera utilisé comme source des transitions abstraites à instancier.
- q et q' : il s'agit respectivement des états abstraits source et cible des transitions à instancier.

- c (ou c_w) et c' (ou c'_w) : il s'agit respectivement des états concrets source et cible de l'instance de la transition abstraite calculée par CXP (i.e. $\alpha(c) = q$ et $\alpha(c') = q'$).
- C_{verts} : cet ensemble contient tous les états concrets de l'état abstrait source courant q calculés au cours de l'exploration et qui sont atteints depuis l'un des états initiaux de l'ATS (i.e. les états concrets coloriés en *vert* par l'heuristique de coloration des états concrets).
- C_{bleus} : cet ensemble contient tous les états concrets de l'état abstrait cible courant q' calculés au cours de l'exploration et dont on ne sait pas si ils sont atteints depuis l'un des états initiaux de l'ATS (i.e. les états concrets coloriés en *bleu* par l'heuristique de coloration des états concrets).

4.2.2/ FONCTIONNEMENT DE L'ALGORITHME CXP

Cette section décrit les deux étapes principales de l'algorithme CXP dont la formalisation est donnée par l'algorithme 1.

4.2.2.1/ CALCUL D'UNE INSTANCE CONCRÈTE DE CHAQUE ÉTAT ABSTRAIT INITIAL

La première étape (de la ligne 3 à 6) calcule une instance concrète de chaque état abstrait initial. Pour cela, CXP itère sur tous les états abstraits q de l'ensemble A donné en entrée (voir la boucle de la ligne 3 à la ligne 6). Si q peut contenir un état initial concret (i.e. la substitution d'initialisation *Init* peut mener à un état concret c de q (ligne 4)), cet état concret est ajouté à l'ensemble des états concrets initiaux C_0 de l'ATS. L'état concret c étant initial, l'état abstrait q l'est aussi et est donc ajouté à l'ensemble Q_0 des états abstraits initiaux. La fonction d'abstraction α est également enrichie puisque l'état abstrait correspondant à c est q. Tout état concret initial construit étant par définition atteint, la couleur associé à c est la couleur verte (voir la ligne 5 où Q_0, C_0, α et κ sont enrichis).

4.2.2.2/ CALCUL DES INSTANCES DE TRANSITIONS ABSTRAITES À PARTIR DE L'ÉTAT INITIAL

La seconde étape calcule au moins une instance concrète de chaque transition may-atteignable. Rappelons qu'une transition $q \xrightarrow{e} q'$ est may-atteignable si et seulement si c'est une may-transition et que l'une des conditions suivantes est vraie :

- soit l'état abstrait source q est un état abstrait initial (i.e. q appartient à Q_0),
- soit il existe au moins une sequence de may-transitions menant à l'état abstrait source q depuis l'un des états abstraits initiaux (i.e. un état de Q_0).

Cette étape de l'algorithme utilise l'ensemble RQ comme l'ensemble des états abstraits may-atteignables depuis lesquels l'exploration n'a pas encore eu lieu. Initialement (voir ligne 8), RQ contient tous les états abstraits initiaux calculés par la première étape. En effet, ces états abstraits sont may-atteignables et l'exploration n'a pas encore eu lieu. La boucle de la ligne 9 à la ligne 39 itère sur l'ensemble des états de RQ qui aura contenu, à la fin de l'exécution de l'algorithme, chaque état abstrait may-atteignable rencontré lors de l'exploration.

Algorithme 1 : Concrete Exploration (CXP)

Inputs : $\langle X, \mathit{Inv}, \mathit{Init}, Ev\rangle$: un système événementiel
A : un ensemble fini d'états abstraits
oEv : la liste ordonnée des événements de Ev
orderStates : $2^A \times A \to L_A$: la fonction d'ordonnancement des états abstraits
Output : $\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa\rangle$: un système de transitions approximé
Variables : RQ : l'ensemble des états abstraits restant à traiter
q, q' : les états abstraits source et cible de la transition à instancier
c, c' : les états concrets calculés pour les états abstraits q et q' respectivement
c_w, c'_w : les états concrets *témoins* de l'existence d'une *may*-transition $q \xrightarrow{e} q'$
C_{verts} : l'ensemble des états concrets *verts* dans q
C_{bleus} : l'ensemble des états concrets *bleus* dans q

1 $Q_0 := \emptyset$; $Q := \emptyset$; $\alpha := emptyMap()$; $\kappa := emptyMap()$; $C_0 := \emptyset$; $\Delta := \emptyset$; $\Delta^c := \emptyset$;
2 // Calcul d'une instance concrète de chaque état abstrait initial
3 **foreach** $q \in A$ **do**
4 $c := \mathtt{SAT}(prd_X(Init) \wedge q[{}^{X'}/_X])[{}^X/_{X'}]$;
5 **if** $c \notin \{unknown, unsat\}$ **then** $Q_0 := Q_0 \cup \{q\}$; $C_0 := C_0 \cup \{c\}$; $\alpha(c) := q$; $\kappa(c) := vert$; **end** ;
6 **end**
7 // Calcul d'une instance concrète de chaque *may*-transition
8 $Q := Q_0$; $C := C_0$; $RQ := Q_0$;
9 **while** $RQ \neq \emptyset$ **do**
10 **choose** $q \in RQ$;
11 $RQ := RQ - \{q\}$;
12 $Q := Q \cup \{q\}$;
13 **foreach** $e \stackrel{def}{=} a \in oEv$ **do**
14 **foreach** $q' \in orderStates(A, q)$ **do**
15 $(c_w, c'_w) := \mathtt{SAT}(prd_X(a) \wedge q \wedge q'[{}^{X'}/_X])$;
16 **if** $(c_w, c'_w) \notin \{unsat, unknown\}$ **then**
17 $\Delta := \Delta \cup \{q \xrightarrow{e} q'\}$;
18 $C_{verts} := \{c_q \mid \alpha(c_q) = q \wedge \kappa(c_q) = vert\}$;
19 $(c, c') := \mathtt{SAT}(prd_X(a) \wedge q'[{}^{X'}/_X] \wedge \bigvee_{c_q \in C_{verts}} c_q)$;
20 **if** $(c, c') \notin \{unsat, unknown\}$ **then**
21 $C := C \cup \{c'\}$;
22 $\alpha(c') := q'$;
23 $\Delta^c := \Delta^c \cup \{c \xrightarrow{e} c'\}$;
24 $\kappa(c') := vert$;
25 $C_{bleus} := \{c'_q \mid \alpha(c'_q) = q' \wedge \kappa(c'_q) = bleu\}$;
26 $(c, c') := \mathtt{SAT}(prd_X(a) \wedge \bigvee_{c_q \in C_{verts}} c_q \wedge (\bigvee_{c'_q \in C_{bleus}} c'_q)[{}^{X'}/_X])$;
27 **if** $(c, c') \notin \{unsat, unknown\}$ **then**
28 $\Delta^c := \Delta^c \cup \{c \xrightarrow{e} c'\}$; $\kappa(c') := vert$;
29 Colorier récursivement les successeurs de c' en $vert$
30 **end**
31 **end**
32 $C := C \cup \{c_w, c'_w\}$; $\Delta^c := \Delta^c \cup \{c_w \xrightarrow{e} c'_w\}$; $\alpha(c_w) := q$; $\alpha(c'_w) := q'$;
33 **if** $c_w \notin Dom(\kappa)$ **then** $\kappa(c_w) := bleu$ **end** ;
34 **if** $c'_w \notin Dom(\kappa) \vee \kappa(c_w) = vert$ **then** $\kappa(c'_w) := \kappa(c_w)$ **end** ;
35 **if** $q' \notin Q$ **then** $RQ := RQ \cup \{q'\}$ **end** ;
36 **end**
37 **end**
38 **end**
39 **end**
40 **return** $\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa\rangle$

Pour chaque état abstrait source may-atteignable q (ligne 10), l'algorithme vérifie pour chaque événement e de *oEv* (ligne 13) et chaque état abstrait cible q' (ligne 14) si $q \xrightarrow{e} q'$ est une may-transition (ligne 15). Lorsque c'est le cas, l'algorithme ajoute $q \xrightarrow{e} q'$ à l'ensemble des transitions may-atteignables Δ (ligne 17).

Lorsque le solveur détermine que $q \xrightarrow{e} q'$ est une may-transition (ligne 15), il fournit également des états concrets source (c_w) et cible (c'_w) qui témoignent qu'une instance de $q \xrightarrow{e} q'$ existe. L'instance concrète $c_w \xrightarrow{e} c'_w$ correspondant à la transition abstraite $q \xrightarrow{e} q'$ sera par la suite ajoutée à l'ensemble des transitions concrètes Δ^c (ligne 32). Cela garantit que l'algorithme calculera toujours au moins une instance concrète de chaque transition may-atteignable trouvée. Cependant, l'instance concrète $c_w \xrightarrow{e} c'_w$ n'est pas nécessairement atteignable sur le système (i.e. l'état concret c_w peut n'être atteignable par aucune séquence concrète de transitions applicable sur le système). De plus, même si cette transition est effectivement atteignable sur le système, la sous-approximation calculée par CXP pourrait très bien ne jamais l'atteindre depuis un état concret initial (si aucune autre instance de transition ne rejoint $c_w \xrightarrow{e} c'_w$ par exemple). On dit alors que cette dernière est *déconnectée* dans la sous-approximation.

Si c_w n'a pas encore de couleur attribuée (i.e. c_w n'a pas encore été rencontré lors de l'exploration), il est colorié en *bleu* ; ce qui signifie que son atteignabilité n'est pas connue (ligne 33). Si c_w était déjà connu comme étant atteint (i.e. c_w appartient à C et $\kappa(c_w) = green$) ou que c'_w n'était pas connu, c'_w est colorié de la même couleur que c_w. En effet, si c_w était *vert* (donc atteint), c'_w l'est également par la transition $c_w \xrightarrow{e} c'_w$, et si c_w n'était pas connu comme étant atteint et que c'_w n'était pas connu, alors c'_w prend la même couleur que c_w (la couleur *bleue*). Cela permet de s'assurer d'une part que chaque état concret rencontré lors de l'exploration possède une couleur par la fonction de coloration κ et qu'aucune information d'atteinte d'un état ne soit perdue.

Comme cela a déjà été dit dans la section 4.2, la connectivité des transitions permet d'améliorer la couverture des états et des transitions abstraites. Dans cet objectif, l'algorithme CXP peut calculer jusqu'à deux instances supplémentaires de la transition abstraite $q \xrightarrow{e} q'$, avec cette fois l'assurance que les instances calculées soient atteintes dans la sous-approximation. C'est ici que la fonction κ trouve toute son utilité.

4.2.2.3/ INSTANCIATION DES TRANSITIONS À PARTIR D'UN ÉTAT VERT

Une fois que la transition $q \xrightarrow{e} q'$ est connue comme étant may-atteignable (lorsque la condition de la ligne 16 est vraie), l'algorithme calcule l'ensemble C_{verts} des états concrets de q connus pour avoir été atteints dans la sous-approximation (i.e. les états concrets de q auxquels la fonction κ attribue la couleur *verte*) (voir ligne 18). Rappelons qu'initialement, tous les états concrets initiaux calculés par la première étape ont été coloriés en *vert*.

L'algorithme vérifie ensuite si la transition abstraite $q \xrightarrow{e} q'$ peut être instanciée depuis l'un des états concrets de C_{verts} (ligne 19). Si c'est le cas, cela signifie que l'instance concrète ($c \xrightarrow{e} c'$ avec c et c' donnés par le solveur ligne 19) de la transition abstraite $q \xrightarrow{e} q'$ est atteinte dans la sous-approximation, puisque son état concret source c est connu comme étant atteint. L'instance concrète $c \xrightarrow{e} c'$ est donc ajoutée à l'ensemble des transitions concrètes Δ^c (ligne 23).

4.2.2.4/ INSTANCIATION DES TRANSITIONS À PARTIR D'UN ÉTAT VERT VERS UN ÉTAT BLEU

L'algorithme calcule ensuite l'ensemble C_{bleus} des états concrets de q' coloriés en *bleu* par κ (ligne 25), puis tente de calculer une instance concrète de la transition abstraite $q \xrightarrow{e} q'$ qui connecte l'un des états de C_{bleus} depuis un état concret atteint, c'est-à-dire un état contenu dans C_{verts} (ligne 26). Si une telle instance existe, la transition $c \xrightarrow{e} c'$ (avec c et c' les témoins donnés par le solveur ligne 26) est atteinte (puisque c appartient à C_{verts}) et est ajouté à l'ensemble des transitions concrètes Δ^c (ligne 28). L'état concret c', auparavant colorié en *bleu*, est colorié en *vert* et cette couleur est également propagée à tous les successeurs de c' connus par Δ^c, ainsi qu'aux successeurs de ses successeurs, et ainsi de suite récursivement (ligne 29).

4.2.2.5/ INSTANCIATION DES TRANSITIONS ABSTRAITES À PARTIR DES ÉTATS *may*-ATTEIGNABLES

Si l'algorithme détermine que la transition $q \xrightarrow{e} q'$ est may-atteignable, cela signifie que q' est un état may-atteignable (puisque q est nécessairement un état may-atteignable). L'exploration doit donc continuer depuis l'état q'. On note au passage que q' pourrait parfaitement être le même état abstrait que q, ou même qu'un autre état abstrait déjà rencontré et depuis lequel l'exploration a déjà eu lieu. Pour cette raison, q' n'est ajouté aux états abstraits à partir desquels l'exploration doit se propager (RQ) que s'il n'est pas encore connu (i.e. il n'appartient pas à l'ensemble des états abstraits connus Q) (ligne 35). La boucle principale de la ligne 9 à la ligne 39 ne termine que lorsque l'exploration a été effectuée à partir de tous les états abstraits may-atteignables découverts au cours de l'exploration (i.e. lorsque RQ est vide).

Pour résumer, la seconde étape effectue au moins un (et jusqu'à trois) appel(s) au solveur (aux lignes 15, 19 et 26). Le premier appel vérifie si la transition $q \xrightarrow{e} q'$ est may-atteignable et, si c'est le cas, l'instance concrète correspondante $c_w \xrightarrow{e} c'_w$ est ajoutée à Δ^c, mais n'est pas nécessairement atteinte dans la sous-approximation. Le second appel au solveur vérifie si une instance de $q \xrightarrow{e} q'$ telle que l'état concret source c est atteint (i.e. colorié en *vert* par κ) peut être trouvée et, si c'est le cas, l'instance concrète correspondante $c \xrightarrow{e} c'$ est ajoutée à Δ^c, et les tests générés depuis la sous-approximation couvriront $q \xrightarrow{e} q'$ de manière certaine. Le troisième appel au solveur vérifie si une instance de $q \xrightarrow{e} q'$ dont l'état concret source est atteint et qui connecterait un état *bleu* de q' peut être trouvée et, si c'est le cas, l'instance concrète $c \xrightarrow{e} c'$ est ajoutée à Δ^c et non seulement les tests générés à partir de la sous-approximation couvriraient $q \xrightarrow{e} q'$, mais toutes les transitions atteintes depuis c' dans la sous-approximation le seraient également.

4.3/ APPLICATION À L'EXEMPLE FIL ROUGE

Dans cette section, le fonctionnement de l'algorithme CXP est illustré sur le distributeur de café.

4.3.1/ PARAMÈTRES EN ENTRÉE

L'ensemble de prédicats d'abstraction utilisé est $\mathcal{P} \stackrel{def}{=} \{p_0, p_1, p_2\}$ (identique à celui utilisé dans l'exemple 2 page 27) où :

- $p_0 \stackrel{def}{=} \mathit{Status} = \mathit{off} \wedge \mathit{Pot} \geq \mathit{MAX\_POT} - 50$
- $p_1 \stackrel{def}{=} \mathit{Status} = \mathit{on}$
- $p_2 \stackrel{def}{=} (\mathit{Status} = \mathit{on} \wedge \mathit{AskChange} = \mathit{false} \wedge \mathit{AskCof} = \mathit{false} \wedge \mathit{Balance} = 0) \vee \mathit{Status} = \mathit{error}$

L'ensemble d'états abstraits A en entrée de l'algorithme CXP contient donc les états abstraits suivants (en omettant les états abstraits vides) qui sont représentés dans la figure 2.4 (voir exemple 3) :

- $q_0 \stackrel{def}{=} \neg p_0 \wedge \neg p_1 \wedge \neg p_2$
- $q_1 \stackrel{def}{=} \neg p_0 \wedge \neg p_1 \wedge p_2$
- $q_2 \stackrel{def}{=} \neg p_0 \wedge p_1 \wedge \neg p_2$
- $q_3 \stackrel{def}{=} \neg p_0 \wedge p_1 \wedge p_2$
- $q_4 \stackrel{def}{=} p_0 \wedge \neg p_1 \wedge \neg p_2$

Par souci de simplicité, les événements sont traités par ordre alphabétique. La fonction d'ordonnancement des événements *oEv* retourne donc toujours les événements dans l'ordre suivant : *addCof*, *autoOut*, *backBalance*, *changeReq*, *cofReq*, *insert50*, *insert100*, *powerDown*, *powerUp*, *serveCof*, *takePot*.

Similairement, la fonction *orderStates* utilisée permet de traiter les transitions réflexives en premier et retourne donc l'état abstrait source de la transition abstraite en cours d'instanciation suivi de tous les autres états abstraits dans l'ordre lexicographique. Par exemple, si l'état abstrait source de la transition à instancier est q_2, la fonction *orderStates* retourne la liste des états abstraits cibles $L_A \stackrel{def}{=} [q_2, q_0, q_1, q_3, q_4]$.

4.3.2/ CALCUL D'UNE INSTANCE CONCRÈTE DE CHAQUE ÉTAT ABSTRAIT INITIAL

Comme cela a déjà été dit auparavant, la substitution d'initialisation *Init* du distributeur de café est déterministe : elle attribue une valeur à chacune des variables d'état du système. Ainsi, le système ne possède qu'un seul état concret initial : $c_0 \stackrel{def}{=} \mathit{Balance} = 0 \wedge \mathit{Pot} = 0 \wedge \mathit{Status} = \mathit{off} \wedge \mathit{CofLeft} = 10 \wedge \mathit{AskCof} = \mathit{false} \wedge \mathit{AskChange} = \mathit{false}$. Il n'existe donc qu'un seul état abstrait initial pour ce système : l'état abstrait q_0 contenant c_0.

La première partie de l'algorithme CXP vérifie si l'application de la substitution d'initialisation *Init* peut mener dans l'un des états abstraits de A (q_0, q_1, q_2, q_3 ou q_4). Ce n'est le cas que de q_0 car la substitution d'initialisation impose que le *Pot* soit 0 (donc inférieur à $\mathit{MAX\_POT} - 50$), ce qui implique que p_0 doit être faux, que le *Status* soit *off* (donc différent de *on* et *error*), ce qui implique que p_1 et p_2 soient tous les deux faux. L'état abstrait q_0 est donc le seul état abstrait initial et est ajouté à l'ensemble des états abstraits initiaux Q_0.

La seule instance concrète de q_0 dans laquelle la substitution d'initialisation peut mener étant c_0, cet état concret est ajouté à l'ensemble des états concrets initiaux C_0 de l'ATS qui sera retourné par CXP. Puisque c_0 est un état initial, il est atteint et donc colorié en *vert*. Son état abstrait correspondant est q_0 et la fonction d'abstraction α est donc enrichie ($\alpha(c_0) = q_0$).

Une fois les étapes d'instanciation des états abstraits initiaux (de la ligne 3 à la ligne 6) et d'initialisation des ensembles Q, C et RQ (à la ligne 8) terminées, CXP a construit

l'ATS $\stackrel{def}{=} \langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa \rangle$ où :

- $Q_0 = \{q_0\}$
- $Q = Q_0 = \{q_0\}$
- $\Delta = \emptyset$
- $C_0 = \{c_0\}$
- $C = C_0 = \{c_0\}$
- $\Delta^c = \emptyset$
- $\alpha = \{c_0 \mapsto q_0\}$
- $\kappa = \{c_0 \mapsto vert\}$

4.3.3/ CALCUL DES INSTANCES DE TRANSITIONS ABSTRAITES À PARTIR DE L'ÉTAT INITIAL

Au début de l'étape d'instanciation des transitions abstraites (de la ligne 9 à la ligne 39), RQ ne contient que l'état abstrait q_0. C'est donc cet état qui sera source des premières transitions instanciées. Ainsi, l'état q_0 est retiré de RQ, l'état source q est affecté à q_0, et q_0 est ajouté à Q (qui le contenait déjà).

Par la suite, pour tout événement e du système (traité par ordre alphabétique par la fonction *oEv*) et tout état abstrait q' de A (traité dans l'ordre imposé par *orderStates*), CXP tente de trouver une instance de la transition $q_0 \xrightarrow{e} q'$. L'ordre de tentative d'instanciation des transitions abstraites est donc le suivant :

1. $q_0 \xrightarrow{addCof} q_0$, $q_0 \xrightarrow{addCof} q_1$, ..., $q_0 \xrightarrow{addCof} q_4$
2. $q_0 \xrightarrow{autoOut} q_0$, $q_0 \xrightarrow{autoOut} q_1$, ..., $q_0 \xrightarrow{autoOut} q_4$
3. ...
4. $q_0 \xrightarrow{takePot} q_0$, $q_0 \xrightarrow{takePot} q_1$, ..., $q_0 \xrightarrow{takePot} q_4$

Parmi toutes ces transitions, seule une partie possède la modalité *may* et est donc effectivement instanciable et instanciée. Il s'agit des transitions suivantes :

- $q_0 \xrightarrow{addCof} q_0$
- $q_0 \xrightarrow{powerUp} q_3$

La première instance de $q_0 \xrightarrow{addCof} q_0$ calculée est la transition concrète $c_1 \xrightarrow{addCof} c_2$ où :

- $c_1 \stackrel{def}{=}$ *Balance* $= 100 \wedge$ *Pot* $= 0 \wedge$ *Status* $=$ *off* $\wedge$ *CofLeft* $= 0 \wedge$ *AskCof* $=$ *false* $\wedge$ *AskChange* $=$ *false*
- $c_2 \stackrel{def}{=}$ *Balance* $= 100 \wedge$ *Pot* $= 0 \wedge$ *Status* $=$ *off* $\wedge$ *CofLeft* $= 1 \wedge$ *AskCof* $=$ *false* $\wedge$ *AskChange* $=$ *false*

L'ATS est enrichi avec les deux nouveaux états concrets c_1 et c_2, la *may*-transition $q_0 \xrightarrow{addCof} q_0$ et la transition concrète $c_1 \xrightarrow{addCof} c_2$, malgré le fait qu'elle ne soit pas atteinte (car il faudrait pour cela que l'état concret source soit l'état initial c_0, ce qui n'est pas le cas). La transition $c_1 \xrightarrow{addCof} c_2$ étant une instance de la *may*-transition $q_0 \xrightarrow{addCof} q_0$, l'état abstrait correspondant à c_1 et c_2 est q_0. La fonction d'abstraction α est donc enrichie. Enfin, puisque $c_1 \xrightarrow{addCof} c_2$ n'est pas atteinte et puisque c_1 n'a pas encore de couleur attribuée, c_1 et c_2 sont tous deux coloriés en *bleu* dans la fonction κ.

La première instance de $q_0 \xrightarrow{powerUp} q_3$ calculée est la transition concrète $c_2 \xrightarrow{powerUp} c_4$ où :

- $c_4 \stackrel{def}{=}$ *Balance* $= 0 \wedge$ *Pot* $= 0 \wedge$ *Status* $=$ *on* $\wedge$ *CofLeft* $= 1 \wedge$ *AskCof* $=$ *false* $\wedge$ *AskChange* $=$ *false*

Puisque l'état c_2 est *bleu*, la transition $c_0 \xrightarrow{powerUp} c_3$ n'est pas non plus atteinte. L'ATS est enrichi avec le nouvel état concret c_4, l'état abstrait q_3, la *may*-transition $q_0 \xrightarrow{powerUp} q_3$ et

la transition concrète $c_2 \xrightarrow{powerUp} c_4$. La transition $c_2 \xrightarrow{powerUp} c_4$ étant une instance de la *may*-transition $q_0 \xrightarrow{powerUp} q_3$, l'état abstrait correspondant à c_4 est q_3. La fonction d'abstraction α est donc enrichie. Enfin, puisque $c_2 \xrightarrow{powerUp} c_4$ n'est pas atteinte, c_4 est colorié en *bleu* dans la fonction κ.

Après l'étape d'instanciation des transitions abstraites dont l'état source est q_0, sans considérer les instances calculées à partir des états verts ou bleus, CXP a construit l'ATS $\stackrel{def}{=} \langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa \rangle$ suivant :

- $Q_0 = \{q_0\}$
- $Q = \{q_0, q_3\}$
- $\Delta = \{q_0 \xrightarrow{addCof} q_0, q_0 \xrightarrow{powerUp} q_3\}$
- $C_0 = \{c_0\}$
- $C = \{c_0, c_1, c_2, c_4\}$
- $\Delta^c = \{c_1 \xrightarrow{addCof} c_2, c_2 \xrightarrow{powerUp} c_4\}$
- $\alpha = \{c_0 \mapsto q_0, c_1 \mapsto q_0, c_2 \mapsto q_0, c_4 \mapsto q_3\}$
- $\kappa = \{c_0 \mapsto vert, c_1 \mapsto bleu, c_2 \mapsto bleu, c_4 \mapsto bleu\}$

4.3.4/ INSTANCIATION DES TRANSITIONS À PARTIR D'UN ÉTAT VERT

Après avoir calculé la première instance de la transition $q_0 \xrightarrow{addCof} q_0$, CXP tente de calculer une nouvelle instance dont l'état source est un état *vert* de q_0 afin d'obtenir une instance atteinte.

Le seul état *vert* de q_0 connu après le calcul de la première instance $c_1 \xrightarrow{addCof} c_2$ est c_0 car c_1, c_2 et c_4, qui sont les seuls autres états concrets connus, sont tous les trois coloriés en *bleu*. Ainsi, l'ensemble des états *verts* de q_0 calculé par CXP est $C_{verts} = \{c_0\}$.

L'algorithme tente alors de calculer une instance concrète de $q_0 \xrightarrow{addCof} q_0$ dont l'état source soit l'un des états de C_{verts} (donc c_0). Dans c_0, le nombre de cafés restant dans la machine (*CofLeft*) est égal au nombre maximum de cafés qu'elle peut contenir (*CofLeft* = *MAX_COF* = 10). L'ajout de café ne peut donc pas avoir lieu depuis cet état. Par conséquent, aucune instance de $q_0 \xrightarrow{addCof} q_0$ autre que $c_1 \xrightarrow{addCof} c_2$ n'est calculée par CXP.
N.B. Cela ne signifie pas que la transition $c_1 \xrightarrow{addCof} c_2$ ne sera jamais atteinte dans l'ATS car une instance atteinte d'une autre transition abstraite pourrait rejoindre c_1, faisant ainsi de $c_1 \xrightarrow{addCof} c_2$ une transition atteinte.

Après avoir calculé la première instance de la transition $q_0 \xrightarrow{powerUp} q_3$ cette fois-ci, le même procédé à lieu : CXP tente de trouver une instance de $q_0 \xrightarrow{powerUp} q_3$ dont l'état concret source soit un état connu *vert* (donc c_0). Dans l'état initial c_0, le *Status* est *off*, le nombre de cafés restant *CofLeft* est supérieur à 0 et le *Pot* est inférieur à *MAX_POT*. Ainsi c_0 réunit toutes les conditions pour déclencher l'événement *powerUp*.

L'instance calculée est $c_0 \xrightarrow{powerUp} c_3$ où :

- $c_3 \stackrel{def}{=}$ *Balance* = 0 ∧ *Pot* = 0 ∧ *Status* = *on* ∧ *CofLeft* = 10 ∧ *AskCof* = *false* ∧ *AskChange* = *false*

Puisque c_0 est atteint, c_3 l'est aussi et il est donc colorié en *vert*. L'ATS est enrichi avec le nouvel état concret c_3 et la nouvelle transition concrète $c_0 \xrightarrow{powerUp} c_3$. La fonction d'abstraction est également enrichie : c_3 appartient à l'état abstrait q_3.

4.3.5/ INSTANCIATION DES TRANSITIONS À PARTIR D'UN ÉTAT VERT VERS UN ÉTAT BLEU

Puisqu'une instance de la transition $q_0 \xrightarrow{powerUp} q_3$ dont l'état concret source est $vert$ à été découverte (l'instance $c_0 \xrightarrow{powerUp} c_3$), l'algorithme CXP tente de calculer une instance dont l'état source est un état $vert$ connu de q_0 et dont l'état cible est un état $bleu$ connu de q_3.

Le seul état $vert$ connu de q_0 est toujours c_0. L'événement *powerUp* étant déterministe, le seul état concret qui puisse être atteint depuis c_0 par cet événement a déjà été atteint (il s'agit de c_3). Par conséquent, aucune autre instance de la transition $q_0 \xrightarrow{powerUp} q_3$ n'est calculée par CXP.

Si une telle transition avait existé en revanche, l'état cible $bleu$ aurait été colorié en $vert$ car l'état source de l'instance (c_0) serait $vert$. De même, tous les successeurs de la cible par la relation de transition Δ^c auraient été coloriés en $vert$ récursivement.

Après toutes ces étapes, CXP a donc calculé l'ATS suivant :

- $Q_0 = \{q_0\}$
- $Q = \{q_0, q_3\}$
- $\Delta = \{q_0 \xrightarrow{addCof} q_0, q_0 \xrightarrow{powerUp} q_3\}$
- $C_0 = \{c_0\}$
- $C = \{c_0, c_1, c_2, c_3, c_4\}$
- $\Delta^c = \{c_1 \xrightarrow{addCof} c_2, c_2 \xrightarrow{powerUp} c_4, c_2 \xrightarrow{powerUp} c_3\}$
- $\alpha = \{c_0 \mapsto q_0, c_1 \mapsto q_0, c_2 \mapsto q_0, c_4 \mapsto q_3, c_3 \mapsto q_3\}$
- $\kappa = \{c_0 \mapsto vert, c_1 \mapsto bleu, c_2 \mapsto bleu, c_4 \mapsto bleu, c_3 \mapsto bleu\}$

4.3.6/ INSTANCIATION DES TRANSITIONS ABSTRAITES À PARTIR DES ÉTATS *may*-ATTEIGNABLES

Les nouveaux états abstraits découverts au cours de l'exploration sont ajoutés à l'ensemble RQ. Ici, q_0 a déjà été utilisé comme état source des transitions à instancier. Il n'est donc pas ajouté à RQ. L'état abstrait q_3 rencontré lors de l'exploration en revanche est ajouté à RQ. L'algorithme calculera donc ensuite des instances de transitions dont l'état source est q_3 exactement de la même manière qu'il l'a fait pour q_0.

L'exploration s'arrête lorsque tous les états abstraits *may*-atteignables rencontrés ont servi d'état source des transitions à instancier. Dans le cas du distributeur de café, tous les états abstraits non vides de A sont *may*-atteignables. L'exploration aura donc lieu depuis les états abstraits q_0, q_1, q_2, q_3 et q_4.

La figure 4.1 représente l'ATS retourné par CXP sous la forme d'un graphe. Les états concrets sont représentés par des rectangles aux coins arrondis. Leur nom est la juxtaposition du nom d'état concret c_i et abstrait q_j. L'indice du nom de l'état concret (i) indique le numéro d'ordre avec lequel il a été inséré dans C. Par exemple, l'état concret c_8q_2 est le $9^{\text{ème}}$ état concret inséré dans C au cours de l'exploration (c_0 étant le 1^{er}) et appartient à l'état abstrait q_2. Les états atteints apparaissent en $vert$ (avec une bordure continue sur la figure) (par exemple c_0q_0, c_3q_3 et c_8q_2) et les états dont on ne sait pas s'ils sont atteints apparaissent en $bleu$ (par exemple $c_{39}q_4$ et $c_{40}q_3$) avec une bordure en pointillés sur la figure.

Les transitions sont représentées par une flèche entre deux états concrets et portent le nom de l'événement qu'elles appliquent entre ces deux états. La flèche est en pointillés

dans le cas où la transition n'est pas atteinte. Par exemple, la transition $c_{39}q_4 \xrightarrow{powerUp} c_{40}q_3$ est en pointillés, donc non atteinte, et la transition $c_0q_0 \xrightarrow{powerUp} c_3q_3$ est en trait continu car atteinte depuis l'état initial.

4.4/ INFÉRENCE AUTOMATIQUE DE L'ORDRE DE TRAITEMENT DES ÉTATS ABSTRAITS : ALGORITHME $\mathrm{CXP}_{\mathrm{ASO}}$

Si un ordre de traitement des événements efficace est en règle générale facile à proposer par le testeur en connaissant la causalité entre les différents événements, un ordre de traitement des états abstraits est beaucoup moins évident à inférer. Dans cette section, on propose une heuristique apportée à l'algorithme CXP permettant d'inférer automatiquement et dynamiquement un ordre de traitement des états abstraits. Cette technique d'ordonnancement permet généralement d'augmenter la couverture obtenue avec CXP. Elle nécessite de modifier le fonctionnement global de l'algorithme CXP tout en préservant le principe d'instanciation de chaque transition abstraite une fois (en l'occurrence, exactement une fois).

Afin de distinguer les deux méthodes (celle avec conception *manuelle* de la fonction d'ordonnancement des états abstraits et celle avec ordonnancement *automatique* de ces états), on appellera cette nouvelle méthode $\mathrm{CXP}_{\mathrm{ASO}}$ pour « Concrete eXPloration with Automatic States Ordering ».

La formalisation de l'algorithme $\mathrm{CXP}_{\mathrm{ASO}}$ est donnée par l'algorithme 2. Pour des raisons de clarté et afin de ne pas répéter ce qui a déjà été dit sur CXP, seules les différences notables entre CXP et $\mathrm{CXP}_{\mathrm{ASO}}$ sont évoquées dans cette section.

N.B. L'algorithme $\mathrm{CXP}_{\mathrm{ASO}}$ ordonnant lui-même les états abstraits, le paramètre d'entrée *orderStates* n'est plus nécessaire.

4.4.1/ CALCUL DE L'ENSEMBLE DES ÉTATS ABSTRAITS CIBLES

Au lieu de parcourir un par un l'ensemble des états abstraits cibles comme CXP le fait, $\mathrm{CXP}_{\mathrm{ASO}}$ constitue plutôt l'ensemble RQ' des états abstraits cibles q' de A tels que la transition $q \xrightarrow{e} q'$ n'a pas encore été instanciée (ligne 15). Ainsi, lorsqu'aucune instanciation n'a encore eu lieu, l'ensemble RQ' est exactement égal à A. Afin de ne pas instancier plusieurs fois une même transition abstraite, q' est retiré de RQ' lorsqu'une transition $q \xrightarrow{e} q'$ est instanciée (ligne 28).

4.4.2/ CALCUL DE L'ENSEMBLE DES ÉTATS CONCRETS CONNUS DANS L'ÉTAT ABSTRAIT SOURCE

$\mathrm{CXP}_{\mathrm{ASO}}$ calcule ensuite l'ensemble RCS des états concrets connus (donc se trouvant dans C) appartenant à l'état abstrait source q de la transition à instancier (ligne 19). Avant toute instanciation, seule une instance concrète de chaque état abstrait initial se trouve donc dans RCS. Par la suite, l'ensemble RCS sera de nouveau calculé mais contiendra toujours des états atteints. En effet, $\mathrm{CXP}_{\mathrm{ASO}}$ n'ajoute que des états atteints à l'ensemble

C duquel est issu RCS. Le calcul de RCS permet de ne calculer que des instances de transitions connectées entre elles et atteintes dans l'ATS calculé.

4.4.3/ CALCUL D'UNE INSTANCE D'UNE *may*-TRANSITION DONT L'ÉTAT CIBLE EST DANS RQ'

L'algorithme CXP_{ASO} calcule ensuite une instance de transition abstraite $q \xrightarrow{e} q'$ dont l'état abstrait cible q' est dans RQ'. C'est le solveur SMT qui détermine, si il existe, un état abstrait q' pouvant être rejoint depuis un état concret de RCS (donc dont l'état abstrait est q) par l'événement e.

Lorsqu'il existe une *may*-transition $q \xrightarrow{e} q'$ telle que $q' \in RQ'$, le solveur en calcule une instance $c \xrightarrow{e} c'$ telle que c est dans RCS et c' est dans q' (ligne 21). Dans ce cas, l'appel au solveur ne retourne plus une paire d'états concrets (c, c') mais un triplet (q', c, c') où :

- q' est l'état abstrait sélectionné par le solveur,
- $c \in RCS$ est l'état concret source de l'instance calculée,
- c' tel que $\alpha(c') = q'$ est l'état concret cible de l'instance calculée.

Puisqu'une instance $c \xrightarrow{e} c'$ de la transition abstraite $q \xrightarrow{e} q'$ dont l'état source c est atteint vient d'être calculée, $q \xrightarrow{e} q'$ est une *may*-transition et est ajoutée à Δ (ligne 24). De même, ligne 25, c' est ajouté à C, $c \xrightarrow{e} c'$ est ajoutée à Δ^c et l'état abstrait associé à c' par la fonction d'abstraction α est q'. Enfin, c' est colorié en *vert* par la fonction de coloration κ car l'état source de l'instance est dans RCS, qui ne contient que des états également *verts*. On remarque que la transition $q \xrightarrow{e} q'$ est donc couverte dans l'ATS par l'instance $c \xrightarrow{e} c'$.

L'état abstrait q' étant *may*-atteignable, il est ajouté à l'ensemble RQ (identique à celui de CXP) des états abstraits à partir desquels l'exploration doit avoir lieu (ligne 27).

4.4.4/ CALCUL D'UNE INSTANCE POUR TOUTES LES AUTRES *May*-TRANSITIONS DONT L'ÉTAT CIBLE EST DANS RQ'

Lorsqu'une transition $q \xrightarrow{e} q'$ est instanciée, l'état abstrait q' est retiré de RQ' afin de ne pas l'instancier de nouveau. Les phases de calcul de l'ensemble des états concrets connus dans l'état abstrait source q (RCS) puis de calcul d'une instance d'une *may*-transition dont l'état cible est dans RQ' sont alors répétées tant qu'une *may*-transition n'ayant pas encore été instanciée existe et peut être instanciée depuis un état de RCS (*vert*).

Ainsi, pour un état abstrait source q donné et un événement e donné, toutes les *may*-transitions $q \xrightarrow{e} q'$ pouvant être instanciées depuis un état connu de q (donc rencontré au cours de l'exploration et dans RCS) le seront avec CXP_{ASO} et seront couvertes.

Similairement à CXP, l'exploration continue depuis tous les états de RQ tant que celui-ci n'est pas vide (ligne 9), c'est-à-dire tant qu'il existe encore des transitions abstraites pour lesquelles aucune tentative d'instanciation depuis un état atteint n'a eu lieu.

4.4.5/ COMPLÉTION DU MTS PARTIELLEMENT CALCULÉ PAR CXP_{ASO}

CXP_{ASO} vérifie s'il existe une instance de transition $q \xrightarrow{e} q'$ dont l'état source est connu, mais ne vérifie pas au préalable que $q \xrightarrow{e} q'$ soit bien une *may*-transition. Si une telle instance existe, $q \xrightarrow{e} q'$ est bien une *may*-transition (raison pour laquelle elle est ajoutée à Δ), mais si elle n'existe pas, cela ne signifie pas que $q \xrightarrow{e} q'$ ne peut pas être instanciée : il est uniquement certain qu'elle ne peut pas l'être depuis un état connu de q.

Par conséquent, le MTS calculé par la première partie de CXP_{ASO} est potentiellement incomplet, en ce sens qu'il peut exister des états et des transitions *may*-atteignables qui n'ont pas été identifiées lors de cette première phase d'exploration. Afin de le compléter, et ainsi obtenir un CTS comparable à celui de CXP, une dernière étape de calcul d'une instance concrète des transitions abstraites n'ayant pas encore été instanciées (n'appartenant pas à Δ) est donc effectuée (lignes 34 à 48). Similairement à CXP, chaque fois qu'une instance $c \xrightarrow{e} c'$ d'une transition $q \xrightarrow{e} q'$ est trouvée, $q \xrightarrow{e} q'$ est ajoutée à Δ, c et c' sont ajoutés à C, $c \xrightarrow{e} c'$ est ajoutée à Δ^c, α est enrichie ($\alpha := \alpha \cup \{(c \mapsto q), (c' \mapsto q')\}$) et c et c' sont coloriés en *bleu* s'ils n'étaient pas encore connus (puisqu'on ne sait pas s'ils sont atteints, à moins qu'il ne s'agisse d'états rencontrés lors des étapes précédentes de CXP_{ASO}).

Après cette étape, le MTS calculé par CXP_{ASO} contient tous les états abstraits et les transitions abstraites *may*-atteignables et l'algorithme retourne l'ATS $\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa \rangle$ calculé (ligne 49) dont le CTS contient une instance concrète de chaque état abstraite et de chaque transition abstraite. L'ajout des transitions bleues n'a pas d'intérêt pour CXP_{ASO}, mais pourra en avoir dans les étapes ultérieures qui pourraient les reconnecter et ainsi en faire des transitions vertes.

4.4.6/ APPLICATION À L'EXEMPLE DU DISTRIBUTEUR DE CAFÉ

Puisque le procédé d'exploration utilisé par CXP_{ASO} est différent de celui de CXP, l'ATS généré peut évidemment varier. Typiquement, CXP_{ASO} peut instancier une transition abstraite $q \xrightarrow{e} q'$ par une transition concrète différente de l'instance (ou des instances) utilisée(s) par CXP. Cela signifie que le CTS (la partie concrète) de l'ATS calculé par CXP_{ASO} peut être différent de celui calculé par CXP. Le MTS reste en revanche inchangé car CXP et CXP_{ASO} calculent tous les états et transitions *may*-atteignables.

En l'occurrence, dans le cas du distributeur de café, le CTS calculé par CXP_{ASO}, donné par la figure 4.2, est légèrement différent de celui calculé par CXP. Les transitions en pointillés correspondent à des instances de transitions abstraites *may*-atteignables qui ne peuvent pas être couvertes par une instance dont la source est un état connu et atteint (un des états dont la bordure est continue sur la figure).

Pour ce qui est des transitions couvertes (celles ayant un trait continu), on constate que les instances calculées par CXP_{ASO} permettent de couvrir les mêmes transitions abstraites que celles calculées par CXP, à une exception près. La transition concrète $c_6q_2 \xrightarrow{\textit{backBalance}} c_1q_3$ calculée par CXP_{ASO} est atteinte et permet de couvrir la transition abstraite $q_2 \xrightarrow{\textit{backBalance}} q_3$. Cette transition abstraite n'est pas couverte dans l'ATS calculé par CXP, ce qui indique que CXP_{ASO} couvre une transition abstraite de plus que CXP. Une autre différence intéressante entre CXP et CXP_{ASO} est le fait que CXP_{ASO} calcule moins d'instances de transitions (voir les 33 instances de transitions calculées par CXP contre

25 pour CXP_{ASO}), tout en offrant une meilleure couverture des transitions abstraites que CXP.

Les résultats expérimentaux montreront d'ailleurs que CXP_{ASO} couvre toujours au moins autant (parfois plus) de transitions abstraites que CXP sur l'ensemble des systèmes étudiés dans cette thèse.

4.5/ DISCUSSIONS SUR LA COMPLEXITÉ, LA TERMINAISON, LA COMPLÉTUDE ET LA CORRECTION DE CXP

Cette section discute la complexité, la terminaison, la complétude et la correction de l'algorithme CXP.

4.5.1/ COMPLEXITÉ DE CXP

La boucle calculant une instance concrète de chaque état abstrait initial (de la ligne 3 à la ligne 6) itère sur l'ensemble des états abstraits A. Sa complexité est donc $\text{O}(|A|)$, où $|A|$ est le nombre d'éléments dans A qui correspond en pratique au nombre d'états abstraits non vides.

La boucle procédant au calcul d'une instance concrète de chaque *may*-transition (de la ligne 9 à la ligne 39) itère sur l'ensemble RQ. Initialement, RQ contient tous les états abstraits initiaux, c'est-à-dire au plus tous les états abstraits de A. Cette boucle ajoute des états abstraits *may*-atteignables à cet ensemble s'ils n'y ont pas encore été ajoutés. La taille maximale de RQ est donc la taille de l'ensemble des états abstraits A. Pour chaque état de RQ, l'instanciation des *may*-transitions requiert d'itérer sur l'ensemble des événements Ev (de la ligne 13 à la ligne 38) et, pour chacun de ces événements, itère de nouveau sur l'ensemble des états abstraits A (de la ligne 14 à la ligne 37).

La complexité globale de l'algorithme CXP est donc $\text{O}(|A| + |A| \times |Ev| \times |A|)$.

4.5.2/ TERMINAISON DE CXP

On considère les appels au solveur SMT comme des instructions dont la terminaison est garantie, sans quoi la terminaison de CXP ne peut elle-même pas être garantie. La coloration récursive des successeurs connus d'un état concret *vert* (ligne 29) termine puisque l'ensemble des successeurs connus de c' est un ensemble fini.

L'algorithme itère sur des ensembles dont le nombre d'éléments est fini. En effet, A contient $2^{|\mathcal{P}|}$ éléments, où $\mathcal{P}$ est l'ensemble fini des prédicats d'abstraction. De même, Ev est l'ensemble fini des événements du système. Enfin, un même état abstrait n'est jamais ajouté plus d'une fois à RQ. Ainsi, la taille de RQ varie au cours de l'instanciation des *may*-transitions, mais finit par être vide lorsque tous les états *may*-atteignables ont été explorés.

Par conséquent, la terminaison de l'algorithme CXP est garantie.

4.5.3/ COMPLÉTUDE ET CORRECTION DE CXP

L'objectif de l'algorithme est de calculer une instance (au moins) de chaque *may*-transition. Le nombre maximal de *may*-transitions pour un système événementiel $\langle X, Inv, Init, Ev\rangle$ et un ensemble d'états abstraits A donnés est égal à $|A| \times |Ev| \times |A|$. En effet, il peut exister au maximum $|A|$ états *may*-atteignables source d'une *may*-transition, à partir duquel au maximum $|Ev|$ événements sont applicables et mènent au plus à $|A|$ états abstraits différents (qui deviennent d'ailleurs *may*-atteignables).

Une notion de complétude pour CXP pourrait considérer qu'il calcule une instance atteinte de toutes les *may*-transitions concrètement atteignables du MTS. Comme cela à déjà été expliqué cependant, ce problème n'est pas décidable et on propose donc d'utiliser une notion un peu plus faible de complétude.

Pour que l'algorithme CXP soit complet, il suffirait donc d'itérer sur l'ensemble des états abstraits q de A, puis d'itérer pour chacun de ces états abstraits sur l'ensemble des événements e de Ev, puis d'itérer pour chacun de ces événements sur l'ensemble des états abstraits q' de A et d'instancier, pour chaque triplet (q, e, q'), la transition $q \xrightarrow{e} q'$, lorsqu'une instance de cette transition existe.

Cependant, si un état abstrait n'est pas *may*-atteignable, l'ensemble des *may*-transitions ne contient pas toutes les transitions abstraites pour lesquelles il existe une instance. Pour qu'une transition $q \xrightarrow{e} q'$ soit une *may*-transition, il est nécessaire que son état abstrait source q soit *may*-atteignable. Son état abstrait cible q' devient alors également nécessairement *may*-atteignable.

Pour que l'algorithme CXP soit correct, il ne doit donc instancier que les *may*-transitions, donc toutes les transitions pouvant être instanciées et dont l'état source est *may*-atteignable. Pour cela, l'ensemble RQ ne doit contenir que des états *may*-atteignables. Initialement, il ne contient que les états abstraits initiaux qui sont *may*-atteignables puisqu'il existe au moins un état concret initial dans chacun de ces états (voir la boucle de la ligne 3 à 6). Par la suite, un état abstrait q' n'est ajouté à RQ que s'il est la cible d'une transition *may*-atteignable (voir la condition de la ligne 16 qui certifie que la transition est une *may*-transition et qu'elle est *may*-atteignable et la ligne 35 dans cette condition qui ajoute la cible de cette *may*-transition à RQ).

L'algorithme CXP instancie donc toutes les *may*-transitions et uniquement les *may*-transitions.

4.6/ BILAN ET CONCLUSION

L'algorithme CXP a pour objectif de générer un ATS dans lequel tous les états abstraits *may*-atteignables et toutes les transitions abstraites *may*-atteignables du MTS sont couverts. Pour cela, il calcule au moins une instance de chacun de ces états et transitions. Une telle instance peut en pratique ne pas être atteinte depuis l'un des états concrets initiaux instanciés par CXP, ce qui signifie qu'elle n'est pas couverte dans l'ATS.

Pour palier ce problème, CXP utilise trois différentes heuristiques permettant de calculer des instances d'états et de transitions en augmentant les chances qu'elles soient d'une part connectées entre elles et d'autre part qu'elles soient atteintes depuis l'un des états concrets initiaux calculés, les rendant ainsi couvertes dans l'ATS.

La première heuristique permet au testeur d'imposer l'ordre dans lequel les événements du système événementiel seront traités lors de l'instanciation des transitions abstraites. En pratique, un ordonnancement suivant le plus possible la causalité entre les événements permet souvent une meilleure connectivité des transitions entre elles. En effet, CXP aura tendance à instancier des états concrets auxquels l'instance de transition suivante pourra être connectée si la causalité est respectée. La conception de cette fonction d'ordonnancement implique une bonne connaissance du système de la part du testeur.

La seconde heuristique permet au testeur d'imposer l'ordre dans lequel les états abstraits seront utilisés comme cible des transitions abstraites à instancier. Favoriser l'instanciation des transitions abstraites réflexives permet d'augmenter le nombre d'états concrets dans un état abstrait *may*-atteignable avant d'essayer de calculer des instances de transitions ayant cet état abstrait pour source et un autre état abstrait pour cible. Cela améliore généralement les chances que ces instances de transitions aient un état source connu et donc connecté à d'autres instances. La conception de cette fonction d'ordonnancement statique et efficace est cependant difficile à faire car celle-ci dépend de l'état abstrait courant. Une fonction configurée dynamiquement et déduite automatiquement permet généralement, comme les résultats expérimentaux le montreront dans la partie 8.3.1, d'obtenir de meilleurs taux de couverture des états et des transitions que lorsqu'elle est conçue manuellement. L'inférence d'une fonction d'ordonnancement dynamique des états abstraits cibles est réalisée par l'algorithme $\mathrm{CXP}_{\mathrm{ASO}}$ qui laisse le solveur décider de l'état abstrait cible parmi ceux atteignables par un état abstrait et un événement appliqué.

La troisième heuristique utilise une couleur attribuée à chacun des états concrets : *vert* si l'état est atteint de manière certaine et *bleu* s'il n'est pas certain que l'état soit atteint. À l'aide de cette information, CXP tente de calculer des instances de transitions abstraites dont l'état concret source est atteint (*vert*), puis des instances dont l'état concret source est atteint et dont l'état concret cible est *bleu* et peut ainsi devenir *vert*. Le calcul d'instances dont l'état source est *vert* permet d'assurer que la transition abstraite instanciée soit couverte. Cette heuristique augmente donc de manière générale le nombre de transitions abstraites couvertes (et ainsi des états abstraits couverts) dans l'ATS calculé par CXP.

L'algorithme $\mathrm{CXP}_{\mathrm{ASO}}$ n'utilise en pratique que les deux premières heuristiques, la première de la même manière que CXP et la seconde en calculant automatiquement un ordre dynamique de traitement des états abstraits. L'heuristique de coloration n'est en revanche pas réellement nécessaire car l'étape principale d'instanciation de $\mathrm{CXP}_{\mathrm{ASO}}$ ne calcule des instances de transitions abstraites qu'à partir d'états connus et atteints. Seule l'étape de complétion du MTS instancie les transitions abstraites sans assurance que ces instances soient atteintes. Afin de rester compatible avec la définition d'un ATS en revanche, la fonction κ est tout de même enrichie lors du calcul d'états concrets. Les expérimentations présentées dans la partie 8.3.1 montrent que l'algorithme $\mathrm{CXP}_{\mathrm{ASO}}$ permet généralement d'obtenir de meilleurs taux de couverture des états abstraits et des transitions abstraites que CXP.

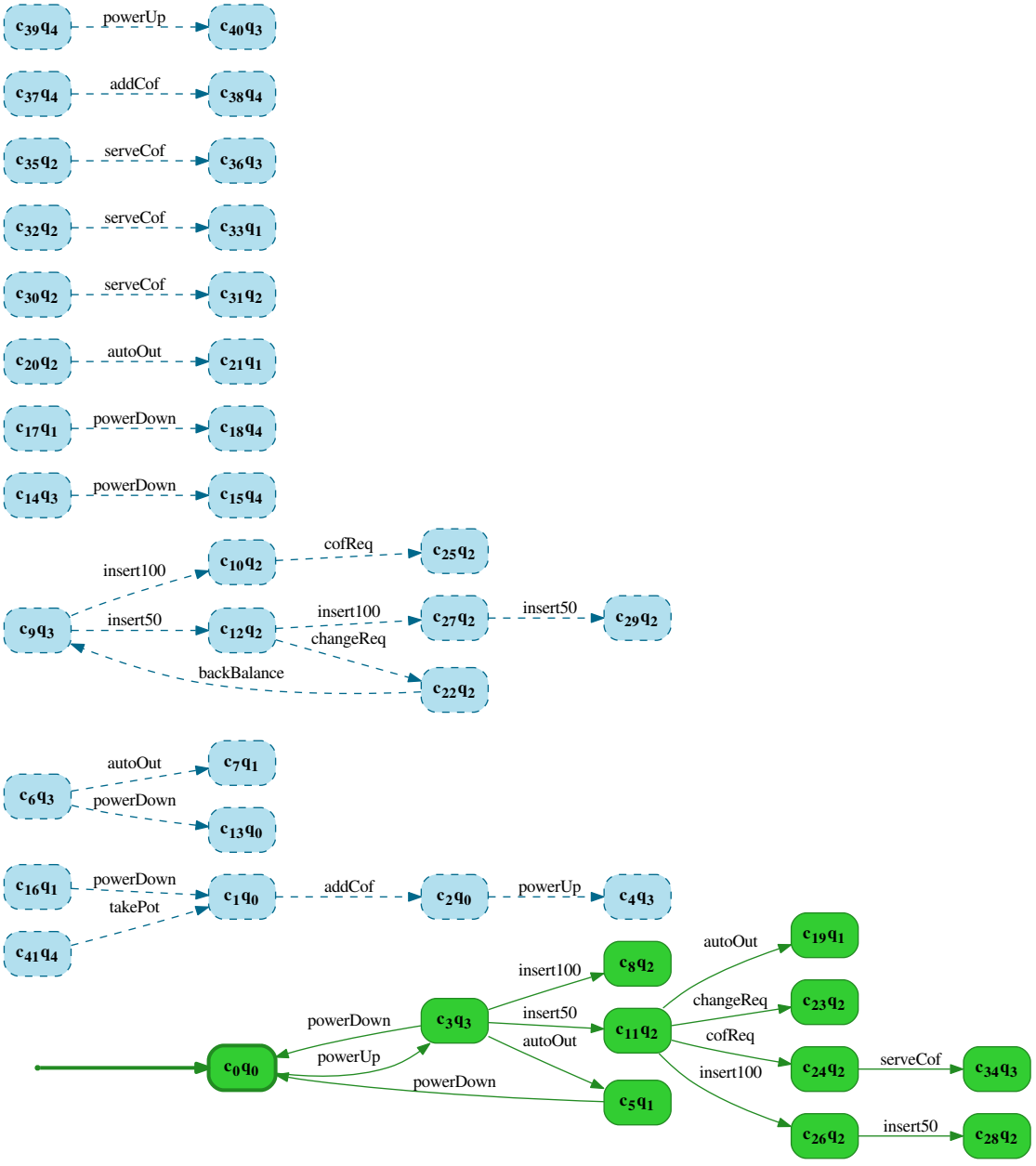


FIGURE 4.1 – CTS calculé par CXP pour l'exemple du distributeur de café

Algorithme 2 : Concrete Exploration with Automatic States Ordering ($\mathrm{CXP}_{\mathrm{ASO}}$)

Inputs : $\langle X, \mathit{Inv}, \mathit{Init}, Ev\rangle$: un système événementiel
A : un ensemble fini d'états abstraits
oEv : la liste ordonnée des événements de Ev
Output : $\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa\rangle$: un système de transitions approximé
Variables : RQ : l'ensemble des états abstraits restant à traiter
RQ' : l'ensemble des états abstraits cible tels que $q \xrightarrow{e} q' \notin \Delta$
q, q' : les états abstraits source et cible de la transition à instancier
c, c' : les états concrets calculés pour les états abstraits q et q' respectivement
RCS : l'ensemble des états concrets atteints (verts dans CXP) dans l'état abstrait q

1 $Q_0 := \emptyset$; $Q := \emptyset$; $\alpha := \mathit{emptyMap}()$; $\kappa := \mathit{emptyMap}()$; $C_0 := \emptyset$; $\Delta := \emptyset$; $\Delta^c := \emptyset$; $RCS := \emptyset$;
2 // Calcul d'une instance concrète de chaque *may*-transition
3 **foreach** $q \in A$ **do**
4 $c := \mathtt{SAT}(\mathit{prd}_X(\mathit{Init}) \wedge q[^{X'}/_X])[^X/_{X'}]$;
5 **if** $c \notin \{\mathit{unknown}, \mathit{unsat}\}$ **then** $Q_0 := Q_0 \cup \{q\}$; $C_0 := C_0 \cup \{c\}$; $\alpha(c) := q$; $\kappa(c) := \mathit{vert}$; **end** ;
6 **end**
7 // Exploration depuis les états abstraits initiaux Q_0
8 $Q := Q_0$; $C := C_0$; $RQ := Q_0$;
9 **while** $RQ \neq \emptyset$ **do**
10 **choose** $q \in RQ$;
11 $RQ := RQ - \{q\}$;
12 $Q := Q \cup \{q\}$;
13 **foreach** $e \stackrel{def}{=} a \in oEv$ **do**
14 // Calcul de l'ensemble des états q' tels que $q \xrightarrow{e} q'$ n'a pas été instanciée
15 $RQ' := \{q' \mid q' \in A \wedge q \xrightarrow{e} q' \notin \Delta\}$;
16 **do**
17 // Les états concrets source de l'instance sont les états connus dans q
18 // On note que ces états sont nécessairement atteints
19 $RCS := \{c \mid c \in C \wedge \alpha(c) = q\}$;
20 // Calcul d'un état abstrait q' qui sera atteint
21 $(q', c, c') := \mathtt{SAT}(\bigvee_{c \in RCS} c \wedge \mathit{prd}_X(a) \wedge \bigvee_{q' \in RQ'} q'[^{X'}/_X])$;
22 **if** $(q', c, c') \notin \{\mathit{unknown}, \mathit{unsat}\}$ **then**
23 // Une nouvelle transition abstraite est atteinte
24 $\Delta := \Delta \cup \{q \xrightarrow{e} q'\}$;
25 $C := C \cup \{c'\}$; $\Delta^c := \Delta^c \cup \{c \xrightarrow{e} c'\}$; $\alpha(c') := q'$; $\kappa(c') := \mathit{vert}$;
26 // q' is reachable and therefore needs to be treated
27 $RQ := RQ \cup \{q'\}$;
28 $RQ' := RQ' - \{q'\}$;
29 **end**
30 **while** $(q', c, c') \notin \{\mathit{unknown}, \mathit{unsat}\}$;
31 **end**
32 **end**
33 // Calcul des transitions *may*-atteignables non atteintes par $\mathrm{CXP}_{\mathrm{ASO}}$
34 $RQ := Q$;
35 **while** $RQ \neq \emptyset$ **do**
36 **choose** $q \in RQ$;
37 $RQ := RQ - \{q\}$;
38 $Q := Q \cup \{q\}$;
39 **foreach** $q \xrightarrow{e} q' \in (A \times \{e \mid e \stackrel{def}{=} a \in Ev\} \times A) - \Delta$ **do**
40 $(c, c') := \mathtt{SAT}(q \wedge \mathit{prd}_x(a) \wedge q'[^{X'}/_X])$;
41 **if** $(c, c') \notin \{\mathit{unsat}, \mathit{unknown}\}$ **then**
42 // $q \xrightarrow{e} q'$ is a may-transition
43 $\Delta := \Delta \cup \{q \xrightarrow{e} q'\}$;
44 $C := C \cup \{c, c'\}$; $\Delta^c := \Delta^c \cup \{c \xrightarrow{e} c'\}$; $\alpha(c) := q$; $\alpha(c') := q'$; $\kappa(c) := \mathit{bleu}$; $\kappa(c') := \mathit{bleu}$;
45 **if** $q' \notin Q$ **then** $RQ := RQ \cup \{q'\}$; **end** ;
46 **end**
47 **end**
48 **end**
49 **return** $\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa\rangle$

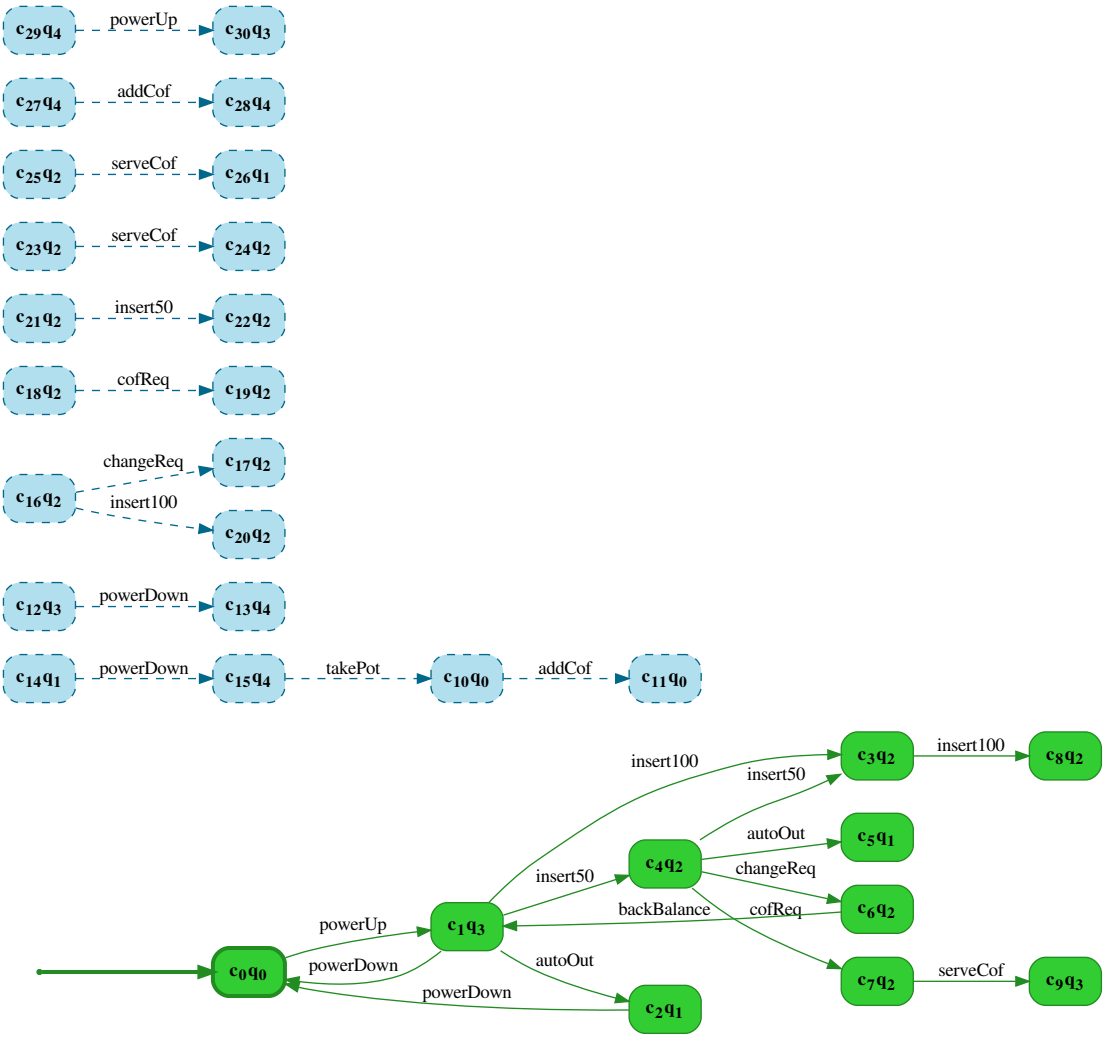


FIGURE 4.2 – CTS calculé par $\mathrm{CXP}_{\mathrm{ASO}}$ pour l'exemple du distributeur de café

5

UTILISATION DES MODALITÉS $must^+$ ET $must^-$ POUR COMPLÉTER LA COUVERTURE DES ÉTATS ET DES TRANSITIONS DE L'ABSTRACTION : ALGORITHME BCI

Ce chapitre présente une méthode d'extension d'un système de transitions approximés obtenu par l'algorithme CXP dans l'espoir d'augmenter la couverture des états abstraits et des transitions abstraites. Cette méthode tire parti du calcul des systèmes tri-modaux (voir définition 14 page 30) dont les transitions peuvent avoir jusqu'à trois modalités (may, $must^+$ et $must^-$). Elle calcule, par le biais d'un algorithme (BCI pour « *Ball Chains Instanciation* »), des chaînes de transitions abstraites (appelées chaînes de Ball) et les instancie. Les instances calculées sont de manière certaine applicables sur le système réel et connectées les unes aux autres sur toute la longueur de la chaîne.

La section 5.1 introduit les problématiques auxquelles la méthode tente d'apporter une solution et expose l'idée principale sur laquelle repose cette solution. La section 5.2 définit les notions de chaînes de Ball et de structures de $must$-transitions. La section 5.3 décrit les principes de l'algorithme BCI ainsi que ses entrées, sorties et les variables qu'il utilise. La section 5.4 formalise la partie la plus importante de l'algorithme BCI et décrit son fonctionnement concret. Enfin, la section 5.6 énonce un bilan sur la contribution et conclut ce chapitre.

5.1/ PROBLÉMATIQUE ET SOLUTION

Certains états et certaines transitions abstraites du MTS calculé par CXP ne sont pas atteints dans l'ATS. Il est par exemple possible que CXP instancie une transition abstraite par une transition concrète dont la source n'est pas atteinte. De même, il se peut qu'aucun des états concrets calculés par CXP pour un état abstrait donné ne soient atteints dans l'ATS. De tels états et transitions ne posséderaient aucune instance dans l'ATS susceptible d'être instanciée sur le système réel.

Pour tenter de palier ce problème, la contribution présentée dans ce chapitre tente d'ajou-

ter de prolonger les états atteints par des séquences de transitions concrètes garanties d'être connectées les unes aux autres. Dans [Ball, 2005], le calcul des modalités $must^-$ et $must^+$, en plus des modalités may, permet de calculer des chaînes de transitions de la forme $(must^-)^*.may.(must^+)^*$. Ces chaînes ont la propriété d'être nécessairement instanciables par des chaînes concrètes connectées.

L'hypothèse sur laquelle se base cette contribution est donc que le calcul de ces chaînes permette de calculer des instances concrètes connectées et qu'elle permettent ainsi, pour peu que leur premier état soit atteint, d'améliorer les sous-approximations et leur partie atteinte.

Nous présentons donc ces chaînes, appelées par la suite « chaînes de Ball », ainsi que des structures plus complexes composant plusieurs chaînes de Ball appelées par la suite « structures de $must$-transitions ».

5.2/ TRANSITIONS TRI-MODALES, CHAÎNES DE BALL ET STRUCTURES DE $must$-TRANSITIONS

Cette section présente les deux notions principales utilisées par l'algorithme BCI : les chaînes de Ball et les structures de $must$-transitions.

5.2.1/ DÉTERMINATION DES MODALITÉS DES TRANSITIONS ABSTRAITES

L'algorithme BCI complète le 3MTS calculé par CXP (ou CXP_{ASO}) par les instances concrètes de chaînes de Ball. L'identification des chaînes de Ball dans un 3MTS nécessite le calcul des modalités des transitions abstraites qui le composent. Pour cela, deux méthodes sont possibles :

1. Calculer la modalité des transitions à la volée, dès que CXP découvre une nouvelle may-transition (méthode la plus efficace).
2. Parcourir les transitions du MTS une fois calculé par CXP et calculer leurs modalités (méthode nécessitant de parcourir à nouveau toutes les transitions abstraites du MTS).

Pour rappel, le calcul de la modalité des transitions s'effectue par un appel à un solveur SMT à l'aide des formules logiques présentées en section 2.3 page 28.

Pour utiliser BCI, on considère, peu importe la méthode utilisée, que les modalités de toutes les transitions ont été calculées et stockées dans le 3MTS, c'est-à-dire que les ensembles de $must$-transitions Δ^- et Δ^+ ont bien été complétés.

5.2.2/ CHAÎNES DE BALL

Les chaînes de Ball sont des séquences de transitions abstraites dont les modalités suivent le schéma décrit dans la définition 20. Elles sont nommées ainsi car présentées par Thomas Ball dans son article [Ball et al., 2005].

Définition 20 : Chaîne de Ball

Une chaîne de Ball est une séquence non vide de transitions abstraites dont les modalités suivent l'expression régulière suivante :

$$(must^-)^*.may^{0/1}.(must^+)^*$$

Les chaînes de Ball sont donc des séquences finies non vides de transitions abstraites composées d'une séquence de longueur quelconque mais finie de $must^-$-transitions, suivie d'une may transition (qui peut éventuellement avoir la modalité $must^-$ et/ou $must^+$), suivie d'une séquence de longueur quelconque mais finie de $must^+$-transitions.

La chaîne de Ball montrée par la figure 5.1 est la séquence $q_0 \xrightarrow{e1^-} q_1$, $q_1 \xrightarrow{e2^-} q_2$, $q_2 \xrightarrow{e3} q_3$, $q_3 \xrightarrow{e4^+} q_4$ et $q_4 \xrightarrow{e5^+} q_5$ (voir les transitions en pointillés). Il est prouvé dans [Ball, 2005] et [Ball et al., 2005] que les instances d'une chaîne de Ball sont nécessairement instanciables sur le système réel et sont connectées les unes aux autres. Il est ainsi possible de compléter l'ATS calculé par CXP avec de nouvelles instances concrètes connectées de transitions. Dans le cas des systèmes événementiels, le flot de contrôle est en revanche implicite, contrairement aux programmes utilisés dans [Ball, 2005]. Une chaîne de Ball ne sera donc instanciable sur le système réel que si le début de la chaîne instanciant cette chaîne est atteignable. On tentera donc d'instancier les chaînes de Ball à partir d'états atteints afin de garantir que toutes les instances de transitions calculées le seront.

La figure 5.1 illustre la concrétisation d'une chaîne de Ball en gras et montre que les instances la constituant sont, de par leurs modalités, nécessairement connectées entre elles.

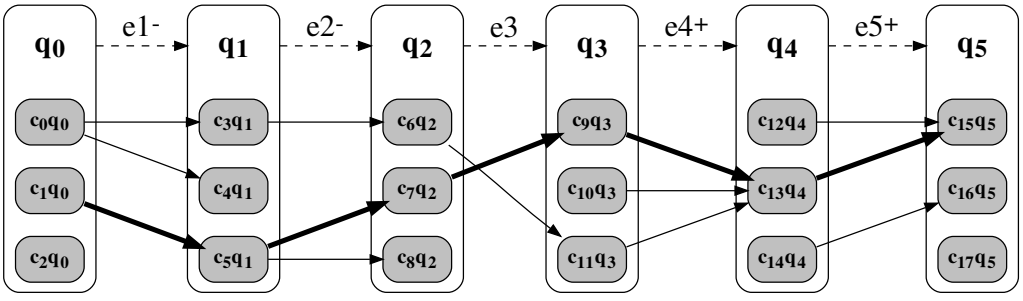
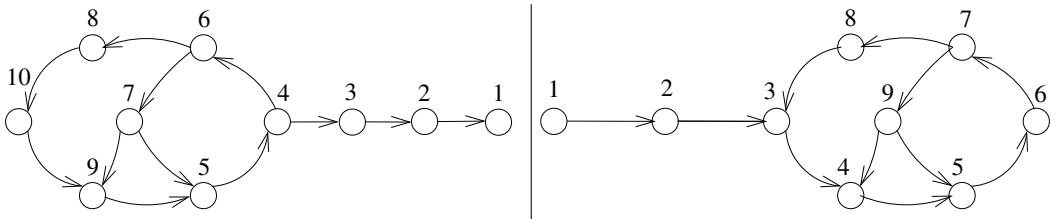


FIGURE 5.1 – Exemple d'instance de chaîne de Ball (en gras)

Tout état concret d'un état abstrait cible d'une $must^-$-transition peut être atteint depuis un état concret de l'état abstrait source de cette transition (voir les instances des transitions abstraites $q_0 \xrightarrow{e1^-} q_1$ et $q_1 \xrightarrow{e2^-} q_2$ sur la figure 5.1). Similairement, tout état concret atteint appartenant à la source d'une séquence de $must^+$-transitions permet d'instancier cette séquence depuis cet état de manière connectée (voir les instances des transitions abstraites $q_3 \xrightarrow{e4^+} q_4$ et $q_4 \xrightarrow{e5^+} q_5$ sur la figure 5.1). Il est ainsi possible d'insérer entre ces deux types de séquence une may-transition dont l'instance aura nécessairement pour source un état correspondant à la cible de la séquence de $must^-$-transitions et pour cible un état



(a) Une $must^-$-structure avec 1 comme point d'entrée ou une $must^+$-structure avec 4, 5, 6, 7, 8, 9 ou 10 comme points d'entrée

(b) Une $must^+$-structure avec 1 comme point d'entrée ou une $must^-$-structure avec 3, 4, 5, 6, 7, 8 ou 9 comme points d'entrée

FIGURE 5.2 – Exemples de $must$-structures

correspondant à la source de la séquence de $must^+$-transitions.

Grâce à ces propriétés liées aux modalités, les chaînes de Ball sont nécessairement instanciées par des transitions connectées entre elles. Cette connexion assure que si la source de l'instance d'une chaîne de Ball est atteinte, alors cette instance est nécessairement instanciable sur le système réel.

5.2.3/ STRUCTURES DE $must$-TRANSITIONS

L'algorithme BCI manipule des structures de $must$-transitions. Ces structures (dites $must$-structures) sont composées soit exclusivement de $must^-$-transitions (on qualifiera cette structure de $must^-$-structure), soit exclusivement de $must^+$-transitions (on qualifiera cette structure de $must^+$-structure).

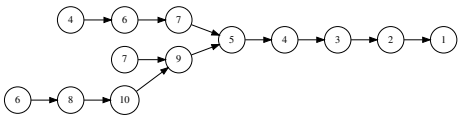
Définition 21 : $Must$-structures

Soient CFC^{mod} une composante fortement connexe possiblement vide de $must^{mod}$-transitions où $mod \in \{-,+\}$ et SEQ^{mod} une séquence possiblement vide de $must^{mod}$-transitions où $mod \in \{-,+\}$.
Une $must^{mod}$-structure est soit une CFC^{mod} suivie de et connectée à une SEQ^{mod}, soit une SEQ^{mod} suivie de et connectée à une CFC^{mod}.

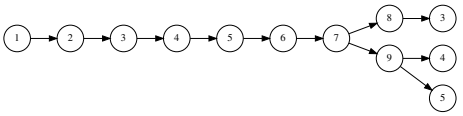
Le point d'entrée d'une $must^-$-structure est soit un état sans successeur, soit, si un tel état n'existe pas, un état quelconque dans la composante fortement connexe de la structure. Le point d'entrée d'une $must^+$-structure est soit un état sans prédécesseur soit, si un tel état n'existe pas, un état quelconque dans la composante fortement connexe de la structure.

Les figures 5.2a et 5.2b illustrent respectivement les concepts de $must^-$-structure avec point d'entrée (et de $must^+$-structure avec plusieurs points d'entrée) et de $must^+$-structure avec point d'entrée (et de $must^-$-structure avec plusieurs points d'entrée).

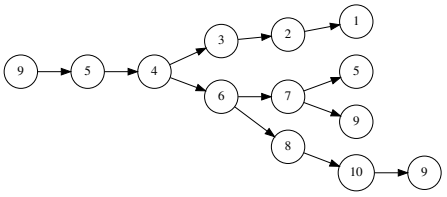
Si les transitions de la structure ont la modalité $must^-$, le point d'entrée est soit l'état abstrait 1 sur la figure 5.2a si la structure possède un état sans successeur, soit l'un des états abstraits 3, 4, 5, 6, 7, 8 ou 9 sur la figure 5.2b si la structure ne possède pas d'état sans successeur. Si les transitions de la structure ont la modalité $must^+$, le point d'entrée est soit l'état abstrait 1 sur la figure 5.2b si la structure possède un état sans prédécesseur, soit l'un des états abstraits 4, 5, 6, 7, 8, 9 ou 10 sur la figure 5.2a si la structure ne possède pas d'état sans prédécesseur.



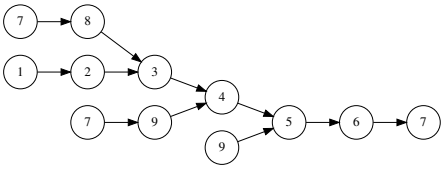
(a) Une concrétisation de la figure 5.2a vue comme une $must^-$-structure avec 1 comme point d'entrée



(b) Une concrétisation de la figure 5.2b vue comme une $must^+$-structure avec 1 comme point d'entrée



(c) Une concrétisation de la figure 5.2a vue comme une $must^+$-structure avec 9 comme point d'entrée



(d) Une concrétisation de la figure 5.2b vue comme une $must^-$-structure avec 7 comme point d'entrée

FIGURE 5.3 – Exemples d'arbres de concrétisation de $must$-structures

Un *arbre de concrétisation* d'une $must$-structure est un arbre, composé de transitions concrètes et dont la racine est le point d'entrée de la structure, dans lequel chaque transition est l'instance d'une $must$-transition de la structure.
N.B. Dans le cas d'une $must^+$-structure, les transitions de l'arbre vont de la racine aux feuilles de l'arbre et dans le cas d'une $must^-$-structure, les transitions de l'arbre vont des feuilles jusqu'à la racine.

La figure 5.3 illustre les concepts d'arbres de concrétisation de $must$-structures. Les figures 5.3a et 5.3c concrétisent la $must$-structure de la figure 5.2a et les figures 5.3b et 5.3d concrétisent la $must$-structure de la figure 5.2b.

5.3/ PRINCIPES DE L'ALGORITHME BCI

L'algorithme CXP instancie chaque may-transition au moins une fois. Il n'est par conséquent pas nécessaire que l'algorithme BCI instancie de nouveau les may-transitions qui ne sont pas des $must$-transitions. En effet, si une may-transition se trouve entre une séquence de $must^-$-transitions et une séquence de $must^+$-transitions, elle appartient à une chaîne de Ball. L'instance de cette may-transition calculée par CXP aura ainsi nécessairement pour source la cible d'une instance d'une $must^-$-transition et pour cible la source d'une instance de $must^+$-transition (voir les instances de la may-transition $q_2 \xrightarrow{e3} q_3$ sur la figure 5.1). De fait, elle sera connectée aux instances de $must^-$-transitions et aux instances de $must^+$-transitions.

L'algorithme BCI va donc se concentrer sur l'instanciation des $must^-$-transitions et des $must^+$-transitions. Plus précisément, il instancie toutes les $must^-$-transitions d'une $must^-$-structure et toutes les $must^+$-transitions d'une $must^+$-structure exactement une fois.

L'algorithme BCI opère différemment selon le type de structure :

- Dans le cas d'une $must^+$-structure, il effectue un parcours en profondeur des $must^+$-transitions de la $must^+$-structure depuis son point d'entrée. Pour chaque $must^+$-transition rencontrée, il calcule une instance dont la source est la cible de l'instance précédemment calculée. S'il s'agit de la première transition de la $must^+$-structure, l'état source de l'instance est un état choisi au hasard parmi les états $verts$ de l'état abstrait source. Si aucun état $vert$ n'existe dans l'état abstrait source, un état $bleu$ est utilisé s'il en existe un et sinon, un nouvel état est calculé.
  Sur la figure 5.1 par exemple, la première transition de la $must^+$-structure est $q_3 \xrightarrow{e4^+} q_4$ et l'instance calculée est $c_9 \xrightarrow{e4} c_{13}$. La transition suivante dans la $must^+$-structure est $q_4 \xrightarrow{e5^+} q_5$ et l'instance correspondante doit avoir pour source l'état c_{13}, il s'agit donc de la transition $c_{13} \xrightarrow{e5} c_{15}$.
- Dans le cas d'une $must^-$-structure, il effectue une exploration en arrière à partir de son point d'entrée. L'objectif de ce parcours en arrière est d'une part de calculer les plus longues chaînes de $must^-$-transitions possibles depuis le point d'entrée et d'autre part d'essayer de faire en sorte que les sources des instances soient des états atteints dans l'ATS (des états $verts$ calculés par CXP). De cette manière, les chaînes de Ball instanciées seront atteintes et permettront ainsi une meilleure couverture des transitions abstraites si l'une des transitions de cette chaîne n'était pas couverte avec CXP.

Toutes les may-transitions ayant déjà été instanciées une fois par CXP, BCI ne les instancie pas de nouveau : il n'instancie que les transitions de modalité $must$. Ainsi, BCI ne manipule pas directement les chaînes de Ball, mais plutôt les séquences de $must^-$-transitions et $must^+$-transitions qui les composent. Il procède à une exploration symbolique en arrière pour instancier les $must^-$-transitions et à une exploration en avant pour les $must^+$-transitions. L'algorithme pour les $must^+$-transitions procède à une simple exploration en profondeur des $must^+$-transitions. Pour cette raison, il n'est pas décrit ici. Afin d'instancier les plus longues chaînes de $must^-$-transitions, l'algorithme d'instanciation correspondant est en revanche moins habituel et c'est donc celui qui sera décrit en section 5.4.

5.3.1/ ENTRÉES, SORTIES ET VARIABLES DE L'ALGORITHME DE CONCRÉTISATION DES $must^-$-STRUCTURES

Cette section décrit les paramètres d'entrée et de sortie ainsi que les variables utilisées par BCI pour concrétiser les $must^-$-structures.

5.3.1.1/ ENTRÉES

Les entrées de l'algorithme sont décrites ci-dessous :

- $qr \xrightarrow{e} qr'$: il s'agit du point d'entrée de la $must^-$-structure à parcourir,
- $\langle X, Inv, Init, Ev \rangle$: il s'agit du système événementiel,
- $\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa \rangle$: il s'agit de l'ATS calculé par CXP,
- $\langle Q_0, Q, \Delta, \Delta^-, \Delta^+ \rangle$: il s'agit du MTS calculé par CXP et complété avec les ensembles Δ^- et Δ^+ contenant respectivement les $must^-$-transitions et les $must^+$-transitions, le tout formant un 3MTS.

5.3.1.2/ SORTIES

Les sorties de l'algorithme sont décrites ci-dessous :

- $\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa \rangle$: il s'agit de l'ATS d'entrée qui sera complété par les états concrets et les transitions concrètes calculés par l'algorithme BCI.

5.3.1.3/ VARIABLES

Les variables utilisées par l'algorithme sont décrites ci-dessous :

- $t \stackrel{def}{=} q \xrightarrow{e} q'$: il s'agit de la transition abstraite en cours de traitement dans l'arbre de $must^{-}$-transitions.
- wp_q : il s'agit d'un état symbolique contenant tous les états pouvant être la source de l'instance de la transition t.
- P_{chaine} : il s'agit d'une pile stockant la chaîne de $must^{-}$-transitions qui sera instanciée.
- P_{branch} : il s'agit d'une pile stockant tous les branchements rencontrés dans l'arbre correspondant à la $must^{-}$-structure explorée.
- B : il s'agit de l'ensemble des branches atteignant l'état q de t.
- M^{-} : il s'agit d'une fonction de marquage des transitions de Δ^{-} ($false$ si la transition n'a pas encore été instanciée, $true$ si la transition à été instanciée).
- V_{wp_q} : il s'agit de l'ensemble des états verts connus appartenant à l'état symbolique wp_q.

5.4/ FONCTIONNEMENT DE L'ALGORITHME D'INSTANCIATION DES $must^{-}$-STRUCTURES

Cette section présente le fonctionnement concret de l'algorithme BCI dans le cas des $must^{-}$-structures. La formalisation de BCI pour le cas des $must^{-}$-structures est donnée par l'algorithme 3. Étant donné que le MTS obtenu avec l'exemple du distributeur de café contient trop peu de $must^{-}$-structures pour illustrer clairement les propos de cette section, on propose d'appliquer l'algorithme BCI à l'arbre de $must^{-}$-transitions représenté par la figure 5.4.

Chaque transition d'un arbre correspondant à une $must^{-}$-structure est concrétisée par BCI par un parcours en profondeur. La méthode repose sur deux itérations imbriquées appliquées à chaque $must^{-}$-structure.

5.4.1/ EXPLORATION DE TOUTES LES CHAÎNES DE $must^{-}$-TRANSITIONS

La première itération, à partir de la ligne 6 de l'algorithme 3, parcourt toutes les chaînes de l'arbre correspondant à la $must^{-}$-structure en arrière à partir d'une transition racine (i.e. une transition dont la cible est l'entrée d'une $must^{-}$-structure).

Le parcours de toutes les chaînes de $must^{-}$-transitions en arrière utilise la pile de branchements P_{branch} afin de mémoriser les transitions à traiter plus tard dans l'ordre de la descente de l'arbre. La pile de branchement contiendra donc toutes les $must^{-}$-transitions non explorées et dont la cible est la racine de la transition racine courante.

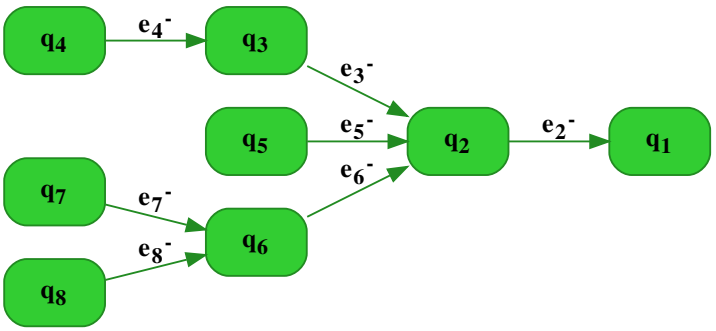


FIGURE 5.4 – Un exemple d'arbre de $must^-$-transitions avec plusieurs branches

Exemple 5 : Illustration d'un parcours en arrière des $must^-$-transitions d'un arbre

Dans la structure de la figure 5.4, on considère que la première chaîne parcourue est la chaîne $q_4 \xrightarrow{e_4^-} q_3 \xrightarrow{e_3^-} q_2 \xrightarrow{e_2^-} q_1$. Elle est parcourue en arrière, donc depuis la transition racine $q_2 \xrightarrow{e_2^-} q_1$ jusqu'à la transition $q_4 \xrightarrow{e_4^-} q_3$. La source de $q_2 \xrightarrow{e_2^-} q_1$ est la cible de plusieurs $must^-$-transitions : $q_3 \xrightarrow{e_3^-} q_2$ qui sera la transition explorée, mais également $q_5 \xrightarrow{e_5^-} q_2$ et $q_6 \xrightarrow{e_6^-} q_2$. Ces deux dernières sont donc stockées dans la pile P_{branch} et seront traitées comme des transitions racines après l'instanciation de la première chaîne.

Finalement, les chaînes parcourues et instanciées pour l'arbre de la figure 5.4 seront les suivantes :

- $q_4 \xrightarrow{e_4^-} q_3 \xrightarrow{e_3^-} q_2 \xrightarrow{e_2^-} q_1$
- $q_5 \xrightarrow{e_5^-} q_2 \xrightarrow{e_2^-} q_1$
- $q_7 \xrightarrow{e_7^-} q_6 \xrightarrow{e_6^-} q_2 \xrightarrow{e_2^-} q_1$
- $q_8 \xrightarrow{e_8^-} q_6 \xrightarrow{e_6^-} q_2 \xrightarrow{e_2^-} q_1$

N.B. Chaque transition de ces chaînes n'est instanciée qu'une seule fois. Par exemple, $q_2 \xrightarrow{e_2^-} q_1$ apparaît dans les quatre chaînes à parcourir mais n'est instanciée qu'une seule fois.

5.4.2/ EXPLORATION SYMBOLIQUE EN ARRIÈRE DES $must^-$-TRANSITIONS D'UNE CHAÎNE

La seconde itération, à partir de la ligne 7 de l'algorithme 3, parcourt toutes les transitions de la chaîne depuis la transition racine courante (la transition $t \overset{def}{=} q \xrightarrow{e} q'$ dans l'algorithme) de manière symbolique et en arrière.

Pour cela, l'algorithme BCI calcule l'ensemble B des $must^-$-transitions n'ayant pas déjà été explorée et dont l'état cible est cette fois l'état source de t (ligne 8). Si B contient des transitions (ligne 9), cela signifie que la chaîne peut être prolongée. L'une d'elle est choisie comme la nouvelle transition $t \overset{def}{=} q \xrightarrow{e} q'$ où q' est donc l'ancien état q de t (ligne 10).

L'algorithme BCI calcule ensuite l'état symbolique wp_q, défini comme le plus grand ensemble d'états concrets de l'état source q de $t \stackrel{def}{=} q \xrightarrow{e} q'$ (même ceux actuellement non présents dans le CTS) menant à sa cible q' par l'événement e (ligne 13). La paire constituée de la transition t et de l'état symbolique wp_q sont enfin ajoutés à la pile P_{chaine} mémorisant les transitions de la chaîne qui devront être instanciées une fois le parcours en arrière terminé.

On note que si un état concret *vert* appartenant à l'état symbolique wp_q est connu dans le CTS, alors toute instance de la transition $q \xrightarrow{e} q'$ appliquant e à cet état serait connectée à la partie atteinte du CTS et donc permettrait de couvrir cette transition abstraite. Cependant, l'objectif étant d'instancier des chaînes de $must^{-}$-transitions aussi longues que possible, l'instanciation n'est effectuée qu'une fois le parcours en arrière terminé.

Si B contient d'autres transitions que la transition t (ligne 15), il s'agit de transitions constituant un branchement et elles sont donc toutes ajoutées à la pile de branchements P_{branch} (ligne 17).

Le parcours de la chaîne de l'arbre en arrière se poursuit tant que B n'est pas vide, donc tant que t contient un prédécesseur dans cette chaîne. Lorsque B ne contient aucune transition, cela signifie que la transition t ne contient pas de prédécesseur non parcouru, ou pas de prédécesseur dans la chaîne explorée. L'instanciation de cette chaîne peut donc débuter car il s'agit du début de la chaîne.

Exemple 6 : Illustration de l'exploration symbolique en arrière des transitions d'une chaîne de $must^{-}$-transitions

Sur la figure 5.4, la transition racine est $q_2 \xrightarrow{e_2^-} q_1$ qui possède comme prédécesseurs les transitions $q_3 \xrightarrow{e_3^-} q_2$, $q_5 \xrightarrow{e_5^-} q_2$ et $q_6 \xrightarrow{e_6^-} q_2$. Supposons que la transition choisie pour poursuivre la chaîne soit la transition $q_3 \xrightarrow{e_3^-} q_2$. Elle est donc ajoutée à la pile P_{chaine} contenant le chemin parcouru depuis la racine. Les transitions $q_5 \xrightarrow{e_5^-} q_2$ et $q_6 \xrightarrow{e_6^-} q_2$ constituent quant à elles des branchements non explorés et sont donc toutes deux empilées dans P_{branch} sous la forme d'une paire contenant également la chaîne parcourue jusqu'alors (la pile P_{chaine}). La transition $q_3 \xrightarrow{e_3^-} q_2$ ne possède qu'un prédécesseur : la transition $q_4 \xrightarrow{e_4^-} q_3$ qui est donc ajoutée à P_{chaine} et constitue le début de la chaîne parcourue. L'instanciation à venir débutera donc par la transition $q_4 \xrightarrow{e_4^-} q_3$.

5.4.3/ INSTANCIATION EN AVANT DES TRANSITIONS DE LA CHAÎNE

L'algorithme BCI sélectionne alors un état c de wp_q, si possible *vert* dans le CTS d'entrée, n'importe lequel s'il n'existe pas dans le CTS (auquel cas il est colorié en *bleu*) ou s'il n'est pas *vert* (lignes 22 à 27). Cet état constituera la source de l'instance concrète de la transition de début de chaîne. Le dépilage de l'élément au sommet de P_{chaine} (ligne 29) permet d'obtenir une paire constituée de :

- La transition $q \xrightarrow{e} q'$ au début de la chaîne : elle fait partie de la chaîne et constitue son début et a donc été ajoutée en dernier à P_{chaine} (sur la figure 5.4, il s'agit de $q_4 \xrightarrow{e_4^-} q_3$).
- Un état symbolique wp_q correspondant au plus grand ensemble d'états concrets qui pourront servir de source à l'instance de la transition suivante de la chaîne (sur

la figure 5.4, il s'agit de la transition $q_3 \xrightarrow{e_3^-} q_2$).

Ainsi, afin de concrétiser la transition $q \xrightarrow{e} q'$, on applique l'événement e à c en s'assurant que l'état concret d'arrivée c' soit dans l'état wp_q qui vient d'être dépilé (ligne 30). Cela permet de garantir que c' pourra être utilisé comme source de l'instance de la transition suivante ($q_3 \xrightarrow{e_3^-} q_2$).

Dans l'ATS, la fonction d'abstraction α associe l'état abstrait q' à l'état concret c', la transition $c \xrightarrow{e} c'$ est ajoutée à l'ensemble de transitions concrètes Δ^c et c' prend la couleur de c dans κ (ligne 31).

Enfin, l'état concret c' servira d'instance concrète source de la transition suivante qui sera instanciée dans la boucle d'instanciation de la ligne 28. L'état c est donc affecté à c' et l'instanciation de la transition suivante dans la chaîne (s'il en existe une) est effectuée en suivant le même processus que décrit précédemment.

Un fois que P_{chaine} est vide, toutes les transitions de la chaîne ont été concrétisées. Il est cependant possible que des branchements aient été rencontrés durant l'exploration symbolique en arrière. Il reste donc a concrétiser toutes les chaînes de tous les branchements restants (mémorisés dans la pile P_{branch}).

5.4.4/ EXPLORATION DES CHAÎNES DE BRANCHEMENT NON EXPLORÉES

Une fois une chaîne concrétisée, si P_{branch} n'est pas vide (ligne 34), cela signifie que des branchements ont été rencontrés lors de l'exploration symbolique en arrière. Le dépilage de l'élément au sommet de P_{branch} (ligne 35) permet d'obtenir une paire constituée de :

- L'ensemble B des transitions qui constitueront un nouveau début de chaîne. Sur la figure 5.4, après avoir concrétisé la première chaîne, cet ensemble contient les transitions $q_5 \xrightarrow{e_5^-} q_2$ et $q_6 \xrightarrow{e_6^-} q_2$).
- La pile P_{chaine} qui contient la séquence de transitions succédant aux transitions de B. Sur la figure 5.4, après avoir concrétisé la première chaîne, la pile P_{chaine} ne contient que la transition $q_2 \xrightarrow{e_2^-} q_1$ puisque $q_5 \xrightarrow{e_5^-} q_2$ et $q_6 \xrightarrow{e_6^-} q_2$ sont les branchements de cette transition.

L'algorithme BCI choisit une transition t de B comme nouvelle transition racine de la chaîne qui sera instanciée par l'exploration symbolique en arrière suivante. La paire qui vient d'être dépilée de P_{branch} est de nouveau empilée dans P_{branch} (ligne 38) après avoir retiré la transition t de l'ensemble B (afin que la (ou les) chaîne(s) dont t est la racine ne soi(en)t pas explorée(s) plusieurs fois).

Enfin, la transition t est ajoutée à la pile P_{chaine} car elle fait partie de la nouvelle chaîne qui sera explorée (ligne 40). La phase d'exploration symbolique en arrière est alors répétée depuis la transition t et ainsi de suite jusqu'à ce que toutes les transitions de tous les chemins de l'arbre aient été instanciées.

Exemple 7 : Ordre d'instanciation des chaînes de branchement

Sur la figure 5.4, si la transition t choisie dans B est la transition $q_6 \xrightarrow{e_6^-} q_2$, elle est retirée de B et ce dernier ne contiendra plus que la transition $q_5 \xrightarrow{e_5^-} q_2$. La chaîne dont $q_5 \xrightarrow{e_5^-} q_2$ est la racine (et l'unique transition) ne sera instanciée qu'après que les chaînes dont $q_6 \xrightarrow{e_6^-} q_2$

est la racine l'auront été. Les chaînes $q_7 \xrightarrow{e_7^-} q_6 \xrightarrow{e_6^-} q_2 \xrightarrow{e_2^-} q_1$ et $q_8 \xrightarrow{e_8^-} q_6 \xrightarrow{e_6^-} q_2 \xrightarrow{e_2^-} q_1$ seront donc instanciées avant la chaîne $q_5 \xrightarrow{e_5^-} q_2 \xrightarrow{e_2^-} q_1$.

5.5/ DISCUSSIONS SUR LA COMPLEXITÉ, LA TERMINAISON, LA COMPLÉTUDE ET LA CORRECTION DE BCI

Cette section discute la complexité, la terminaison, la complétude et la correction de l'algorithme BCI.

5.5.1/ COMPLEXITÉ DE BCI

La complexité de l'algorithme BCI est égale à la somme de la complexité de l'algorithme de concrétisation des $must^+$-structures avec la complexité de l'algorithme de concrétisation des $must^-$-structures.

L'algorithme de concrétisation des $must^+$-structures correspond à un algorithme de parcours en profondeur des $must^+$-transitions. Sa complexité est donc $O(|\Delta^+|)$, avec $|\Delta^+|$ le nombre de transitions dans l'ensemble des $must^+$-transitions Δ^+.

L'algorithme de concrétisation des $must^-$-structures réalise un parcours en arrière des $must^-$-transitions, puis à un parcours en avant de ces $must^-$-transitions. La complexité est donc $O(2 \times |\Delta^-|) = O(|\Delta^-|)$

5.5.2/ TERMINAISON DE BCI

Dans le cas de l'instanciation des $must^+$-transitions, l'algorithme correspond à un parcours en avant des transitions. Cet algorithme termine donc lorsque toutes les $must^+$-transitions ont été instanciées. On discute donc plutôt la terminaison de l'algorithme d'instanciation des $must^-$-transitions (l'algorithme 3).

La boucle de la ligne 7 à 20 marque toutes les transitions d'une chaîne de $must^-$-transitions à partir de la fin de cette chaîne. Elle termine lorsque B, qui contient toutes les transitions non marquées dont la cible est la source de la dernière transition marquée ($t \overset{def}{=} q \xrightarrow{e} q'$), devient vide. Autrement dit, la boucle termine lorsqu'une transition marquant le début d'une chaîne à été marquée par la ligne 12. En effet, si cette transition est le début d'une chaîne, cela signifie qu'elle ne possède aucune transition prédécessrice ou que toutes ses transitions prédécessrices ont été marquées, et donc la construction de l'ensemble B à la ligne 8 correspond à l'ensemble vide.

La boucle de la ligne 28 à 33 instancie toutes les transitions de la chaîne courante. Elle termine lorsque chaque tuple de P_{chaine} a été enlevé de la pile. Étant donné qu'à chaque tour de boucle la ligne 29 enlève un élément de la pile P_{chaine}, la boucle est effectuée exactement autant de fois qu'il existe d'éléments dans P_{chaine} et termine donc nécessairement.

La boucle de la ligne 6 à 42 effectue les étapes précédentes pour chaque branchement non marqué, c'est-à-dire pour chaque transition non marquée correspondant à la dernière transition d'une chaîne de $must^-$-transitions. Chacune de ces transitions sera donc au

final marquée par la ligne 12. Ainsi, tous les branchements finiront par être marqués et B finira par être vide, ce qui correspond à la condition d'arrêt de la boucle et garantit donc la terminaison de l'algorithme.

5.5.3/ COMPLÉTUDE ET CORRECTION DE BCI

L'algorithme BCI est complet et correct s'il instancie toutes les $must^+$-transitions et toutes les $must^-$-transitions et uniquement ces $must$-transitions exactement une fois. Les seules transitions échappant à cette règle sont les $must^\#$-transitions qui sont instanciées deux fois : une fois comme $must^-$-transition et une fois comme $must^+$-transition. Plus précisément, il est complet et correct s'il instancie exactement une fois toutes les transitions composant toutes les chaînes de toutes les $must^+$-structures et de toutes les $must^-$-structures. En effet, toutes les $must^-$-transitions sont contenues exactement une fois dans l'une des $must^-$-structures et toutes les $must^+$-transitions sont contenues exactement une fois dans l'une des $must^+$-structures.

Le parcours et l'instanciation des $must^+$-transitions d'une $must^+$-structure sont effectués sur chaque $must^+$-structure, ainsi toutes les $must^+$-transitions sont garanties d'être instanciées.

Le parcours en arrière des $must^-$-transitions est également effectué sur chaque $must^-$-structure. Pour que l'algorithme 3 soit complet, il faut donc qu'il instancie chaque transition de la $must^-$-structure. Pour cela, l'algorithme 3 explore une chaîne de $must^-$-transitions de la $must^-$-structure et enregistre tous les branchements rencontrés (ligne 17), c'est-à-dire les $must^-$-transitions n'appartenant pas à cette chaîne mais appartenant à la $must^-$-structure. Après avoir instancié toutes les transitions de la chaîne (boucle de la ligne 28 à 33), il instancie les chaînes dont les branchements précédemment enregistrées sont la racine (la transition sans successeur non marqué, donc sans successeur non instancié).

Chaque transition sur le point d'être instanciée est marquée (par la ligne 12) afin de ne pas instancier plusieurs fois la même transition. Si ces transitions n'était pas marquées d'ailleurs, l'algorithme ne terminerait pas.

5.6/ BILAN ET CONCLUSION

L'algorithme BCI instancie d'abord les structures de $must^-$-transitions à l'aide, dans un premier temps, d'une exploration symbolique en arrière dans l'objectif de pouvoir concrétiser les plus longues chaînes de $must^-$-transitions possible. Une fois cette étape réalisée, il calcule des instances des transitions abstraites composant ces chaînes en partant, si possible, d'un état connu et $vert$ (donc atteint). Si le début d'une chaîne de $must^-$-transitions est $vert$, alors toute les transitions de la séquence instanciant cette chaîne seront atteintes. Si la chaîne contient des transitions abstraites qui n'étaient pas couvertes dans l'ATS d'entrée, alors ces transitions abstraites seront couvertes dans l'ATS complété. S'il n'existe pas d'état vert pouvant être source de la transition concrète instanciant la première transition de la chaîne, l'état concret de départ est choisi au hasard parmi les candidats possibles. Avant d'instancier les transitions suivantes de la chaîne, on pourrait alors de nouveau vérifier s'il existe un état $vert$ pouvant servir de source. On sacrifierait donc le début de la chaîne, qu'il n'est pas possible d'atteindre avec l'ATS obtenu

en entrée puisque les instances ne partent pas d'un état $vert$, au profit d'une séquence plus petite de transitions mais qui seront atteintes de manière certaine.

Pour concrétiser les $must^+$-structures, l'algorithme effectue un simple parcours en profondeur des $must^+$-transitions composant ces structures. Afin de garantir que les instances soient atteintes dans le CTS complété par BCI, cette exploration en profondeur est effectuée depuis un état $vert$ si possible, similairement au procédé utilisé pour l'instanciation des $must^-$-transitions.

Les performances de l'algorithme (en termes de temps de calcul) dépendent fortement du nombre de $must$-structures composant le MTS et du nombre de branchements dans chacune des $must^-$-structures puisque l'exploration symbolique en arrière est coûteuse.

Les résultats expérimentaux que nous avons obtenus montreront que peu de $must$-structures apparaissent généralement et que ces $must$-structures sont généralement composées de très peu de transitions. Si cela permet généralement d'appliquer BCI sans que cela ne soit rédhibitoire en termes de temps de calcul, cela présente en revanche l'inconvénient de n'instancier des transitions abstraites auparavant non couvertes que très rarement.

Algorithme 3 : ATS Calcul et instanciation des chaînes de $must^-$-transitions par exploration symbolique arrière et concrétisation en avant

Inputs : $q \xrightarrow{e} q'$: la dernière transition de la $must^-$-structure (q' est le point d'entrée)
$\langle X, Inv, Init, Ev\rangle$: le système événementiel
$\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa\rangle$: l'ATS calculé par CXP
$\langle Q_0, Q, \Delta, \Delta^-, \Delta^+\rangle$: le 3MTS calculé par CXP

Output : $\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa\rangle$: l'ATS complété par les instances de chaînes de Ball

Variables : $t \stackrel{def}{=} q \xrightarrow{e} q'$: la transition actuelle dans l'arbre de $must^-$-transitions
wp_q : $q \wedge wp(e, q')$
P_{chaine} : chemin parcouru, stocké comme une pile, depuis la racine de la $must^-$-structure
P_{branch} : pile des branchements restants à traiter
B : ensemble des branches atteignant l'état q actuel
M^- : $\Delta^- \to \{false, true\}$: fonction de marquage des transitions de Δ^-
V_{wp_q} : l'ensemble des états $verts$ dans l'état symbolique wp_q

1 $P_{branch} := \emptyset$; $P_{chaine} := \emptyset$; $t := q \xrightarrow{e} q'$;
2 $wp_q := q'$;
3 $B := \{q \xrightarrow{e} q'\}$;
4 $M^- := \Delta^- \times \{false\}$;
5 **push** $(P_{chaine}, \langle t, wp_q\rangle)$;
6 **while** $B \neq \emptyset$ **do** // Parcours des arbres dont les racines sont dans B
7 **while** $B \neq \emptyset$ **do** // Calcul des branches non marquées atteignant q
8 $B := \{s \xrightarrow{f} q \mid \exists(s, f).(s \xrightarrow{f} q \in \Delta^- \wedge \neg M^-(s \xrightarrow{f} q)\}$;
9 **if** $B \neq \emptyset$ **then** // Il existe au moins une branche dans B
10 **choose** $t \stackrel{def}{=} q \xrightarrow{e} q' \in B$;
11 // La branche t sera traitée et est donc marquée
12 $M^-(t) := true$;
13 $wp_q := q \wedge wp(e, q')$;
14 **push** $(P_{chaine}, \langle t, wp_q\rangle)$;
15 **if** $B - \{t\} \neq \emptyset$ **then** // Il existe au moins deux branches dans B
16 // Sauvegarde des branches restantes à parcourir
17 **push** $(P_{branch}, \langle B - \{t\}, P_{chaine}\rangle)$;
18 **end**
19 **end**
20 **end**
21 // Sélection d'un état c dans wp_q, vert si possible
22 $V_{wp_q} := \{c \mid c \in wp_q \wedge \kappa(c) = vert\}$;
23 **if** $V_{wp_q} = \emptyset$ **then**
24 **choose** $c \in V_{wp_q}$;
25 **else**
26 **choose** $c \in wp_q$;
27 **end**
28 **while** $\neg empty\_stack?(P_{chaine})$ **do** // Il existe une transition a instancier dans la chaîne
29 $\langle (q, e, q'), wp_q\rangle := \mathbf{pop}(P_{chaine})$;
30 $c' := \mathtt{SAT}(c \wedge prd_X(e) \wedge wp_q[^{X'}/_X])$;
31 $\alpha(c') := q'$; $\Delta^c := \Delta^c \cup \{c \xrightarrow{e} c'\}$; $\kappa(c') := \kappa(c)$;
32 $c := c'$;
33 **end**
34 **if** $\neg empty\_stack?(P_{branch})$ **then** // Retour au prochain chemin de la structure à parcourir
35 $\langle B, P_{chaine}\rangle := \mathbf{pop}(P_{branch})$;
36 **choose** t in B ; $wp_q := q \wedge wp(e, wp_q)$;
37 **if** $B - \{t\} \neq \emptyset$ **then**
38 **push** $(P_{branch}, \langle B - \{t\}, P_{chaine}\rangle)$;
39 **end**
40 $B := \{t\}$; **push**$(P_{chaine}, \langle t, wp_q\rangle)$;
41 **end**
42 **end**

6

AMÉLIORATION D'UNE SOUS-APPROXIMATION GUIDÉE PAR UN PRÉDICAT DE PERTINENCE : ALGORITHME RCXP

Ce chapitre décrit les algorithmes RCXP (pour « Relevant Concrete eXPloration ») et RCXP_{ASO} (pour « Relevant Concrete eXPloration with Automatic States Ordering ») en même temps. En effet, RCXP_{ASO} est exactement le même algorithme que RCXP ; la seule différence étant que RCXP prend en entrée un ATS provenant de l'algorithme CXP et que RCXP_{ASO} prend en entrée un ATS provenant de CXP_{ASO}. Pour le reste de cette section, CXP désigne indifféremment l'algorithme CXP et l'algorithme CXP_{ASO}. De même, RCXP désigne à la fois l'algorithme RCXP et sa version RCXP_{ASO}.

La section 6.1 expose les lacunes potentielles de l'algorithme CXP que RCXP cherche à corriger. La section 6.2 présente les prédicats de pertinence qui constituent le concept principal sur lequel repose RCXP. La section 6.3 présente la contribution développée dans ce chapitre et évoque ses principes généraux de fonctionnement. Les principes de fonctionnement de cette contribution son présentés en section 6.3.1. Les sections 6.3.2 et 6.3.3 décrivent respectivement les entrées, sorties et variables utilisées par l'algorithme RCXP et son fonctionnement général. La section 6.4 applique l'algorithme RCXP sur l'exemple du distributeur de café. La section 6.5 discute la complexité, la terminaison, la complétude et la correction de l'algorithme. Enfin, la section 6.6 dresse un bilan de cette contribution et conclut ce chapitre.

6.1/ LIMITATION DE CXP ET OBJECTIFS DE L'ALGORITHME RCXP

L'ATS calculé par CXP peut ne pas couvrir l'ensemble des états et/ou des transitions abstraites du MTS d'un système. Souvent, il s'agit d'états ou de transitions qui requièrent l'application d'une même séquence de transitions plusieurs fois pour pouvoir être atteint. CXP ne vise à couvrir qu'une seule fois chaque transition abstraite par souci d'efficacité, raison pour laquelle les transitions et états nécessitant l'application répétée de séquences de transitions ne seront pas couverts par cet algorithme.

Par exemple, dans le cas du distributeur de café, couvrir une transition menant d'un état

où il reste du café à un état où il n'en reste plus nécessite que tous les cafés en stock aient été vendus. La séquence d'opérations qui simule la vente d'un café (l'insertion de monnaie, la demande d'un café et la délivrance d'un café) devra donc être déclenchée autant de fois qu'il y avait de cafés en stock dans le distributeur à l'état initial.

L'algorithme CXP, puisqu'il ne cherche à couvrir qu'une fois chaque transition, n'est donc pas adapté pour ce type de problème et ne peut donc pas couvrir la transition réalisant la vente du dernier café à partir d'un état atteint.

L'algorithme RCXP a pour objectif de permettre au testeur de couvrir les états et transitions abstraites jugées atteignables qui n'auraient pas été couverts par l'ATS calculé par CXP, mais qu'il souhaite malgré tout voir couvertes. Il repose sur l'idée de faire progresser le système vers le déclenchement de ces transitions.

RCXP procède ainsi à une nouvelle phase d'exploration à partir des états concrets de l'ATS calculé en ciblant certaines transitions. Ce ciblage de transitions permet de maîtriser la taille de l'espace d'états calculé par RCXP. Il est réalisé par le biais d'un moyen de pilotage nommé *prédicat de pertinence* qui, en plus de guider et restreindre l'exploration, permet l'extraction d'un variant (comme nous le verrons dans la section 6.2.3) assurant la finitude de cette exploration.

6.2/ PRÉDICATS DE PERTINENCE

Un prédicat de pertinence est conçu par le testeur et fait appel à sa connaissance du système. Il permet de guider l'exploration réalisée par l'algorithme RCXP afin que ce dernier concrétise tous les états et les transitions qui doivent l'être pour couvrir un ensemble d'états et de transitions abstraites que le testeur souhaite couvrir.

Un prédicat de pertinence prend la forme d'un prédicat avant-après portant sur une ou plusieurs variables d'état du système événementiel pour lequel il est conçu. Pour qu'une transition abstraite $q \xrightarrow{e} q'$ soit instanciée par RCXP à partir de l'état concret c appartenant à q, il est nécessaire qu'il existe un état cible c' appartenant à q' atteint par l'événement $e \stackrel{def}{=} a$ et que le couple (c, c') satisfasse le prédicat de pertinence $pert_X$. Ainsi, étant donné $pert_X$ un prédicat de pertinence, la transition ne sera instanciée par $c \xrightarrow{e} c'$ que si $\mathtt{SAT}(c \wedge prd_X(e) \wedge pert_X) = sat$ est vraie.

Pour cela, le testeur doit savoir comment les variables du système vont évoluer en appliquant l'événement correspondant à une transition pertinente. Par exemple, si l'événement incrémente une variable x, le prédicat avant-après devra être évalué à $vrai$ lorsque la valeur de x croît. Dans ce cas, le prédicat avant-après $pert_X$ pourrait être exprimé ainsi : $x' > x$ avec x et x' dénotant respectivement l'état de la variable x *avant* puis *après* application de l'événement.

6.2.1/ MÉTHODE DE CONCEPTION D'UN PRÉDICAT DE PERTINENCE

La méthode de conception d'un prédicat de pertinence décrite dans cette section fonctionne en quatre étapes successives :

1. Déterminer les états abstraits et les transitions abstraites non couvertes par CXP et sélectionner un sous-ensemble d'entre-elles comme transitions cibles à couvrir.

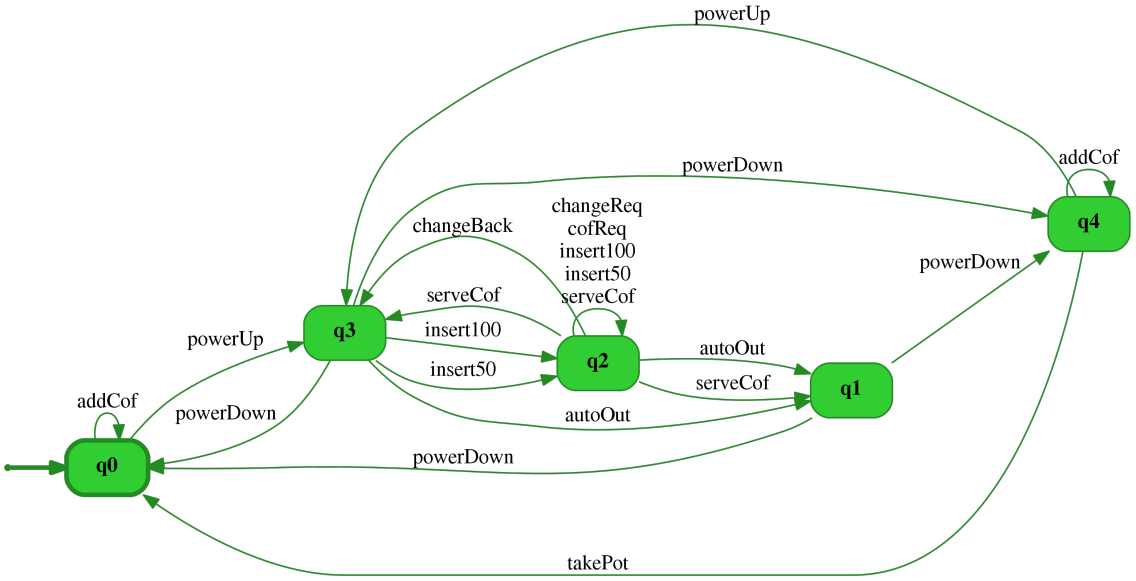


FIGURE 6.1 – MTS calculé par CXP pour l'exemple du distributeur de café

2. Identifier, à l'aide du MTS calculé par CXP, les cycles en amont des transitions cibles choisies afin de déterminer les transitions à instancier plusieurs fois lors de l'exploration.
3. Énumérer un sous-ensemble d'événements activés dans chaque cycle et suffisant pour permettre la répétition menant à la transition cible (plus ce sous-ensemble est petit, plus l'exploration sera efficace).
4. Exprimer le prédicat de pertinence comme la disjonction des prédicats avant-après qui caractérisent la substitution des événements du sous-ensemble précédemment défini.

Afin d'illustrer l'application de la méthode de conception d'un prédicat de relevance, on propose de la dérouler sur l'exemple du distributeur de cafés avec l'ensemble de prédicats d'abstraction $\mathcal{P} = \{p_0, p_1, p_2\}$ utilisé tout au long de ce mémoire.

Après avoir exécuté l'algorithme CXP, le testeur obtient le MTS montré par la figure 6.1. L'ATS calculé est celui montré précédemment par la figure 4.1, page 74.

Les étapes de conception du prédicat de pertinence sont alors les suivantes :

1. Le testeur détermine les états abstraits et les transitions non couvertes par l'ATS calculé par CXP. Ici, il s'agit de l'état q_4 et des neuf transitions suivantes :

   - $q_0 \xrightarrow{addCof} q_0$
   - $q_1 \xrightarrow{powerDown} q_4$
   - $q_2 \xrightarrow{changeBack} q_3$
   - $q_2 \xrightarrow{serveCof} q_1$
   - $q_2 \xrightarrow{serveCof} q_2$
   - $q_3 \xrightarrow{powerDown} q_4$
   - $q_4 \xrightarrow{addCof} q_4$
   - $q_4 \xrightarrow{powerUp} q_3$
   - $q_4 \xrightarrow{takePot} q_0$

   Il sélectionne ensuite un sous-ensemble d'entre-elles qu'il souhaite voir couvertes. Ici, on supposera que le testeur souhaite couvrir la transition $q_2 \xrightarrow{serveCof} q_1$ réalisant le service du dernier café restant dans le distributeur.

2. Le testeur détermine que servir le dernier café requiert auparavant le service de tous les autres cafés. À l'aide du MTS fourni par CXP (figure 6.1), il constate que le service répété de cafés requiert de boucler entre les circuits q_3, q_2, q_3 et q_2, q_2, par exemple par les transitions abstraites $q_3 \xrightarrow{insert50} q_2$, $q_2 \xrightarrow{cofReq} q_2$ et $q_2 \xrightarrow{serveCof} q_3$.
3. Les événements correspondants à ces transitions abstraites sont *insert50*, *cofReq* et *serveCof* (il est également possible d'utiliser *insert100* au lieu de *insert50*). On remarque que les événements *changeBack* de q_2 à q_3 et *changeReq* de q_2 à q_2 ne sont pas utiles ici pour que le distributeur délivre un café. Au contraire, les autoriser permettrait de faire un cycle sans servir de café, donc qui ne ferait pas progresser vers l'objectif.
4. Le prédicat de pertinence correspond donc à la disjonction des prédicats avant-après caractérisant les substitutions des quatre événements utiles identifiés *insert50*, *insert100*, *cofReq* et *serveCof*. Ces prédicats doivent donc permettre l'augmentation de la balance par les opérations *insert50* et *insert100*, la prise en compte d'une demande de café par l'opération *cofReq* et la diminution du nombre de cafés dans le distributeur par l'opération *serveCof* :

$$\underbrace{(Balance' > Balance)}_{insert50,\ insert100} \vee \underbrace{(AskCof = false \wedge AskCof' = true)}_{cofReq} \vee \underbrace{(CofLeft' < CofLeft)}_{serveCof}$$

6.2.2/ LANGAGE DE DESCRIPTION DES PRÉDICATS DE PERTINENCE

La suite de règles 1 permet, selon les besoins du testeur, d'exprimer un prédicat de pertinence à partir duquel l'algorithme RCXP pourra être appliqué.

Règles 1 : Langage d'expression d'un prédicat de pertinence

Soient x une variable d'état du système sur lequel on souhaite définir un prédicat de pertinence, v et v' des valeurs possibles de la variable d'état x et $b_1, ..., b_m$ des prédicats quelconques sur les variables d'état du système et tels qu'il existe un unique $i \in 1..m$ tel que b_i soit vrai.
Le langage d'expression d'un prédicat de pertinence $pert_X$ est défini par les règles suivantes :

$$pert_X ::= pr_1 \vee \ldots \vee pr_n \tag{6.1}$$
$$pr ::= pa \mid pc \tag{6.2}$$
$$pa ::= x' < x \mid x' > x \mid x = v \ \wedge \ x' = v' \tag{6.3}$$
$$pc ::= b_1 \Rightarrow pa_1 \wedge \ldots \wedge b_m \Rightarrow pa_m \tag{6.4}$$

La règle 6.1 permet au testeur de combiner plusieurs prédicats avant-après (de pr_1 à pr_n) par une disjonction afin de constituer le prédicat de pertinence. La règle 6.2 exprime le fait que les prédicats constituant le prédicat de pertinence peuvent être des prédicats dits *atomiques* (pa) ou *conditionnels* (pc). La règle 6.3 permet au testeur d'exprimer des prédicats *atomiques* (pa) : il s'agit de prédicats avant-après indiquant qu'une variable d'état x décroît ($x' < x$), croît ($x' > x$) ou passe d'une valeur énumérée à une autre ($x = v \wedge x' = v'$ où v et v' sont deux valeurs distinctes dans le domaine de la variable énumérée x) par application d'un événement. La règle 6.4 permet au testeur d'exprimer des prédicats conditionnels (pc). Ils prennent la forme d'une conjonction d'implications $b_i \Rightarrow ap_i$ (avec $i \in 1..m$) où b_i est une condition sur l'état source de la transition.

N.B. Pour tout état concret c, il doit exister exactement une valeur de i telle que b_i est satisfait par c (noté $c \models b_i$).

Le langage donné précédemment est conçu de telle sorte qu'un variant puisse être extrait à partir de chacune des règles.

6.2.3/ VARIANT ASSOCIÉ AUX ÉTATS CONCRETS EN FONCTION D'UN PRÉDICAT DE PERTINENCE

Pour que l'algorithme RCXP termine, on associe à chaque état concret de l'ATS qu'il produit une valeur de variant. Pour cela, on enrichit la notion d'ATS en lui ajoutant une fonction $v : C \to \mathbb{N}$ associant un entier positif ou nul à chaque état concret. Les sections 6.4.2 et 6.4.3 présentent des exemples de calcul des valeurs de variants pour certains états de l'exemple fil rouge. L'algorithme RCXP se décompose en deux étapes :

1. Définition de la fonction v pour tous les états concrets de l'ATS d'entrée produit par CXP ou CXP_{ASO}.
2. Prolongation des exécutions à partir de ces états par des séquences d'exécution sur lesquelles le variant décroît.

On qualifie un événement de « pertinent » lorsqu'il existe une transition abstraite par cet événement dont l'instance est susceptible d'être pertinente. L'ensemble des événements pertinents Ev_{pert} constitue donc un sous-ensemble de l'ensemble des événements d'un ES.

Afin d'optimiser l'exploration réalisée par RCXP, seules les transitions abstraites dont les événements pertinents seront instanciées, puisqu'on a la garantie que les autres événements ne peuvent pas donner lieu à des transitions pertinentes. Un événement e appartient à l'ensemble Ev_{pert} si et seulement si $\texttt{SAT}(Inv \wedge Inv[{}^{X'}/_{X}] \wedge prd_X(e) \wedge pert_X) = sat$.

Étant donné l'ensemble des événements pertinents Ev_{pert}, un prédicat de pertinence pr et un état concret c, le calcul de la valeur du variant sur les états de l'ATS d'entrée est effectué par l'ensemble de règles 2.

Règles 2 : Calcul du variant sur les états de l'ATS d'entrée

La valeur du variant de c est $v(c) \stackrel{def}{=} V_{init}(pert_X, c)$ où V_{init} est défini par les règles suivantes :

$$V_{init}(pa, c) \stackrel{def}{=} |Ev_{pert}| \times Card(Dom(x)) \text{ où } x \text{ est la variable ciblée par } pa \tag{6.5}$$

$$V_{init}(\bigwedge_{i=1}^{m} b_i \Rightarrow pa_i, c) \stackrel{def}{=} V_{init}(pa_i, c) \text{ avec } i \text{ unique tel que } \texttt{SAT}(c \wedge b_i) \tag{6.6}$$

$$V_{init}(\bigvee_{i=1}^{n} pr_n, c) \stackrel{def}{=} V_{init}(pr_1, c) + \ldots + V_{init}(pr_n, c) \tag{6.7}$$

La règle 6.5 permet de calculer la valeur initiale du variant d'un état c pour un prédicat atomique pa apparaissant dans le prédicat de pertinence $pert_X$ à partir du nombre d'événements pertinents et du cardinal du domaine de la variable ciblée par pa. Ce calcul n'est évidemment applicable qu'à des systèmes dont les variables d'état ont des domaines finis. Cependant, dans le cas où des variables auraient un domaine infini, on

pourrait imaginer qu'une borne supérieure soit arbitrairement fixée par le testeur. Intuitivement, le variant est calculé à partir des domaines de définition des variables afin que lors de l'exploration, une variable ciblée par un prédicat atomique de $pert_X$ puisse prendre toutes les valeurs de son domaine successivement.
La règle 6.6 permet de calculer la valeur initiale du variant d'un état c pour un prédicat conditionnel pc. Cette valeur correspond à la valeur du variant initial de c pour le prédicat atomique pa_i dont la condition b_i est satisfaite par c. On considère qu'il n'existe qu'une condition satisfaisant c dans pc.
Enfin, la règle 6.7 permet de calculer la valeur initiale du variant d'un état c pour une disjonction de prédicats avant-après pr, c'est-à-dire pour un prédicat de pertinence tel que définit par la règle 6.1. La valeur de ce variant est la somme des valeurs des variants initiaux de c pour les prédicats avant-après pr.

Étant donné un prédicat de pertinence $pert_X$ et une transition $c \xrightarrow{e} c'$ ajoutée par RCXP, la valeur du variant de c' est définie par l'ensemble de règles 3. Lorsque l'algorithme RCXP détermine si une transition $c \xrightarrow{e} c'$ est pertinente, la valeur du variant de c' (notée $v(c')$) est calculée. Elle dépend notamment du domaine de définition des variables apparaissant dans le prédicat de pertinence et de la valeur du variant de c (notée $v(c)$), d'où le fait que la fonction V prenne les trois paramètres $pert_X$, c et c'.

Règles 3 : Calcul du variant sur les états instanciés par RCXP

La valeur du variant de c' est $v(c') \stackrel{def}{=} V(pert_X, c, c')$ où V est définie par les règles suivantes :

$$V(x' < x, c, c') \stackrel{def}{=} v(c) - |c'.x - c.x| \tag{6.8}$$

$$V(x' > x, c, c') \stackrel{def}{=} v(c) - |c'.x - c.x| \tag{6.9}$$

$$V(x = v \wedge x' = v', c, c') \stackrel{def}{=} v(c) - 1 \tag{6.10}$$

$$V(\bigwedge_{i=1}^{m} b_i \Rightarrow pa_i, c, c') \stackrel{def}{=} V(pa_i, c, c') \text{ avec } i \text{ unique tel que } \mathtt{SAT}(c \wedge b_i) \tag{6.11}$$

$$V(\bigvee_{i=1}^{n} pr_n, c, c') \stackrel{def}{=} \sum_{\{i \mid i \in 1..n\}} \begin{cases} v(c) \text{ si } \mathtt{SAT}(c \wedge c'[{}^{X'}/_{X}] \wedge \neg pr_i) \\ V(pr_i, c, c') \text{ si } \mathtt{SAT}(c \wedge c'[{}^{X'}/_{X}] \wedge pr_i) \end{cases} \tag{6.12}$$

L'ensemble de règles 3 indique comment calculer la valeur du variant d'un état c' cible d'une transition $c \xrightarrow{e} c'$ où e est un événement quelconque.
La règle 6.8 permet de calculer le variant pour un prédicat atomique imposant la décroissance d'une variable. La valeur du variant de c' ($v(c')$) est celle du variant de c ($v(c)$) à laquelle on soustrait la variation de valeurs de la variable ($|c.x' - c.x|$).
La règle 6.9 permet de calculer le variant pour un prédicat atomique imposant la croissance d'une variable. De même que pour la règle 6.8 valeur du variant de c' ($v(c')$) est celle du variant de c ($v(c)$) à laquelle on soustrait la variation de valeurs de la variable ($|c.x' - c.x|$).
La règle 6.10 permet de calculer le variant pour un prédicat atomique imposant un changement de valeur d'une variable dont le domaine est une énumération. Dans ce cas, afin que la variable ciblée par ce prédicat atomique puisse prendre toutes les valeurs de son domaine durant l'exploration, on soustrait 1 à la valeur de $v(c)$.
La règle 6.11 indique que la valeur du variant pour un prédicat conditionnel est celle du variant pour le prédicat atomique pa_i dont la condition b_i est satisfaite par c. À nouveau, on considère qu'il n'existe qu'une condition b_i satisfaisant c dans le prédicat conditionnel.

La règle 6.12 permet de calculer la valeur du variant pour une disjonction de prédicat avant-après (donc pour le prédicat de pertinence $pert_X$. On considère qu'il est égal à la somme des valeurs de variant de l'état c par un prédicat avant-après pr_i lorsque c et c' ne satisfont pas pr_i et, lorsqu'ils le satisfont, à la valeur du variant de c' pour pr_i.

Garantie de décroissance du variant : La décroissance du variant au cours de l'exploration est garantie par les règles 6.8, 6.9, 6.10, 6.11 et 6.12. En effet, un prédicat de pertinence est considéré comme une disjonction de prédicat avant-après. C'est donc la règle 6.12 qui est utilisée initialement lorsqu'il faut calculer la valeur du variant d'un état c' d'une transition $c \xrightarrow{e} c'$.

Pour chaque prédicat avant-après pr_i apparaissant dans $pert_X$, si la conjonction de c et c' ne satisfait pas pr_i, alors la valeur de $v(c)$ est utilisée, sinon, la valeur du variant de c' pour pr_i est utilisée. La règle 6.12 calculant la somme des variants pour chaque prédicat avant-après pr_i, il est nécessaire de s'assurer que cette somme décroît strictement. Si la conjonction de c et c' ne satisfait aucun prédicat avant-après, la valeur du variant de c' est la somme des valeurs des variants de c et ne décroît pas. Cependant, on s'assurera toujours, lors de l'exploration, que la transition est pertinente avant de calculer la valeur du variant d'un état c'. Ainsi, la conjonction de c et c' est garantie de satisfaire au moins un prédicat avant-après du prédicat de pertinence. Par conséquent, il suffit que toutes les règles de calcul de variant pour un prédicat avant-après soient garanties de décroître pour que le variant décroisse lors de l'exploration.

La règle 6.8 garantit la décroissance du variant entre c et c' puisqu'elle soustrait une valeur positive à $v(c)$. En effet, $|c'.x - c.x|$ est nécessairement positive étant donné que x' est inférieure *strictement* à x (ceci est imposé par le prédicat atomique $x' < x$). La règle 6.9 garantit la décroissance du variant pour les mêmes raisons que pour la règle 6.8. La règle 6.10 soustrait 1 à $v(c)$ et garantit donc également la décroissance du variant. Enfin, la règle 6.11 fait appel aux règles concernant les prédicats atomiques (c'est-à-dire les règles 6.8, 6.9 et 6.10) dont la décroissance est garantie.

On déduit donc que la décroissance du variant est bien garantie par l'ensemble de règles 3.

6.3/ ALGORITHME RCXP

Cette section détaille l'algorithme RCXP en donnant son principe de fonctionnement général, puis ses paramètres d'entrée, de sortie et les variables qu'il manipule et enfin son fonctionnement détaillé.

6.3.1/ PRINCIPES DE L'ALGORITHME RCXP

L'algorithme RCXP complète l'ATS qu'il reçoit en entrée en prolongeant l'exploration à partir de chaque état concret de l'ATS d'entrée (et de chaque état concret qu'il ajoute) dont le variant est positif ou nul. On dit d'un état concret dont le variant est positif ou nul qu'il est *pertinent* (« relevant » en anglais). RCXP instancie exactement une fois chaque may-transition pouvant être instanciée depuis un état concret *pertinent* rencontré durant l'exploration (i.e. un état pour lequel l'évaluation du variant avec le prédicat de pertinence

$pert_X$ était supérieure ou égale à 0). RCXP ajoute tout état concret cible c' obtenu par un événement e quelconque à partir d'un état concret pertinent c et qui respecte le prédicat de pertinence (i.e. dont la transition $c \xrightarrow{e} c'$ satisfait le prédicat de pertinence).

6.3.2/ ENTRÉES, SORTIES ET VARIABLES DE L'ALGORITHME RCXP

Cette section décrit les paramètres d'entrée et de sortie ainsi que les variables utilisées par RCXP.

6.3.2.1/ ENTRÉES

Les entrées de l'algorithme sont décrites ci-dessous :

- $\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa \rangle$: il s'agit d'un ATS n'ayant pas couvert toutes les transitions abstraites que le testeur souhaitait voir couvertes. Il correspond à la sortie de l'algorithme CXP dans le cas de RCXP, et à la sortie de CXP_{ASO} dans le cas de RCXP_{ASO}. Cet ATS est complété par l'algorithme RCXP.
- A : il s'agit de l'ensemble contenant tous les états abstraits satisfaisant l'invariant du système à approximer (i.e. les états abstraits non vides).
- $pert_X$: il s'agit d'un prédicat dit de *pertinence*. Il prend la forme d'une disjonction de prédicats avant-après portant chacun sur une des variables du système à approximer. Ce prédicat de pertinence doit être vrai pour tout état concret que le testeur souhaite atteindre pour couvrir les transitions abstraites non couvertes. Dans le cas du distributeur de café cité au début de cette section, ce prédicat devrait donc accepter l'insertion de monnaie dans le distributeur, la demande d'un café et la délivrance d'un café si le testeur souhaite couvrir la dernière vente de café.

6.3.2.2/ SORTIES

Les sorties de l'algorithme sont décrites ci-dessous :

- $\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa, v \rangle$: il s'agit de l'ATS d'entrée qui sera complété, d'une part, par la fonction $v : C \to \mathbb{N}$ qui associe une valeur de variant à chaque état concret et, d'autre part, par les nouveaux états concrets et les nouvelles transitions concrètes calculés par l'algorithme RCXP.

6.3.2.3/ VARIABLES

Les variables utilisées par l'algorithme sont décrites ci-dessous :

- c, c' : il s'agit des états source et cible courants.
- vc' : il s'agit de la valeur du variant de l'état c'.
- RCS : il s'agit de l'ensemble des états concrets à partir desquels l'exploration doit procéder. Ces états doivent satisfaire le prédicat de pertinence ; on parle donc d'états pertinents.
- $PRCS$: il s'agit de l'ensemble des états concrets instanciés par RCXP et à partir desquels l'exploration à déjà eu lieu.
  N.B. RCXP n'instancie que des états concrets pertinents.

6.3.3/ FONCTIONNEMENT DE L'ALGORITHME RCXP

Cette section détaille le fonctionnement de l'algorithme RCXP dont la version formelle est donnée par l'algorithme 4. **N.B.** Afin de simplifier l'algorithme, le calcul de l'ensemble des événements pertinents Ev_{pert} n'apparaît pas dans cet algorithme mais est bel et bien pris en compte dans le calcul du variant. De plus, seuls les transitions dont l'événement e appartient à Ev_{pert} sont instanciées.

Initialement, l'ATS de sortie est identique à l'ATS d'entrée (ligne 1). Puis l'algorithme 4 calcule le variant des états concrets existants en entrée. Ensuite RCXP complète l'ATS en prolongeant les exécutions par des états pertinents dont le variant est positif ou nul.

Au départ, l'ensemble des états concrets pertinents $PRCS$ est vide puisque l'exploration n'a encore eu lieu depuis aucun état concret (ligne 5). L'algorithme RCXP considère également que tout état concret atteint ($vert$) dans l'ATS fourni en entrée est pertinent (à condition que la valeur initiale de son variant soit supérieure ou égale à 0). Puisque l'exploration devra avoir lieu depuis tous ces états, ils sont ajoutés à l'ensemble RCS (ligne 6). L'exploration effectuée par RCXP se poursuit tant que des états concrets pertinents existent dans RCS (voir la boucle de la ligne 7 à la ligne 40).

6.3.3.1/ CALCUL DE L'ENSEMBLE DES ÉTATS CONCRETS PERTINENTS DE DÉPART

L'algorithme choisit un état concret c parmi ceux contenus dans RCS, les ajoute à l'ensemble $PRCS$ des états concrets pertinents dont l'exploration a déjà eu lieu (ligne 10) et procède à l'exploration depuis c. Notons que l'état abstrait q correspondant à c est nécessairement connu : si c a été calculé par CXP ou CXP_{ASO}, alors son état abstrait q correspondant est $q \stackrel{def}{=} \alpha(c)$. Si c a été calculé au cours de l'exploration par RCXP, alors son état abstrait correspondant était nécessairement connu au moment où il à été ajouté dans RCS et peut donc également être obtenu par $\alpha(c)$ (ligne 11).

6.3.3.2/ CALCUL D'UNE INSTANCE CONCRÈTE PERTINENTE DE CHAQUE TRANSITION ABSTRAITE *may*-ATTEIGNABLE

Le processus d'exploration est similaire à celui de CXP : une instance concrète de chaque transition abstraite pouvant être instanciée depuis c est calculée. Par conséquent, l'algorithme RCXP itère sur l'ensemble des états abstraits A et sur l'ensemble des événements $e \in Ev$ (lignes 12 et 13 respectivement).

Si la transition abstraite $q \xrightarrow{e} q'$ est may-atteignable (ligne 14), une instance concrète pertinente de cette transition dont l'état source est c pourrait exister. Par conséquent, l'algorithme RCXP tente de trouver une transition concrète menant de l'état source c à un état cible de q' par l'événement e et pour laquelle le prédicat de pertinence soit vérifié (ligne 15).

Si une telle transition $c \xrightarrow{e} c'$ existe (avec c' l'état concret cible tel que $\alpha(c') \stackrel{def}{=} q'$), elle est pertinente et RCXP vérifie si la valeur du variant associée à c' par le prédicat de pertinence $pert_X$ est supérieur ou égal à 0 (ligne 18). Si c'est le cas, cela signifie que l'exploration peut et doit se poursuivre depuis l'état concret c' instancié. Il est donc ajouté à l'ensemble RCS (ligne 19) sauf si l'exploration a déjà eu lieu depuis lui (i.e. si c' avait

déjà été instancié avant, par CXP, CXP_{ASO} ou RCXP pendant l'exploration, et appartient donc à $PRCS$).

Si $c \xrightarrow{e} c'$ est pertinente et même si la valeur du variant en c' est inférieure à 0, cette transition à été calculée et l'ATS est donc complété (voir lignes 21 à 25). L'état c' n'est toutefois pas ajouté à RCS car n'est pas considéré comme pertinent. Si aucune transition *pertinente* dont l'état source est c ne peut être trouvée par l'événement e, cela signifie que c était un état ciblé par le testeur (car la fonction de pertinence qu'il a conçue ne permet pas à l'exploration de se poursuivre plus loin).

6.3.3.3/ CALCUL DES TRANSITIONS DÉCLENCHABLES APRÈS L'ATTEINTE D'UN ÉTAT PERTINENT

Une dernière transition par e est instanciée depuis cet état c (si cela est possible) pour permettre au testeur d'observer ce qui peut se passer une fois l'état cible atteint (ligne 28). Si une telle transition existe, l'ATS est évidemment complété (voir lignes 30 à 34), là encore sans enrichir RCS. L'exploration se termine lorsque RCS est vide, c'est-à-dire lorsqu'aucun nouvel état concret instancié lors de l'exploration ne s'est révélé pertinent, ou lorsqu'aucune nouvelle transition concrète n'est trouvée depuis l'un des états de RCS.

6.4/ APPLICATION A L'EXEMPLE FIL ROUGE

Cette section présente le déroulement de l'algorithme RCXP sur l'exemple fil rouge du distributeur de cafés.

6.4.1/ PARAMÈTRES EN ENTRÉE

L'ATS utilisé en entrée de l'algorithme RCXP pour cet exemple est celui généré par CXP représenté par un graphe sur la figure 4.1 page 74. L'ensemble des états abstraits est identique à celui fourni en entrée à CXP : $A \stackrel{def}{=} \{q_0, q_1, q_2, q_3, q_4\}$. Enfin, le prédicat de pertinence $pert_X$ est celui illustré en section 6.2.1 permettant la couverture de la transition servant le dernier café ($q_2 \xrightarrow{serveCof} q_1$) :

$$pert_X \stackrel{def}{=} (\mathit{Balance'} > \mathit{Balance}) \vee (\mathit{AskCof} = \mathit{false} \wedge \mathit{AskCof'} = \mathit{true}) \vee (\mathit{CofLeft'} < \mathit{CofLeft})$$

On fixe ici les valeurs des constantes *MAX_BAL*, *MAX_POT* et *MAX_COF* à 200, 500 et 10 respectivement.

6.4.2/ CALCUL DE L'ENSEMBLE DES ÉTATS CONCRETS PERTINENTS DE DÉPART

L'ATS fourni en entrée de RCXP contient onze états concrets $verts$, donc atteints (voir figure 4.1 page 74). Tous ces états concrets sont donc potentiellement pertinents. Pour qu'ils le soient, il faut également que leur variant vis-à-vis du prédicat de pertinence soit supérieur ou égal à 0.

L'algorithme RCXP calcule donc pour chacun la valeur de leurs variants respectifs vis-à-vis du prédicat de pertinence $pert_X$. Pour l'état $c_0q_0 \stackrel{def}{=} \mathit{Status} = \mathit{error} \wedge \mathit{AskChange} = \mathit{false} \wedge \mathit{AskCof} = \mathit{false} \wedge \mathit{Balance} = 0 \wedge \mathit{Pot} = 100 \wedge \mathit{CofLeft} = 0$, les étapes de calcul du variant sont les suivantes (le nombre d'événements pertinents étant 4) :

$$
\begin{array}{rcl}
V_{init}(pert_X, c_0q_0) & \stackrel{def}{=} & 4 \times (V_{init}(\mathit{Balance'} > \mathit{Balance}, c_0q_0) + \\
& & V_{init}(\mathit{AskCof} = \mathit{false} \wedge \mathit{AskCof'} = \mathit{true}, c_0q_0) + \\
& & V_{init}(\mathit{CofLeft'} < \mathit{CofLeft}, c_0q_0)) \\
& \stackrel{def}{=} & 4 \times (Card(Dom(\mathit{Balance})) + \\
& & Card(Dom(\mathit{AskCof})) + \\
& & Card(Dom(\mathit{CofLeft}))) \\
& \stackrel{def}{=} & 4 \times (|\{0, 50, 100, 150, 200\}| + |\{\mathit{false}, \mathit{true}\}| + |0..\mathit{MAX\_COF}|) \\
& \stackrel{def}{=} & 4 \times (5 + 2 + 11) \\
& \stackrel{def}{=} & 72
\end{array}
$$

Le calcul pour tous les autres états $verts$ cités précédemment est exactement identique et vaut donc 72 pour chacun d'entre eux. Cela signifie qu'au maximum 72 événements pourront être déclenchés depuis chacun de ces états.

En l'occurrence, la vente des 10 cafés dans le distributeur initialement ne requière l'application que de trente événements une fois le distributeur mis en état de fonctionnement ($\underbrace{10}_{\text{nombre initial de cafés}} \times \underbrace{3}_{|\{\mathit{insert50}, \mathit{askCof}, \mathit{serveCof}\}|} = 30$).

6.4.3/ CALCUL D'UNE INSTANCE CONCRÈTE PERTINENTE DE CHAQUE TRANSITION ABSTRAITE *may*-ATTEIGNABLE

Les transitions *may*-atteignables sont connues car présentes dans l'ensemble Δ fourni par CXP. D'après le MTS calculé par CXP (figure 6.1), on constate qu'il en existe 21. RCXP tente alors de calculer une instance de ces transitions dont l'état concret soit pertinent (donc dans RCS). La transition $q_2 \xrightarrow{\mathit{insert50}} q_2$ peut par exemple être instanciée par la transition $c_{11}q_2 \xrightarrow{\mathit{insert50}} c_8q_2$, $c_{11}q_2$ se trouvant dans RCS et c_8q_2 ayant déjà été calculé par CXP. Cette instance est pertinente car le prédicat de pertinence $pert_X$ autorise l'événement $insert50$. Le variant de $c_{11}q_2$ vaut 72 comme tous les autres états calculés par CXP, mais le variant de c_8q_2 est recalculé lors de l'instanciation de la transition $q_2 \xrightarrow{\mathit{insert50}} q_2$. Le calcul permettant de déterminer la valeur du variant pour c_8q_2 lorsqu'il est atteint par la transition $c_{11}q_2 \xrightarrow{\mathit{insert50}} c_8q_2$ est le suivant :

$$
\begin{array}{rcl}
V(pert_X, c_{11}q_2, c_8q_2) & \stackrel{def}{=} & V(\mathit{Balance'} > \mathit{Balance}, c_{11}q_2, c_8q_2) + \\
& & V(\mathit{AskCof} = \mathit{false} \wedge \mathit{AskCof'} = \mathit{true}, c_{11}q_2, c_8q_2) + \\
& & V(\mathit{CofLeft'} < \mathit{CofLeft}, c_{11}q_2, c_8q_2) \\
& \stackrel{def}{=} & V_{init}(\mathit{Balance'} > \mathit{Balance}, c_{11}q_2) - 1 + \\
& & V_{init}(\mathit{AskCof} = \mathit{false} \wedge \mathit{AskCof'} = \mathit{true}, c_{11}q_2) + \\
& & V_{init}(\mathit{CofLeft'} < \mathit{CofLeft}, c_{11}q_2) \\
& \stackrel{def}{=} & 4 \times Card(Dom(\mathit{Balance})) - 1 + \\
& & 4 \times Card(Dom(\mathit{AskCof})) + \\
& & 4 \times Card(Dom(\mathit{CofLeft})) \\
& \stackrel{def}{=} & (4 \times 5) - 1 + (4 \times 2) + (4 \times 11) \\
& \stackrel{def}{=} & 71
\end{array}
$$

Le variant de l'état c_8q_2 valant 72 au début de l'algorithme est donc décrémenté et vaut 71 lors du calcul de l'instance $c_{11}q_2 \xrightarrow{insert50} c_8q_2$. L'état concret c_8q_2 étant pertinent et la valeur de son variant étant supérieure à 0, il est ajouté à l'ensemble RCS et l'exploration pourra donc se poursuivre depuis cet état. L'état cible (ici c_8q_2) et la transition calculée sont ajoutés aux ensembles C et Δ^c respectivement et la fonction d'abstraction α est enrichie par $\alpha(c_8q_2) = q_2$.

RCXP ne termine que lorsqu'aucun nouvel état pertinent n'est atteint lors de l'exploration ou lorsqu'aucun variant d'état atteint lors de l'exploration n'est supérieur ou égal à 0. Ici, l'exploration instancie les transitions *insert50*, *insert100*, *cofReq* et *serveCof* jusqu'à atteindre les états où l'une des conditions suivantes est vérifiée :

- Il ne reste plus de café dans le distributeur ($CofLeft = 0$)
- La balance est pleine ($Balance = MAX\_BALANCE$)
- Le pot est plein ($Pot > MAX\_POT + 50$)

L'exploration atteint donc les états ciblés (dans lesquels il ne reste aucun café) et la transition abstraite $q_2 \xrightarrow{serveCof} q_1$ est donc couverte (plusieurs fois par différentes séquences de transitions en l'occurrence). La valeur du variant des états atteints dans lesquels il ne reste plus de café est comprise entre 42 et 45 avec les paramètres utilisés pour ce cas d'étude.

6.4.4/ CALCUL DES TRANSITIONS DÉCLENCHABLES APRÈS L'ATTEINTE DU DERNIER ÉTAT PERTINENT

Après la phase d'instanciation des transitions pertinentes, RCXP calcule, lorsque c'est possible, une instance concrète de chaque transition abstraite à partir des états concrets instanciés mais non pertinents.

En l'occurrence, cette étape permet la couverture de l'état abstrait q_4 et de la transition $q_1 \xrightarrow{powerDown} q_4$ par exemple qui nécessite que le *Pot* soit rempli, ce qui n'est le cas (avec les paramètres utilisés) que lorsque tous les cafés ont été vendus. La couverture de cette transition nécessite donc d'abord d'atteindre un état concret dans lequel il ne reste plus de café, ce que vise justement le prédicat de pertinence. Ainsi, même si l'événement *powerDown* n'était pas autorisé par le prédicat de pertinence, la transition $q_1 \xrightarrow{powerDown} q_4$ est couverte dans l'ATS calculé par RCXP grâce à cette ultime phase d'instanciation.

6.5/ DISCUSSIONS SUR LA COMPLEXITÉ, LA TERMINAISON, LA COMPLÉTUDE ET LA CORRECTION DE RCXP

Cette section discute la complexité, la terminaison, la complétude et la correction de l'algorithme RCXP.

6.5.1/ COMPLEXITÉ DE RCXP

La boucle de la ligne 2 à la ligne 4 calcule la valeur du variant pour les états $verts$ présents dans l'ensemble C de l'ATS $in \stackrel{def}{=} \langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa \rangle$ passé en paramètre de l'algo-

rithme. Le nombre d'états $verts$ est au maximum égal au nombre d'éléments dans C : il s'agit du cas où tous les états instanciés sont atteints. En considérant que la complexité du calcul de la valeur du variant à partir d'un prédicat d'abstraction est $\mathrm{O}(1)$, la complexité de cette boucle est donc $\mathrm{O}(|\mathsf{in}.C| \times \mathrm{O}(1)) = \mathrm{O}(|\mathsf{in}.C|)$.

La boucle de la ligne 7 à la ligne 40 ne s'arrête pas tant qu'une transition pertinente, dont la cible n'est pas connue, est découverte lors de l'exploration. Sa complexité dépend donc du nombre de transitions pertinentes découvertes, donc du prédicat de pertinence. Cependant, le variant est conçu pour décroître lorsqu'une transition pertinente est instanciée. Ainsi, l'algorithme garantit que la plus grande séquence de transitions explorée par RCXP sera composée d'au plus $|Ev_{pert}| \times n \in \mathbb{N}$ transitions, où $|Ev_{pert}|$ est le nombre d'événements pertinents (une constante) et n est la plus grande valeur de variant attribuée aux états de l'ATS $\ll \mathsf{in} \gg$.

Dans le pire des cas, l'algorithme RCXP instancie donc des séquences de transitions de taille $|Ev_{pert}| \times max(\{v(c) \mid c \in \mathsf{in}.C \wedge \kappa(c) = vert\})$ à partir de chaque état concret de $\mathsf{in}.C$. La valeur de $max(\{v(c) \mid c \in \mathsf{in}.C \wedge \kappa(c) = vert\})$ est dépendante du cardinal des domaines des variables utilisées dans le prédicat de pertinence. On note que la complexité de RCXP, dans le cas où la valeur du variant est le facteur limitant l'exploration (et non le prédicat de pertinence), ne dépend pas du nombre d'états abstraits. De plus, puisque $|Ev_{pert}|$ est une constante, il n'intervient pas dans le calcul de la complexité.

La complexité globale de l'algorithme RCXP est ainsi calculée de la manière suivante : $\mathrm{O}(|\mathsf{in}.C| + max(\{v(c) \mid c \in \mathsf{in}.C \wedge \kappa(c) = vert\}) \times |\mathsf{in}.C|) = \mathrm{O}(max(\{v(c) \mid c \in \mathsf{in}.C\}) \times |\mathsf{in}.C|)$.

6.5.2/ TERMINAISON DE RCXP

La boucle calculant la valeur du variant pour les états de l'ATS fourni en entrée de RCXP (de la ligne 2 à la ligne 4) termine car elle itère sur les états $verts$ de l'ensemble fini $\mathsf{in}.C$.

L'algorithme RCXP procède ensuite à une exploration des transitions à partir de l'ensemble des états $\mathsf{in}.C$ de l'ATS $\ll \mathsf{in} \gg$ passé en entrée. En raison de la boucle de la ligne 7 à la ligne 40, il s'arrête lorsqu'aucun nouvel état concret cible d'une transition pertinente n'est découvert au cours de l'exploration. En effet, la boucle se termine lorsque RCS est vide (voir la condition de la ligne 7). Or, RCS n'est complété que lorsqu'un nouvel état cible d'une transition pertinente est découvert et que son variant est positif ou nul (voir la condition de la ligne 18 et la ligne 19 où c' est ajouté à RCS).

Puisque les règles de calcul du variant garantissent sa décroissance lorsqu'une nouvelle transition pertinente est instanciée, la condition de la ligne 19 finira nécessairement par être fausse (vc' finira par être inférieur à 0). Ainsi, RCS finira par être vide et la boucle principale de la ligne 7 à 40 terminera également.

La terminaison de l'algorithme RCXP est donc garantie en raison de la garantie de décroissance du variant des états instanciés lors de l'exploration. Il peut également terminer plus tôt lorsque le prédicat de pertinence ne permet plus d'instancier de nouvelles transitions pertinentes.

6.5.3/ COMPLÉTUDE ET CORRECTION DE RCXP

L'algorithme RCXP pourrait être considéré comme complet si et seulement si toutes les transitions abstraites ciblées par le testeur sont couvertes par les transitions instanciées lors de l'exploration. Le problème de déterminer si une transition abstraite peut être couverte n'étant cependant pas décidable, il n'est pas possible de conclure sur la complétude de l'algorithme vis-à-vis de ce critère. De plus, les valeurs des variants sont calculées en fonction des domaines de définition des variables. Il est donc possible que ces valeurs soient trop basses pour que les séquences de transitions pertinentes trop longues ne soient pas instanciées complètement : lorsque le variant devient inférieur à 0, les transitions ne sont plus instanciées. Cela peut empêcher la concrétisation de transitions abstraites accessibles uniquement par de nombreux cycles ou de trop longues séquences de transitions.

L'algorithme RCXP n'instancie que les transitions pertinentes selon le prédicat de pertinence conçu par le testeur (voir la ligne 15 qui s'assure qu'une instance de transition pertinente existe). De plus, l'ajout d'un état concret source de l'exploration n'est réalisé qu'à condition qu'une instance de transition pertinente dont le variant est positif ou nul ait effectivement été découverte (voir les conditions des lignes 16 et 18). Lorsqu'un état cible est atteint, une instance de transition supplémentaire est calculée si elle existe (ligne 28) afin d'observer les comportements applicables depuis un tel état. L'algorithme RCXP est donc correct en ce sens qu'il ne calcule que des instances de transitions pertinentes conformément au prédicat de pertinence, avec cependant une étape d'instanciation supplémentaire une fois les transitions pertinentes instanciées.

6.6/ BILAN ET CONCLUSIONS

L'algorithme CXP peut générer un ATS dans lequel certaines transitions abstraites dont les instances ne peuvent être atteintes par une seule occurrence de certaines transitions ne soient pas couvertes. Il calcule malgré tout un MTS complet et permet de savoir quelles sont ces transitions non couvertes. À partir de cette information et de sa connaissance du système, le testeur peut alors estimer les transitions qui sont effectivement atteignables et qu'il souhaite voir couvertes dans l'ATS. Après avoir identifié les événements susceptibles de mener le système dans un état où ces transitions (dites transitions *pertinentes* ou *ciblées*) peuvent être déclenchées, il conçoit un prédicat de pertinence exprimant l'évolution des variables par application de ces événements. Ce prédicat de pertinence guide l'exploration réalisée par l'algorithme RCXP de telle sorte que les instances de transitions calculées par ce dernier mènent à un état permettant de couvrir les transitions ciblées. La terminaison de la phase d'exploration est garantie par le calcul d'un variant initialement positif et décroissant strictement à partir du prédicat de pertinence conçu par le testeur.

Ainsi, si le prédicat de pertinence autorise les opérations permettant de mener le système jusqu'à un (ou plusieurs) état(s) à partir desquels le déclenchement de la (ou les) transition(s) à couvrir est possible, ces transitions cibles seront couvertes dans l'ATS généré par RCXP. L'algorithme RCXP adresse donc le problème de couverture pouvant être engendré par CXP. Il requiert l'intervention du testeur pour la conception du prédicat de pertinence, mais est garanti de terminer en un temps fini par l'extraction d'un variant strictement décroissant.

Il est important de noter que le prédicat de pertinence ne doit en revanche pas être trop faible (par exemple en autorisant tous les événements à être déclenchés), auquel cas l'exploration réalisée, malgré sa finitude, s'apparenterait à une exploration de l'espace d'état complet du système.

Algorithme 4 : Relevant Concrete eXPloration (RCXP)

Inputs : in = $\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa\rangle$: un ATS ne couvrant pas toutes les transitions abstraites
A : l'ensemble des états abstraits
$pert_X$: le prédicat de pertinence pour couvrir les transitions non couvertes
Output : out = $\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa, v\rangle$: l'ATS enrichi
Variables : c, c' : les états concrets source et cible de la transition instanciée
vc' : le variant de l'état concret c'
RCS : l'ensemble des états concrets à partir desquels l'exploration doit procéder
$PRCS$: l'ensemble des états dont l'exploration a déjà eu lieu

1 out.$\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa\rangle$:= in.$\langle Q_0, Q, \Delta, C_0, C, \Delta^c, \alpha, \kappa\rangle$;
2 **foreach** $c \in$ out.$C \wedge \kappa(c) = vert$ **do**
3 $v(c) := V_{init}(pert_X, c)$
4 **end**
5 $PRCS := \emptyset$;
6 $RCS := \{c \mid c \in C \wedge \kappa(c) = vert \wedge v(c) \geq 0\}$;
7 **while** $RCS \neq \emptyset$ **do**
8 **choose** $c \in RCS$;
9 $RCS := RCS - \{c\}$;
10 $PRCS := PRCS \cup \{c\}$;
11 $q := \alpha(c)$;
12 **foreach** $q' \in A$ **do**
13 **foreach** $e \stackrel{def}{=} a \in Ev$ **do**
14 **if** $q \xrightarrow{e} q' \in \Delta$ **then**
15 $(c, c') := \mathtt{SAT}(c \wedge prd_X(a) \wedge q'[^{X'}/_X] \wedge pert_X)$;
16 **if** $(c, c') \notin \{unknown, unsat\}$ **then**
17 $vc' := V(pert_X, c, c')$;
18 **if** $vc' \geq 0 \wedge c' \notin PRCS$ **then**
19 $RCS := RCS \cup \{c'\}$;
20 **end**
21 $\alpha(c') := q'$;
22 $\kappa(c') := vert$;
23 $v(c') := vc'$;
24 $C := C \cup \{c'\}$;
25 $\Delta^c := \Delta^c \cup \{c \xrightarrow{e} c'\}$;
26 **else**
27 `// Un état cible est atteint`
28 $(c, c') := \mathtt{SAT}(c \wedge prd_X(a) \wedge q'[^{X'}/_X])$;
29 **if** $(c, c') \notin \{unknown, unsat\}$ **then**
30 $\alpha(c') := q'$;
31 $\kappa(c') := vert$;
32 $C := C \cup \{c'\}$;
33 $v(c') := V(pert_X, c, c')$;
34 $\Delta^c := \Delta^c \cup \{c \xrightarrow{e} c'\}$;
35 **end**
36 **end**
37 **end**
38 **end**
39 **end**
40 **end**
41 **return** out

7

MÉTHODE DE GÉNÉRATION DE TESTS PAR EXPLORATION CONCRÈTE ET PERTINENTE D'UNE ABSTRACTION PAR PRÉDICATS

Ce chapitre présente une suggestion de méthode complète de génération de tests permettant d'abord au testeur d'exprimer ses besoins en termes d'objectifs de test puis de générer les tests couvrant ces objectifs de test. La méthode combine les procédures CXP, CXP_{ASO}, BCI et RCXP présentés précédemment.

7.1/ PROCÉDURE COMPLÈTE POUR LA GÉNÉRATION DE TESTS POUR UN SYSTÈME ÉVÉNEMENTIEL

L'ingénieur souhaitant générer des tests pour un système événementiel doit évidemment en premier lieu créer le modèle. Une fois le modèle conçu, il est nécessaire de vérifier, à l'aide en général des techniques de « model-checking », sa conformité vis-à-vis des spécifications desquelles il est issu.

Il convient ensuite d'énoncer un objectif de test, par exemple sous la forme de patrons de spécifications tels que présentés en section 2.6 page 34. On propose d'extraire les prédicats d'abstraction de cet objectif de test selon la démarche suivante :

- si l'objectif de test fait apparaître des événements, on suggère d'utiliser la garde de ces événements comme prédicats d'abstraction,
- si l'objectif de test fait apparaître des expressions booléennes sur les variables d'état du système, on suggère d'utiliser ces expressions comme prédicats d'abstraction et enfin,
- si l'objectif de test fait apparaître à la fois des événements et des expressions booléennes, on propose d'utiliser à la fois les gardes des événements et les expressions comme prédicats d'abstractions.

N.B. Si la garde d'un événement est composée de plusieurs conditions, il est également envisageable d'utiliser chacune des conditions comme un prédicat d'abstraction unique. Par exemple, pour un événement dont la substitution est $P_1 \Rightarrow \ldots \Rightarrow P_n \Rightarrow S$ où $P_1, \ldots, P_n$ avec $n \in \mathbb{N}$ sont des expressions booléennes et S une substitution, chacune des expres-

sions $P_1, \ldots, P_n$ pourrait être utilisée comme un prédicat d'abstraction.

Une fois les prédicats d'abstraction extraits, la première étape consiste à calculer une première sous-approximation, à l'aide de CXP ou CXP_{ASO}. Ces algorithmes ne requièrent que le calcul des états abstraits non vides (l'ensemble A fourni en entrée des algorithmes CXP, CXP_{ASO} et RCXP vus précédemment). Cette étape est cependant entièrement automatisée dans l'outil STRATEST développé dans le cadre de cette thèse et présenté plus loin en section 8.1. L'ingénieur fournit donc uniquement à STRATEST le modèle de système événementiel et l'ensemble de prédicats d'abstraction et, après avoir sélectionné l'algorithme qu'il souhaite appliquer, l'outil se charge de calculer l'ensemble des états abstraits non vides et d'appliquer ces algorithmes. Nos résultats expérimentaux, présentés en section 8.3, suggèrent que CXP_{ASO} est plus intéressant que CXP, en ce sens qu'il permet généralement une meilleure couverture des états et des transitions du MTS.

Une fois l'algorithme CXP (ou CXP_{ASO}) appliqué par STRATEST, l'ingénieur analyse le MTS calculé et la couverture de ses états et transitions par la sous-approximation (toutes ces données sont fournies par STRATEST dès qu'un algorithme est appliqué). S'il juge la couverture suffisante, les tests générés automatiquement par STRATEST sont déjà disponibles sous forme d'un ensemble de séquences de transitions dans un fichier texte. Il ne reste donc qu'à les mettre en œuvre sur l'implémentation du système après avoir réalisé la phase d'adaptation des tests à l'implémentation (voir l'étape 4 en section 3.1.4 page 40).

Si l'ingénieur juge la couverture des états et des transitions du MTS obtenue par la sous-approximation calculée par CXP ou CXP_{ASO} insuffisante, il a la possibilité d'appliquer l'un des algorithmes permettant de compléter cette sous-approximation (BCI ou RCXP). Les modalités des transitions sont précisées sur le MTS fourni par STRATEST, et il est donc possible pour l'ingénieur de savoir si l'application de BCI, qui ne concrétisera de toute façon que les *must*-transitions, peut compléter la sous-approximation de manière à couvrir les états et transitions non couverts. Si les transitions non couvertes dans la sous-approximation calculée par CXP ou CXP_{ASO} n'ont pas la modalité *must*, il n'est donc pas utile d'appliquer BCI, à moins de souhaiter couvrir les *must*-transitions plusieurs fois dans les tests. Une fois BCI appliqué, les tests sont automatiquement générés à partir de la sous-approximation et si la couverture obtenue n'est toujours pas satisfaisante, la dernière solution est d'appliquer RCXP.

Pour utiliser l'algorithme RCXP (ou indifféremment RCXP_{ASO} si CXP_{ASO} a été utilisé), le testeur doit identifier les cycles sur le MTS susceptibles de devoir être exécutés pour couvrir les états et transitions non couverts et qu'il souhaite couvrir. Il identifie ensuite l'ensemble des événements apparaissant dans ces cycles et conçoit un prédicat de pertinence autorisant ces événements à se déclencher. L'algorithme RCXP prend notamment en paramètre en entrée ce prédicat de pertinence ainsi qu'un ATS. Cet ATS peut aussi bien être celui issu de CXP, CXP_{ASO} ou BCI, si l'utilisateur à choisi de compléter une première fois l'ATS par l'instanciation des *must*-transitions. L'application de RCXP, comme celle des autres algorithmes, est entièrement automatisée par STRATEST une fois que le testeur a précisé l'ATS et le prédicat de pertinence à utiliser.
Il est également possible de ne cibler qu'une partie des transitions non couvertes dans un premier temps, par exemple si différents cycles sont nécessaires pour couvrir différents états ou transitions. En effet, RCXP calcule également un ATS et cet ATS peut parfaitement être réutilisé en entrée de RCXP avec un prédicat de pertinence différent. Cette méthode appliquant successivement RCXP à l'ATS calculé par RCXP lui-même

présente l'avantage de ne pas devoir concevoir un prédicat de pertinence autorisant trop d'événements à se déclencher. Si un premier cycle à exécuter implique un ensemble d'événements et qu'un autre cycle à exécuter implique un ensemble différent d'événements, le prédicat de pertinence sera probablement différent dans les deux cas. Si l'ingénieur utilisait un seul prédicat de pertinence, les événements à déclencher pour le second cycle qui pourraient également être déclenchés à partir des états atteints lors de l'exploration du premier cycle seraient déclenchés pour les deux cycles. Cela impliquerait donc l'instanciation de plus de transitions concrètes que nécessaire. Ainsi, l'application successive de RCXP est à préférer.

Lorsqu'un algorithme est appliqué, les tests sont automatiquement générés après le calcul de la sous-approximation. Il ne reste donc plus au testeur qu'à appliquer les tests à l'implémentation du système et à vérifier que les tests soient tous passants.

7.2/ ÉTAPES PRINCIPALES DE LA MÉTHODE

Dans cette section, on présente les différentes étapes successives à réaliser dans le cadre de la méthode proposée. On suppose que l'ingénieur de test a conçu un modèle de l'implémentation à partir du cahier des charges de l'application. Pour générer des tests, il devra d'abord définir et décrire un objectif de test.

Puis, pour chaque objectif de test déterminé lors de cette première phase, on propose d'appliquer les étapes décrites dans les sections suivantes, à savoir :

- l'utilisation de la garde des événements apparaissant dans l'objectif de test comme prédicats d'abstraction,
- le calcul du système abstrait issu de ces prédicats d'abstraction et le calcul d'une sous-approximation de cette abstraction,
- identifier les états abstraits et les transitions abstraites non couverts par la sous-approximation,
- identifier un sous-ensemble des transitions précédemment identifiées qu'on souhaite effectivement couvrir,
- concevoir un prédicat de pertinence susceptible de guider la complétion de la première sous-approximation afin de couvrir le sous-ensemble précédemment constitué,
- calculer une nouvelle sous-approximation avec l'algorithme RCXP à partir de la première sous-approximation et du prédicat de pertinence élaboré,
- générer les tests en appliquant l'algorithme du postier chinois à partir des états concrets initiaux de la sous-approximation.

7.2.1/ MODÉLISATION DES EXIGENCES COMPORTEMENTALES (OBJECTIF DE TEST PRIMAIRE)

Le testeur doit avant tout exprimer ses besoins en termes d'objectifs de test : il doit exprimer les comportements spécifiques du système que les tests générés devront viser. Ces comportements seront exprimés sous forme d'exigences comportementales, c'est-à-dire de propriétés temporelles que le système doit vérifier. Ces propriétés temporelles seront alors considérées comme les objectifs de test *primaires* (ils pourront être complétés par

des objectifs de test dits *secondaires* par la suite).

Les formules temporelles instancient des schémas tels que ceux proposés par [Dwyer et al., 1999] pour spécifier des propriétés sur des systèmes finis.

Exemple 8 : Objectif de test pour l'exemple fil rouge

Dans le cas du distributeur de café, l'objectif de test (OT) suivant a été défini et utilisé :

$$OT_1 \stackrel{def}{=} \textbf{Jamais } takePot \textbf{ entre } autoOut \textbf{ et } powerDown$$

Cette propriété temporelle spécifie que si une erreur est survenue, la machine doit être mise hors service (*Status* := *off*) avant que le technicien ne puisse récupérer le contenu du *Pot*.

7.2.2/ EXTRACTION DES PRÉDICATS D'ABSTRACTION

Une fois l'objectif de test déterminé, l'extraction des prédicats d'abstraction est simple : elle consiste à calculer la garde des événements apparaissant dans cet objectif de test. Chaque garde constituera alors un prédicat d'abstraction. Les états abstraits calculés à partir des prédicats d'abstraction ainsi définis correspondront donc aux situations dans lesquelles tout ou partie des événements apparaissant dans l'objectif de test sont déclenchables et non déclenchables. On note que d'autres méthodes permettant d'extraire des prédicats d'abstraction à partir d'un objectif de test ont été proposées dans [Bride et al., 2016a].

Exemple 9 : Extraction des prédicats d'abstraction pour l'exemple fil rouge

Dans le cas du distributeur de café, les prédicats d'abstraction extraits à partir de l'objectif de test OT_1 correspondent à ceux présentés dans l'exemple 2 page 27, soit (pour rappel) :

- $p_0 \stackrel{def}{=} \texttt{grd}(takePot) \stackrel{def}{=} Status = off \wedge Pot \geq MAX\_POT - 50$
- $p_1 \stackrel{def}{=} \texttt{grd}(autoOut) \stackrel{def}{=} Status = on$
- $p_2 \stackrel{def}{=} \texttt{grd}(powerDown) \stackrel{def}{=} (Status = on \wedge AskChange = false \wedge AskCof = false \wedge Balance = 0) \vee Status = error$

Ainsi, les états abstraits dans lesquels p_0 est vrai (faux respectivement) seront les états dans lesquels l'événement *takePot* (dont p_0 est la garde) sera déclenchable (non déclenchable respectivement). Le principe est similaire pour p_1 et l'événement *autoOut* (dont p_1 est la garde) et pour p_2 et l'événement *powerDown* (dont p_2 est la garde).

7.2.3/ CALCUL D'UN ATS PAR L'ALGORITHME CXP

Lorsque les prédicats d'abstraction ont été extraits d'un objectif de test, l'application de l'algorithme CXP (ou CXP_{ASO}) est possible. Elle permet d'obtenir une sous-approximation contenant une instance de chaque état abstrait et chaque transition abstraite au moins une fois. L'intégralité du MTS associé au système événementiel et à l'ensemble de prédicats d'abstractions est donc connue.

7.2.4/ COMPLÉTION ÉVENTUELLE DE L'ATS PAR L'ALGORITHME BCI

En supposant que CXP calcule également les modalités des transitions, BCI peut ensuite être appliqué à l'ATS calculé par CXP. Cette étape ne devrait être effectuée que si certaines transitions non couvertes par CXP sont des *must*-transitions, puisque BCI n'instancie que celles-ci. Dans les étapes suivantes, on considère que l'ATS utilisé est celui de BCI (et non pas celui de CXP) si BCI a effectivement été appliqué. De plus, seule l'étape de génération de tests est nécessaire si BCI est parvenu a couvrir tous les états et toutes les transitions que le testeur souhaitait couvrir.

7.2.5/ IDENTIFICATION DES ÉTATS ABSTRAITS ET DES TRANSITIONS ABSTRAITES NON COUVERTS

Grâce à l'heuristique de coloration de l'algorithme CXP, les états abstraits et les transitions abstraites couverts sont connus, de même que les états abstraits et les transitions abstraites non couvertes.

- Les états abstraits couverts sont ceux pour lesquels il existe une instance *verte* dans l'ATS.
- Les transitions abstraites couvertes sont celles pour lesquelles il existe une instance dont l'état source est *vert* dans l'ATS.
- Tous les autres états abstraits et transitions abstraites sont non couverts par l'ATS.

7.2.6/ SÉLECTION D'UN SOUS-ENSEMBLE DES TRANSITIONS NON COUVERTES À COUVRIR (OBJECTIF DE TEST SECONDAIRE)

Parmi les transitions abstraites non couvertes par l'ATS, il peut en exister des non atteignables sur le système réel. Le testeur, avec sa connaissance du système, doit identifier les états et transitions dont il pense qu'ils sont effectivement atteignables et qu'il souhaite couvrir parmi celles non couvertes par l'ATS. Cet ensemble de transitions à couvrir constitue un objectif de test dit *secondaire* que le prédicat de pertinence visera à atteindre.

Exemple 10 : Application à l'exemple fil rouge

Dans le cas du distributeur de café, la transition $q_2 \xrightarrow{serveCof} q_1$ a été retenue comme objectif de test secondaire (on rappelle qu'elle correspond au service du dernier café restant dans la machine).

7.2.7/ DÉRIVATION D'UN PRÉDICAT DE PERTINENCE VISANT L'OBJECTIF DE TEST SECONDAIRE

Une fois les transitions à cibler par les tests identifiées, le testeur doit, avec sa connaissance du système, dériver un prédicat de pertinence permettant de les atteindre et autorisant le déclenchement d'un minimum d'événements (afin de limiter l'espace d'états obtenu par l'exploration).

Exemple 11 : Application à l'exemple fil rouge

Comme cela a déjà été dit, le service du dernier café requiert l'achat de tous les autres cafés par un utilisateur et leur service par la machine. Le prédicat de pertinence ainsi dérivé est :

$$pert_X \stackrel{def}{=} (Balance' > Balance) \vee (AskCof = false \wedge AskCof' = true) \vee (CofLeft' < CofLeft)$$

7.2.8/ APPLICATION DE L'ALGORITHME RCXP POUR COMPLÉTER LE PREMIER ATS

Une fois le prédicat de pertinence $pert_X$ dérivé de l'objectif de test secondaire, l'algorithme RCXP peut être appliqué au premier ATS en vue de le compléter avec des instances de transitions ciblant les transitions de cet objectif secondaire. Si le prédicat de pertinence autorise le déclenchement des événements adéquats, les transitions de l'objectif de test seront alors couvertes par l'ATS complété par RCXP.

7.2.9/ APPLICATION DE LA MÉTHODE À TOUS LES OBJECTIFS DE TEST SECONDAIRES

Les étapes 7.2.6, 7.2.7 et 7.2.8 sont à répéter jusqu'à ce que l'ATS obtenu avec RCXP satisfasse ses attentes en termes de couverture d'états abstraits et de transitions abstraites. En effet, si les objectifs de test secondaires ne sont pas tous couverts avec le prédicat de pertinence élaboré, cela signifie que ce dernier ne suffit pas à guider l'exploration vers chacun d'eux. Par conséquent, il convient de reprendre l'ATS obtenu lors de la première application de RCXP, de concevoir un nouveau prédicat de pertinence visant un nouveau sous-ensemble des objectifs secondaires et de répéter ce procédé tant que tous ne sont pas couverts.

7.2.10/ GÉNÉRATION DE TESTS PAR L'ALGORITHME DU POSTIER CHINOIS

Une fois qu'un ATS satisfaisant pour l'utilisateur est obtenu, la génération des tests se fait par l'algorithme du postier chinois. L'algorithme du postier chinois (qui, une fois le graphe préparé, peut être exécuté en temps polynomial) effectue un parcours de graphe (ici la partie concrète de l'ATS) en visitant tous les états et toutes les transitions au moins une fois tout en évitant au maximum de repasser par ceux et celles ayant déjà été visité(e)s. Chaque séquence de transitions unique visitée par cet algorithme depuis un état initial concret de l'ATS constitue alors un cas de test. L'algorithme du postier chinois favorise ainsi l'apparition d'un minimum de cas de test, ce qui dans le domaine du test est généralement recherché afin d'éviter l'instanciation répétée de séquences d'initialisation coûteuses par exemple.

8

IMPLÉMENTATION ET ÉVALUATION EXPÉRIMENTALE

Les solutions constituant les contributions de cette thèse ont été en grande partie guidés par les résultats expérimentaux obtenus. Les algorithmes implémentés visent à permettre la génération de tests couvrant tous les états et toutes les transitions d'abstractions de modèles judicieusement choisies relativement à des objectifs de tests. Des expérimentations ont donc été menées sur chacun des algorithmes afin d'évaluer et de comparer leurs performances et leur efficacité vis-à-vis de ce problème en particulier.

Les expérimentations ont d'abord été effectuées sur CXP avec et sans heuristique. Cette étude comparative ainsi que l'algorithme lui-même sont l'objet de la première publication [Julliand et al., 2017a] en rapport avec cette thèse. Les résultats ayant montré qu'en général CXP ne couvre pas tous les états abstraits ni toutes les transitions abstraites, la première piste étudiée pour les couvrir à été celle de l'instanciation des chaînes de Ball initialement introduites dans [Ball, 2005]. En effet, l'intégration du calcul des modalités des transitions nécessaire au calcul des chaînes de Ball dans CXP s'avère simple à mettre en œuvre et n'augmente pas sa complexité.

La variante de CXP appelée CXP_{ASO}, vise à éviter le recours à des heuristiques d'ordonnancement (notamment l'heuristique d'ordonnancement des états abstraits). Cependant, si CXP_{ASO} produit généralement de meilleurs taux de couverture des états abstraits et des transitions abstraites que CXP, tous ces états et ces transitions ne sont toujours pas couverts pour autant, ce qui justifie le recours aux chaînes de Ball.

L'algorithme conçu pour instancier les chaînes de Ball, l'algorithme BCI, est le deuxième algorithme ayant fait l'objet d'une publication [Julliand et al., 2017c] lors de cette thèse. Il tire parti des informations d'atteinte fournies par l'ATS calculé par CXP : il utilise la coloration des états concrets afin de maximiser la taille des chaînes de Ball instanciées. Les expérimentations effectuées sur BCI ont montré que les chaînes de Ball instanciées n'augmentaient pas nécessairement la couverture déjà obtenue par CXP. Cependant, l'analyse de ces expérimentations a permis de mettre en avant la nécessité, pour une grande majorité des cas d'étude utilisés dans cette thèse, d'instancier des cycles de transitions abstraites plusieurs fois avant de pouvoir atteindre les états permettant de couvrir de nouvelles transitions abstraites.

Pour palier ce problème, l'algorithme RCXP s'inspire du concept de prédicat de pertinence introduit sous les termes de « relevance function » dans [Grieskamp et al., 2002]. Cet algorithme fait l'objet de la troisième publication [Julliand et al., 2018b] en relation avec cette thèse. Il autorise, par le biais du prédicat de pertinence, l'exploration et l'ins-

tanciation répétée des cycles de transitions nécessaires à l'amélioration de la couverture des états et des transitions abstraites. Les résultats montrent que RCXP rend la couverture de la totalité des états et des transitions abstraites possible tout en limitant le nombre d'instanciations à effectuer et en garantissant la finitude de cette exploration.

Afin de mener toutes ces expérimentations, un outil nommé STRATEST permettant d'appliquer tous les algorithmes CXP, CXP_{ASO}, BCI et RCXP à des modèles de systèmes événementiels a été développé. Ce chapitre fournit en section 8.1 une explication du fonctionnement de STRATEST et présente le protocole expérimental et les différents modèles utilisés pour mener les expérimentations en section 8.2. La section 8.3 propose une analyse des résultats expérimentaux obtenus à partir des différents algorithmes. Enfin, la section 8.4 conclut sur ces résultats.

8.1/ FONCTIONNALITÉS ET IMPLÉMENTATION DE L'OUTIL STRATEST

Cette section décrit en 8.1.1 les fonctionnalités offertes par l'outil de génération de tests STRATEST et présente certains éléments importants de son implémentation en 8.1.2.

8.1.1/ FONCTIONNALITÉS PRINCIPALES DE STRATEST

STRATEST est un outil permettant de générer de manière entièrement automatisée des tests pour un système événementiel à partir d'un modèle de ce système, d'un ensemble de prédicats d'abstraction et éventuellement d'un prédicat de pertinence. La section 8.1.1.1 explique les fonctionnalités fournies par STRATEST vis-à-vis de la génération d'ATS et de tests par l'application des différents algorithmes constituant les contributions de cette thèse. La section 8.1.1.2 présente l'interface de programmation applicative (« Application Programming Interface » en Anglais) proposée par STRATEST et permettant l'utilisation des méthodes formelles.

8.1.1.1/ GÉNÉRATION D'ATS ET DE TESTS

Afin de générer des tests pour un système événementiel, le testeur peut choisir le ou les algorithme(s) à appliquer :

1. CXP ou CXP_{ASO} uniquement
2. CXP ou CXP_{ASO} puis BCI
3. CXP ou CXP_{ASO} puis RCXP

Une fois ces paramètres d'entrée renseignés, STRATEST se charge de faire l'analyse syntaxique et d'extraire la sémantique du modèle, des prédicats d'abstraction et du prédicat de pertinence (dans le cas où RCXP doit être appliqué) puis applique automatiquement les différents algorithmes demandés. Les ATS calculés par les différents algorithmes appliqués sont ensuite sauvegardés dans différents fichiers de rapport dont les plus importants sont les suivants.

- Un fichier contenant un rapport détaillé sur l'ATS indiquant notamment :
  - le nombre des états et des transitions abstraits du MTS,
  - les taux de couverture des états et des transitions abstraits,
  - le nombre d'instance d'états et de transitions calculés,
  - les taux de couverture des transitions ciblées par RCXP si cet algorithme à été appliqué,
  - la liste des états et des transitions abstraits couverts et non couverts,
  - ...
- Des fichiers représentant visuellement les graphes correspondant au MTS et au CTS composant l'ATS (avec coloration des états et des transitions concrets),
- Un fichier listant les différents cas de test en explicitant les transitions appliquées par chaque pas de test.

8.1.1.2/ INTERFACE DE PROGRAMMATION APPLICATIVE POUR LES MÉTHODES FORMELLES

STRATEST agit également comme une interface de programmation applicative facilitant la mise en œuvre de méthodes formelles sur les systèmes événementiels. Il propose notamment les éléments suivants.

- Une interface simple de communication avec le solveur SMT Z3 ([de Moura et al., 2008]) en langage objet, permettant de vérifier directement la satisfiabilité de n'importe quelle formule du premier ordre. Cette interface ne nécessite aucune connaissance ou manipulation du langage natif SMT-Lib2 ([Barrett et al., 2017]) utilisé par la majorité des solveurs SMT.
- Des fonctions permettant le calcul direct de la modalité d'une transition (may, $must^{-}$, $must^{+}$, $must^{\#}$).
- Un parseur de modèles B événementiel exprimés dans un format XML simple à prendre en main et à utiliser et dont la syntaxe est librement accessible en ligne à l'adresse suivante : https://github.com/stratosphr/stratestx. Ce parseur retourne un objet contenant toutes les informations liées à ce modèle :
  - la liste des variables et des fonctions déclarées ainsi que leurs domaines,
  - l'invariant du système,
  - la substitution d'initialisation,
  - la liste des événements sous forme d'objet dont les méthodes permettent le calcul direct de la garde et du prédicat avant-après.
- Une implémentation de l'algorithme du postier chinois [Thimbleby, 2003] applicable à un graphe fortement connexe.
- Une implémentation de l'algorithme de Tarjan [Tarjan, 1971] de détection des composantes fortement connexes dans un graphe (appelé simplement « algorithme de Tarjan » par la suite).

8.1.2/ ÉLÉMENTS D'IMPLÉMENTATION DE STRATEST

STRATEST est développé dans le langage de programmation orienté objet JAVA dans sa version 8. Il est constitué d'environ 150 classes et 10000 lignes de code.

L'outil utilise des implémentations de l'algorithme de Tarjan [Tarjan, 1971] et du postier chinois [Thimbleby, 2003] pour générer les tests une fois qu'un ATS a été calculé. L'al-

gorithme du postier chinois procède à un parcours de toutes les transitions d'un graphe fortement connexe tout en minimisant le nombre de fois ou ces transitions sont parcourues.

Les CTS calculés par les différents algorithmes n'étant pas nécessairement fortement connexes, des transitions fictives y sont ajoutées pour les rendre fortement connexes. La méthode employée pour rendre un CTS fortement connexe est celle présentée en détail dans la thèse de Pierre-Christophe Bué [Bue, 2011]. Cette méthode a été complétée pour prendre en compte des cas particuliers qui n'avaient pas été traités, notamment lorsqu'une composante fortement connexe apparaît dans le CTS. C'est la raison pour laquelle le recours à l'algorithme de détection des composantes fortement connexes de Tarjan est nécessaire.

Une fois le graphe rendu fortement connexe, l'algorithme du postier chinois peut être appliqué et les chemins parcourus par cet algorithme constituent chacun un test.

8.2/ CAS D'ÉTUDE ET PROTOCOLE EXPÉRIMENTAL

Cette section présente d'abord informellement les différents cas d'étude utilisés pour mener les expérimentations. Le protocole expérimental mis en place ainsi que les différentes mesures effectuées afin de mettre en évidence les apports de chaque algorithme sont ensuite décrits.

8.2.1/ CAS D'ÉTUDE

Les cinq cas d'étude sur lesquels les algorithmes ont été appliqués sont décrits ici. Les modèles en langage B événementiel peuvent être consultés en ligne à l'adresse suivante : https://github.com/stratosphr/stratestx. Tous les systèmes étudiés sont paramétrés, en ce sens qu'ils utilisent tous des constantes dont la modification influe sur la taille de l'espace d'états du LTS sémantique du système. De manière générale pour ces cinq cas d'étude, plus les valeurs de ces constantes sont élevées, plus la taille de l'espace d'états sera élevée. La modification de ces constantes servira par la suite à tester les différents algorithmes sur des systèmes de grande taille.

8.2.1.1/ DISTRIBUTEUR DE CAFÉ

Le distributeur de café (noté **CM**) est celui utilisé comme exemple fil rouge, publié dans [Julliand et al., 2017c] et [Julliand et al., 2018b].

Le paramètre principal de ce système est le nombre de cafés initialement présents dans le distributeur. Deux autres paramètres peuvent cependant être utilisés : la quantité d'argent maximale dans le pot ainsi que la quantité d'argent maximale que l'utilisateur peut insérer dans le distributeur avant de devoir acheter un café ou récupérer cet argent.

8.2.1.2/ SYSTÈME ÉLECTRIQUE

Le système électrique (noté **ELEC** par la suite et publié dans [Julliand et al., 2017a]) simule un système composé d'un certain nombre de batteries (au moins deux) ayant pour objectif d'alimenter en permanence un dispositif critique.

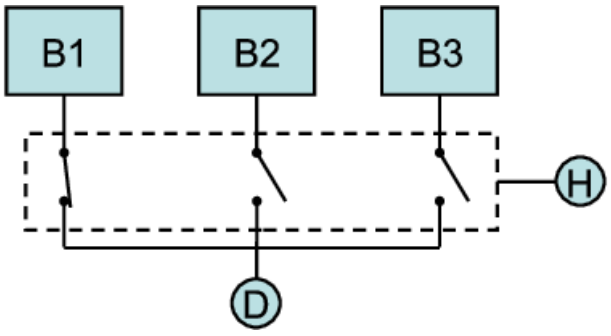


FIGURE 8.1 – Représentation du système **ELEC**

A un instant donné, une seule batterie alimente ce dispositif. Cette batterie est désignée par un interrupteur qui commute sur une autre batterie non déchargée lors d'un signal d'horloge. Si toutes les batteries autres que la batterie active sont déchargées, la batterie active est conservée. A tout moment, le rechargement d'une batterie par un technicien ou sa réparation lorsqu'elle est hors d'usage est possible et le modèle considère que le système ne se trouve jamais dans un cas où toutes les batteries sont déchargées ou hors d'usage.

Ce système est paramétré par le nombre de batteries utilisées par le système. Il est représenté par la figure 8.1 sur laquelle B1, B2 et B3 sont les trois batteries, H correspond à l'horloge et D correspond au dispositif critique à alimenter.

8.2.1.3/ LIGNE DE MÉTRO 14

La ligne de métro 14 (notée **L14** par la suite) modélise le comportement d'un certain nombre de trains circulant sur des rails formant un anneau. Ce système n'a pas fait l'objet d'une publication mais modélise partiellement le fonctionnement de la ligne 14 du métro parisien dont la spécification utilise le langage B. Il est représenté par la figure 8.2 sur laquelle les chiffres de 0 à 8 représentent les stations et les flèches indiquent le sens de circulation des rames sur les rails.

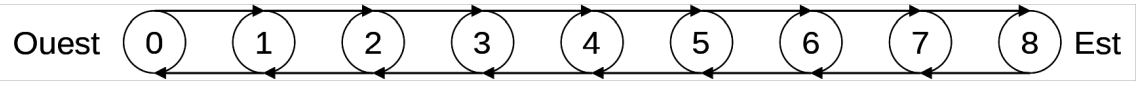


FIGURE 8.2 – Représentation du système **L14**

La ligne est composée de plusieurs stations auxquelles les trains doivent toujours s'arrêter avant de poursuivre vers la suivante. Lorsqu'un train arrive à une station et lorsqu'il change de direction, il doit ouvrir puis refermer ses portes afin de déposer et/ou prendre des passagers. Le changement de direction d'un train ne peut avoir lieu que lorsqu'il est arrivé à la dernière station de son trajet, ou lorsqu'il est revenu à la station de départ (après être arrivé à toutes les stations dans un sens puis dans l'autre). Deux

trains circulant dans la même direction ne peuvent pas se dépasser mais peuvent avoir la même position (se trouver à la même station) si l'un d'entre eux vient de quitter cette station mais n'est pas encore arrivé à la suivante. De même, deux trains circulant dans des directions opposées peuvent se trouver à la même station puisqu'ils se trouvent alors sur un côté différent de la ligne.

Le nombre de trains et de stations constituent les deux paramètres de ce système.
N.B. Même si ces deux paramètres ont généralement une valeur proche, le nombre de stations peut parfaitement être plus grand ou plus petit que le nombre de trains (puisque deux trains peuvent se trouver à la même station).

8.2.1.4/ ASCENSEUR

Comme son nom l'indique, l'ascenseur (noté **ELEV**) modélise un ascenseur desservant un certain nombre d'étages. Ce système n'a pas fait l'objet d'une publication mais s'apparente à la ligne 14 par son fonctionnement, la différence majeure étant que l'ascenseur peut changer de direction dans n'importe quelle position. Afin de simplifier le modèle, l'ascenseur s'arrête à chaque étage (tous les étages au dessus de lui sont desservis lorsqu'il monte, et tous les étages en dessous de lui sont desservis lorsqu'il descend). Lorsque l'ascenseur atteint un étage, les portes à l'étage s'ouvrent avant que celles de la cabine elle-même ne s'ouvrent. Après cela, les portes de la cabine doivent être refermées avant que les portes à l'étage ne le soient. L'ascenseur peut uniquement se déplacer lorsque les portes de la cabine ainsi que celles à l'étage ont été ouvertes puis refermées. Si l'ascenseur monte, il continue de monter jusqu'à ce que plus personne dans la cabine et aux étages supérieurs ne souhaite aller plus haut. De manière similaire, si l'ascenseur descend, il continue de descendre jusqu'à ce que plus personne dans la cabine et aux étages inférieurs ne souhaite aller plus bas.

Le nombre d'étages desservis par l'ascenseur constitue l'unique paramètre de ce système.

8.2.1.5/ GSM

Le GSM, noté **GSM** par la suite et publié dans [Bernard et al., 2004], modélise une partie du standard de communication GSM-11.11. La partie modélisée correspond à l'exploration et à la lecture de fichiers dans une arborescence (représentée par la figure 8.3) et possédant différents droits d'accès.

Cette arborescence est intégrée dans une carte SIM (pour « Subscriber Identity Module »). Le répertoire racine est appelé *MF* (pour « Master File ») contenant le répertoire *GSM* et le fichier *ICCID*. Le répertoire *GSM* contient à son tour les fichiers *LP*, *IMSI* et *AD*. Le fichier *ICCID* ne peut jamais être lu (il est créé par le fournisseur de la carte SIM et ne doit être utilisé par aucune application tierce). Le fichier *LP* peut toujours être lu sans qu'aucune authentification ne soit requise. Le fichier *IMSI* peut uniquement être lu si la carte SIM n'est pas bloquée et que l'utilisateur est authentifié (i.e. lorsqu'il a entré le bon code, ou PIN pour « Personal Identification Number »). Enfin, le fichier *AD* ne peut être lu que par l'autorité administrative appropriée et ne peut jamais l'être dans la partie du GSM-11.11 modélisée ici.

Pour que le fichier *IMSI* puisse être lu, le PIN correspondant à la carte SIM doit être

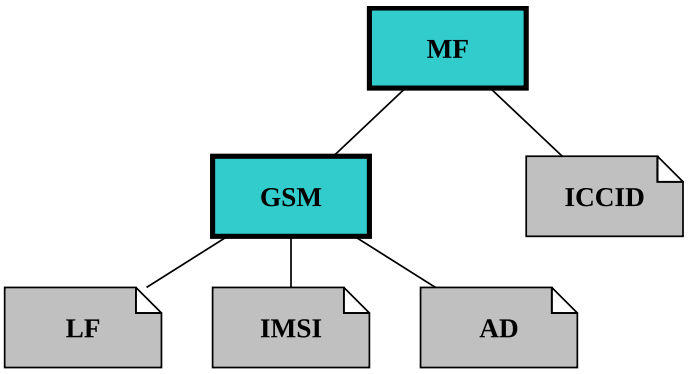


FIGURE 8.3 – Représentation de l'arborescence du système **GSM**

entré. Si l'utilisateur entre le bon PIN, le fichier *IMSI* peut être lu tant que la session dure. Lorsqu'une session est terminée, le PIN doit de nouveau être entré pour que le fichier *IMSI* puisse être lu de nouveau. Après un certain nombre de tentatives successives avec un mauvais PIN, la carte SIM est bloquée. Lorsqu'elle est bloquée, seul le PUK (pour « PIN Unlock Key ») correspondant à la carte SIM peut la débloquer. Après un certain nombre de tentatives successives avec un mauvais PUK, la carte SIM est définitivement inutilisable.

Les deux paramètres de ce système sont le nombre de tentatives successives d'entrée du PIN avant blocage de la carte SIM ainsi que le nombre de tentatives successives d'entrée du PUK avant blocage définitif de la carte SIM.

8.2.2/ PROTOCOLE EXPÉRIMENTAL ET MESURES EFFECTUÉES

Le protocole expérimental mis en place avant l'analyse des résultats ainsi que les mesures effectuées sur les ATS calculés par les algorithmes sont présentés ici.

8.2.2.1/ PROTOCOLE EXPÉRIMENTAL

Tous les algorithmes conçus dans le cadre de cette thèse ont été appliqués aux cinq cas d'étude présentés en section 8.2.1. Un algorithme effectuant l'exploration complète de l'espace d'états et donc l'instanciation de toutes les transitions possibles du système a également été appliqué à tous les cas d'étude. Cet algorithme, nommé FULL par la suite, procède simplement à une exploration en largeur de toutes les transitions applicables d'après le modèle du système à partir de tous ses états initiaux. Naturellement, cet algorithme ne termine pas dans le cas de systèmes infinis ou utilise trop de mémoire ou de temps dans le cas de systèmes de très grande taille. Par conséquent, il arrive que FULL ne produise pas d'ATS et que les différentes mesures effectuées sur les ATS calculés par les autres algorithmes ne puissent pas l'être dans le cas de FULL.

Les expérimentations présentées ont cependant été menées sur des systèmes paramétrés de telle sorte que FULL soit calculable dessus (sauf pour **GSM**), afin de mesurer les couvertures obtenues avec les différents algorithmes par rapport à ce qui peut effectivement être couvert. Les différents paramètres des systèmes ont ainsi été fixés à des valeurs suffisamment hautes pour mettre en évidence les qualités des algorithmes

par rapport à de telles méthodes. Le temps maximum imparti au calcul d'un ATS par un algorithme a arbitrairement été fixé à 72 heures.

On donne ci-dessous les valeurs des différents paramètres pour chaque cas d'étude :

1. Pour **CM**, les valeurs des paramètres sont les suivantes :
   - nombre de cafés initialement dans la machine : 9,
   - valeur maximale du pot : 450,
   - balance maximale de l'utilisateur : 200.
2. Pour **ELEC**, le nombre de batteries a été fixé à 8.
3. Pour **L14**, les valeurs des paramètres sont les suivantes :
   - nombre de trains circulant sur la ligne : 3,
   - nombre de stations sur la ligne : 4.
4. Pour **ELEV**, le nombre d'étages a été fixé à 4.
5. Pour **GSM**, les valeurs des paramètres sont les suivantes :
   - nombre de tentatives successives de renseignement du PIN avant blocage de la carte SIM : 3,
   - nombre de tentatives successives de renseignement du PUK avant blocage définitif de la carte SIM : 10.

Pour le système **CM**, on fera varier ces paramètres et les résultats associés pourront être observés sur la figure 8.6.

Pour chaque cas d'étude, la méthode de génération de tests présentée dans le chapitre 7 a été appliquée. Les tests calculés l'ont été sur tous les ATS obtenus, même intermédiaires. Plus précisément, les mesures ainsi que la génération de tests ont été effectuées sur les ATS calculés par CXP puis sur ceux calculés par RCXP, bien que ces derniers incluent l'ATS de CXP.

La machine utilisée pour le calcul des ATS ainsi que la génération des tests est en pratique plus puissante qu'un ordinateur classique car dotée d'un processeur INTEL® XEON® X5650 cadencé à 2.67 GHz et dispose de 16 GiO de mémoire vive.

8.2.2.2/ MESURES EFFECTUÉES

De nombreuses mesures, dont les résultats sont accessibles grâce aux rapports générés par STRATEST, ont été effectuées sur les ATS calculés par les algorithmes sur les cas d'étude. Dans les sections qui suivent, les résultats des mesures jugées les plus importantes pour évaluer la couverture des états et des transitions des MTS ainsi que l'efficacité des algorithmes sont présentés dans des tableaux. Les noms des colonnes de ces tableaux, ainsi que la mesure correspondante, sont donnés ci-dessous :

- **Sys.** : nom du système pour lequel les tests sont générés,
- **#Ev** : nombre d'événements du modèle,
- **#AP** : nombre de prédicats d'abstraction ($= |\mathcal{P}|$),
- **Alg.** : algorithme utilisé pour calculer l'ATS,
- **#AS** / **#AT** : nombre d'états / transitions abstrait(e)s dans le MTS,
- **%AS** / **%AT** : taux de couverture des états / transitions abstrait(e)s du MTS,
- **#AT**^{rel} : nombre d'états / transitions ciblé(e)s par le prédicat de pertinence,
- **%AT**^{rel} : taux de couverture des transitions abstraites ciblées par RCXP,

- **#CS** / **#CT** : nombre d'états / transitions concrets instancié(e)s,
- **#CS**$_{rchd}$ / **#CT**$_{rchd}$: nombre d'états / transitions concret(e)s instancié(e)s et atteint(e)s,
- **#Steps** : nombre de pas de tests générés.

Le nombre de transitions abstraites atteintes, notée **#AT**$_{rchd}$, n'est pas donné dans les tables de résultats. Elle peut s'approximer par le calcul $\frac{\textbf{\%AT}\times\textbf{\#AT}}{100}$.

Un autre aspect intéressant mis en évidence par les résultats est la différence de ce qu'on pourrait qualifier de *rendement* des différents algorithmes. Afin de couvrir les états abstraits et les transitions abstraites, chaque algorithme instancie ces transitions abstraites. Les instances ainsi calculées n'étant pas toujours atteintes, on propose d'analyser, pour chaque algorithme, deux notions de rendement différentes exprimées par un nombre dans l'intervalle $[0..1]$ concernant les transitions abstraites et leurs instances :

1. $\mathcal{R}_{CT} \stackrel{def}{=} \frac{\textbf{\#CT}_{rchd}}{\textbf{\#CT}}$: Le rapport entre le nombre d'instances de transitions atteintes dans les tests (**#CT**$_{rchd}$) et le nombre d'instances de transitions calculées (**#CT**). Ce ratio indique en moyenne combien de transitions concrètes sont calculées pour qu'une transition concrète soit atteinte. Cette mesure permet d'apprécier la connexion des instances calculées avec la partie atteinte du CTS de l'ATS : plus $\mathcal{R}_{CT}$ est proche de 1, plus la proportion d'instances calculées et atteintes est proche de 100%.
2. $\mathcal{R}_{AT} \stackrel{def}{=} \frac{\textbf{\#AT}_{rchd}}{\textbf{\#CT}}$: Le rapport entre le nombre de transitions abstraites atteintes (**#AT**$_{rchd}$) et le nombre d'instances de transitions calculées (**#CT**). Ce ratio indique en moyenne combien chaque transition concrète couvre de transitions abstraites. Il permet donc d'apprécier la propension des instances calculées à couvrir des transitions abstraites : plus $\mathcal{R}_{AT}$ est proche de 1, plus chaque instance calculée tend à permettre à elle seule la couverture de la transition abstraite qu'elle instancie.

Enfin, dans le cas de l'évaluation expérimentale de BCI, on utilise également les colonnes suivantes indiquant le nombre de transitions de modalité $must$ dans le MTS :

- **#Must**$^{-}$: indique le nombre de $must^{-}$-transitions qui **ne sont pas** des $must^{\#}$-transitions.
- **#Must**$^{+}$: indique le nombre de $must^{+}$-transitions qui **ne sont pas** des $must^{\#}$-transitions.
- **#Must**$^{\#}$: indique le nombre de $must^{\#}$-transitions.

N.B. La colonne **#Must**$^{-}$ ne comptabilise pas les $must^{-}$-transitions qui ont également la modalité $must^{+}$. De même, la colonne **#Must**$^{+}$ ne comptabilise pas les $must^{+}$-transitions qui ont également la modalité $must^{-}$

8.3/ RÉSULTATS EXPÉRIMENTAUX ET ANALYSE COMPARATIVE DES ALGORITHMES

Cette section présente et analyse les résultats obtenus expérimentalement par application des différents algorithmes présentés dans cette thèse aux cinq cas d'étude présentés en section 8.2. Les résultats sont présentés de manière à comparer les résultats produits par les algorithmes entre eux : la section 8.3.1 compare CXP avec CXP_{ASO}, la section 8.3.2 compare CXP avec BCI, la section 8.3.3 compare CXP avec RCXP et FULL et

la section 8.3.4 compare RCXP avec RCXP_{ASO}.

8.3.1/ ANALYSE COMPARATIVE DES RÉSULTATS OBTENUS AVEC CXP ET CXP_{ASO}

La table 8.1 présente les résultats obtenus avec CXP et CXP_{ASO} sur les différents systèmes présentés précédemment afin de comparer les deux algorithmes.

TABLE 8.1 – Comparaison des résultats entre CXP et CXP_{ASO}

Sys.	#Ev	#AP	Alg.	#AS	%AS	#AT	%AT	$\mathcal{R}_{AT}$	#CS	$\#CS_{rchd}$	#CT	$\#CT_{rchd}$	$\mathcal{R}_{CT}$	#Steps
CM	11	3	CXP	5	80	21	57,14	0,36	40	11	33	12	0,36	21
			CXP_{ASO}	5	80	21	61,9	0,52	32	10	25	13	0,52	22
		3	CXP	4	50	23	30,43	0,23	41	6	30	7	0,23	11
			CXP_{ASO}	4	100	23	73,91	0,63	32	14	27	17	0,63	28
		11	CXP	14	57,14	54	44,44	0,31	87	21	78	27	0,35	53
			CXP_{ASO}	14	57,14	54	44,44	0,44	67	19	55	24	0,44	49
ELEC	4	2	CXP	3	66,67	9	66,67	0,40	23	6	15	6	0,40	11
			CXP_{ASO}	3	66,67	9	66,67	0,46	20	6	13	6	0,46	6
		2	CXP	3	66,67	11	54,55	0,35	25	6	17	7	0,41	9
			CXP_{ASO}	3	66,67	11	72,73	0,50	21	8	16	8	0,50	11
		4	CXP	6	66,67	17	70,59	0,41	39	14	29	16	0,55	23
			CXP_{ASO}	6	66,67	17	70,59	0,57	29	12	21	12	0,57	17
ELEV	11	3	CXP	4	75	27	44,44	0,31	56	13	39	12	0,31	16
			CXP_{ASO}	4	100	27	74,07	0,50	54	23	40	22	0,55	58
		2	CXP	4	50	38	34,21	0,25	67	14	51	13	0,25	32
			CXP_{ASO}	4	50	38	44,74	0,32	70	18	53	17	0,32	41
		3	CXP	5	40	33	24,24	0,20	55	9	41	8	0,20	9
			CXP_{ASO}	5	60	33	51,52	0,38	58	18	45	17	0,38	22
		11	CXP	119	26,89	755	11,92	0,11	1267	99	849	107	0,13	543
			CXP_{ASO}	119	41,18	755	28,61	0,25	1182	179	854	216	0,25	1106
L14	15	3	CXP	4	50	53	7,55	0,07	104	5	57	4	0,07	5
			CXP_{ASO}	4	50	53	18,87	0,16	108	11	61	10	0,16	11
		3	CXP	4	75	50	12	0,11	102	7	56	6	0,11	9
			CXP_{ASO}	4	75	50	24	0,20	101	13	59	12	0,20	15
		5	CXP	25	52	372	6,72	0,06	723	23	397	25	0,06	68
			CXP_{ASO}	25	100	372	43,82	0,42	533	108	389	163	0,42	490
GSM	16	3	CXP	8	25	92	18,48	0,16	157	17	109	17	0,16	38
			CXP_{ASO}	8	25	92	19,57	0,17	148	14	106	18	0,17	29
		3	CXP	6	33,33	72	23,61	0,19	128	16	89	17	0,19	43
			CXP_{ASO}	6	33,33	72	25	0,21	122	15	87	18	0,21	35
		16	CXP	18	16,67	153	15,69	0,14	243	20	177	24	0,14	60
			CXP_{ASO}	18	29,63	153	22,61	0,11	381	39	309	91	0,29	182
Moyennes	11,40	4,81	CXP	14,50	51,83	107,82	31,19	0,22	191,82	17,18	124,94	18,35	0,23	56,24
			CXP_{ASO}	14,50	61,84	107,82	44,82	0,35	180,35	30,47	130,65	39,65	0,37	125,47

COMPARAISON DES TAUX DE COUVERTURE DE CXP ET CXP_{ASO}

Sur les systèmes étudiés ici, on constate que CXP_{ASO} produit toujours au moins les mêmes taux de couverture des états abstraits et des transitions abstraites que CXP. Ceci est exprimé dans les colonnes **%AS** et **%AT** pour lesquelles les lignes CXP_{ASO} indiquent toujours un taux de couverture supérieur ou égal à celui de la ligne CXP. Les taux de couverture des états abstraits obtenus avec CXP_{ASO} sont supérieurs à ceux obtenus avec CXP pour 7 cas sur 16 et atteignent 100% dans le cas de **L14** avec le $3^{ème}$ ensemble de prédicats d'abstraction. L'amélioration de la couverture des transitions abstraites par CXP_{ASO} par rapport à CXP est encore meilleure : dans 14 cas sur 16, la couverture des transitions abstraites est améliorée et est même multipliée par deux ou plus dans 6 cas sur 16. Aucun cas étudié n'a donc montré de taux de couverture des états abstraits ou des transitions abstraites inférieurs pour CXP_{ASO} par rapport à CXP.

Pour **ELEV** avec le $4^{ème}$ ensemble de prédicats d'abstraction et **L14** avec le $3^{ème}$ ensemble de prédicats d'abstraction, les taux de couverture des états abstraits ainsi que des

transitions abstraites sont *nettement* meilleurs avec $\mathrm{CXP}_{\mathrm{ASO}}$ qu'avec CXP. Ces résultats montrent expérimentalement que $\mathrm{CXP}_{\mathrm{ASO}}$ peut améliorer les taux de couverture même lorsque le MTS est composé de plusieurs dizaines d'états abstraits et de plusieurs centaines de transitions abstraites.

Naturellement, plus la couverture des états abstraits est élevée, plus la couverture des transitions abstraites a des chances d'être élevée : si un état abstrait n'est pas couvert, aucune transition abstraite dont la source ou la cible est cet état ne peut l'être. On remarque d'ailleurs de fortes améliorations de la couverture des transitions abstraites par $\mathrm{CXP}_{\mathrm{ASO}}$ chaque fois que ce dernier couvre plus d'états abstraits que CXP : c'est le cas par exemple pour le deuxième ensemble de prédicats d'abstraction de **CM**, pour tous les ensembles de prédicats d'abstraction de **ELEV** à part le $2^{ème}$ (où la couverture des états abstraits est inchangée) et également pour le $3^{ème}$ ensemble de prédicats d'abstraction de **L14**. Pour le $2^{ème}$ ensemble de prédicats d'abstraction de **CM** par exemple, la couverture des états abstraits atteint 100% avec $\mathrm{CXP}_{\mathrm{ASO}}$ contre seulement 50% avec CXP, et la couverture des transitions abstraites est plus de deux fois meilleure avec $\mathrm{CXP}_{\mathrm{ASO}}$ qu'avec CXP ($73,91\%$ de couverture avec $\mathrm{CXP}_{\mathrm{ASO}}$ contre seulement $30,43\%$ avec CXP).

COMPARAISON DU RENDEMENT DES DEUX ALGORITHMES

En ce qui concerne le rendement des deux algorithmes, on constate que les valeurs de $\mathcal{R}_{CT}$ et $\mathcal{R}_{AT}$ sont améliorées (plus proches de 1) dans tous les cas avec $\mathrm{CXP}_{\mathrm{ASO}}$. $\mathrm{CXP}_{\mathrm{ASO}}$ obtient un rendement $\mathcal{R}_{CT}$ moyen de $0,37$ contre $0,23$ pour CXP et un rendement $\mathcal{R}_{AT}$ moyen de $0,35$ contre $0,22$ pour CXP. Les résultats obtenus pour $\mathcal{R}_{CT}$ montrent donc que $\mathrm{CXP}_{\mathrm{ASO}}$ tend à instancier moins de transitions qui ne seront pas atteintes dans les tests que CXP. Les résultats obtenus pour $\mathcal{R}_{AT}$ montrent quant à eux, puisqu'ils sont souvent plus proches de 1 avec $\mathrm{CXP}_{\mathrm{ASO}}$ qu'avec CXP, que $\mathrm{CXP}_{\mathrm{ASO}}$ instancie moins de transitions superflues (qui ne seront pas atteintes dans les tests) que CXP.
Le rendement $\mathcal{R}_{AT}$ étant toujours inférieur à 0.5 pour CXP, cela signifie que CXP, qui instancie jusqu'à trois fois les transitions abstraites (contre une seule fois pour $\mathrm{CXP}_{\mathrm{ASO}}$), calcule souvent des instances de transitions qui ne seront pas connectées à la partie atteinte et ne permettront donc pas la couverture de la transition abstraite qu'elles instancient. $\mathrm{CXP}_{\mathrm{ASO}}$ effectue donc moins de travail d'instanciation que CXP pour des résultats en termes de taux de couverture toujours au moins aussi bons que ceux obtenus avec ce dernier.
N.B. Ces résultats en termes de rendement entre CXP et $\mathrm{CXP}_{\mathrm{ASO}}$ ne sont pas surprenants puisque CXP calcule justement souvent plus d'instances que $\mathrm{CXP}_{\mathrm{ASO}}$. Ces résultats sont cependant également liés au fait que CXP couvre généralement moins d'états abstraits et de transitions abstraites que $\mathrm{CXP}_{\mathrm{ASO}}$.

COMPARAISON DES COMPORTEMENTS DES DEUX ALGORITHMES AVEC DES MTS DE GRANDE TAILLE

On constate enfin que CXP et $\mathrm{CXP}_{\mathrm{ASO}}$ peuvent tous les deux traiter des MTS composés de plusieurs dizaines d'états abstraits et de plusieurs dizaines (et même plusieurs centaines) de transitions abstraites. En effet, sur les plus gros ensembles de prédicats d'abstraction de chacun des systèmes, les taux de couverture des états abstraits sont toujours supérieurs ou égaux à 50% avec les deux algorithmes. Les taux de couverture des tran-

sitions abstraites sur les plus gros MTS sont quant-à-eux proches de ceux obtenus avec des MTS plus petits. Ces taux de couverture sont identiques à ceux obtenus avec le $2^{ème}$ ensemble de prédicats d'abstraction, malgré le fait que ce dernier contienne 8 prédicats de moins. Cela montre que les taux de couverture ne sont pas grandement affectés lorsque le nombre de transitions abstraites dans le MTS est grand.

PAS DE TESTS CALCULÉS

On constate que lorsque le nombre de pas de tests (**#Steps**) est plus grand avec $\mathrm{CXP}_{\mathrm{ASO}}$, cette augmentation se traduit également toujours par une augmentation des taux de couverture des états abstraits ou des transitions abstraites par rapport à CXP. Pour le système **CM** avec le $2^{ème}$ ensemble de prédicats d'abstraction par exemple, les tests générés à partir de l'ATS calculé avec CXP sont constitués de 11 pas de tests contre 28 avec l'ATS calculé par $\mathrm{CXP}_{\mathrm{ASO}}$. Mais les taux de couverture des états abstraits passent de 50% avec CXP à 100% avec $\mathrm{CXP}_{\mathrm{ASO}}$ et les taux de couverture des transitions abstraites passent de $30,43\%$ à $73,91\%$.

Lorsque le nombre de pas de tests est égal ou même inférieur avec $\mathrm{CXP}_{\mathrm{ASO}}$ par rapport à CXP, les taux de couverture obtenus pour les états abstraits et les transitions abstraites sont identiques. Pour **ELEC** avec le troisième ensemble de prédicats d'abstraction par exemple, 17 pas de tests sont obtenus à partir de l'ATS calculé par $\mathrm{CXP}_{\mathrm{ASO}}$ contre 23 avec CXP, mais les taux de couverture des états abstraits $(66,67\%)$ et des transitions abstraites $(70,59\%)$ sont identiques pour les deux algorithmes.

Ces résultats penchent en faveur des tests obtenus à partir de l'ATS calculé par $\mathrm{CXP}_{\mathrm{ASO}}$:

- soit le nombre de pas de tests est sensiblement plus grand avec $\mathrm{CXP}_{\mathrm{ASO}}$, donc généralement plus coûteux à instancier que ceux obtenus avec CXP, mais au bénéfice d'une meilleure couverture des états abstraits et/ou des transitions abstraites,
- soit le nombre de pas de tests est proche, donc globalement aussi coûteux à instancier sur le système réel que CXP, pour les mêmes taux ou des taux meilleurs de couverture des états abstraits et des transitions abstraites,
- soit le nombre de pas de tests est plus petit avec $\mathrm{CXP}_{\mathrm{ASO}}$, donc généralement moins coûteux à instancier que ceux obtenus avec CXP, pour les mêmes taux de couverture des états abstraits et des transitions abstraites.

En résumé, dans le premier cas, on perd sur le nombre de pas de test mais au profit d'une meilleure couverture. Dans le dernier cas, on gagne sur les deux critères : le nombre de pas de test moindre et taux de couverture au moins aussi bons.

COMPARAISON DES TEMPS DE CALCUL DES DEUX ALGORITHMES

Pour tous les cas d'étude, les ATS sont calculés en quelques secondes par CXP et $\mathrm{CXP}_{\mathrm{ASO}}$. Ces temps de calcul sont quasiment identiques, et ce quel que soit l'ensemble de prédicats d'abstraction considéré.

Les temps de calcul des tests correspondant à un parcours en profondeur (donc de complexité linéaire) des transitions constituant les CTS, ceux-ci sont également de l'ordre de la seconde, même pour les plus gros CTS.

CONCLUSION

Globalement, les résultats présentés dans la table 8.1 confirment expérimentalement l'hypothèse que l'ordonnancement automatique et dynamique utilisé par $\mathrm{CXP}_{\mathrm{ASO}}$ permet généralement d'obtenir de meilleurs taux de couverture que l'ordonnancement statique de CXP. Les moyennes des taux de couverture des états abstraits et des transitions abstraites sont en effet meilleures pour $\mathrm{CXP}_{\mathrm{ASO}}$ que pour CXP ($51,83\%$ pour CXP contre $61,84\%$ pour $\mathrm{CXP}_{\mathrm{ASO}}$ pour les états abstraits et $31,19\%$ pour CXP contre $44,82\%$ pour $\mathrm{CXP}_{\mathrm{ASO}}$ pour les transitions abstraites).

L'aspect statique de l'heuristique d'ordonnancement des états abstraits utilisée par CXP (voir l'heuristique 2. page 58) est la principale raison pour laquelle cette différence des taux de couverture apparaît : elle ne permet pas de tirer pleinement parti de la connaissance de l'ATS au cours de l'exploration. De plus, l'heuristique d'ordonnancement des états abstraits a l'inconvénient de devoir être déterminée par le testeur, tandis que $\mathrm{CXP}_{\mathrm{ASO}}$ ne requiert pas d'intervention de la part de ce dernier.

Les moyennes des différents rendements $\mathcal{R}_{AT}$ et $\mathcal{R}_{CT}$ sont également meilleurs pour $\mathrm{CXP}_{\mathrm{ASO}}$ que pour CXP ce qui confirme encore l'intérêt de l'heuristique d'ordonnancement dynamique. Pour ce qui est du nombre d'étapes de test calculées, ils sont en moyenne environ deux fois plus élevées pour $\mathrm{CXP}_{\mathrm{ASO}}$ que pour CXP. Cependant, l'apport de $\mathrm{CXP}_{\mathrm{ASO}}$ en termes de taux de couverture des états et des transitions de l'abstraction est plus important dans notre démarche que sa capacité à limiter le nombre de pas de tests issus de la sous-approximation.

8.3.2/ ANALYSE COMPARATIVE DES RÉSULTATS OBTENUS AVEC CXP ET BCI

La table 8.2 présente les résultats obtenus avec CXP et BCI.

TABLE 8.2 – Comparaison des résultats entre CXP et BCI

Sys.	#Ev	#AP	#Must^{-}	#Must^{+}	#Must$^{\#}$	Alg.	#AS	%AS	#AT	%AT	$\mathcal{R}_{AT}$	#CS	#CS$_{rchd}$	#CT	#CT$_{rchd}$	$\mathcal{R}_{CT}$	#Steps
CM	11	3	3	5	0	CXP	5	80	21	57,14	0,36	40	11	33	12	0,36	21
						BCI	5	80	21	57,14	0,35	41	12	34	13	0,37	22
		3	0	4	0	CXP	4	50	23	30,43	0,23	41	6	30	7	0,23	11
						BCI	4	50	23	30,43	0,18	43	8	33	10	0,28	18
		11	9	19	8	CXP	14	57,14	54	44,44	0,31	87	21	78	27	0,35	53
						BCI	14	57,14	54	44,44	0,23	104	27	104	34	0,33	49
ELEC	4	2	3	2	0	CXP	3	66,67	9	66,67	0,40	23	6	15	6	0,40	11
						BCI	3	66,67	9	66,67	0,32	26	6	19	6	0,32	11
		2	2	2	0	CXP	3	66,67	11	54,55	0,35	25	6	17	7	0,41	9
						BCI	3	66,67	11	54,55	0,32	25	8	19	9	0,44	11
		4	4	4	5	CXP	6	66,67	17	70,59	0,41	39	14	29	16	0,55	23
						BCI	6	66,67	17	70,59	0,29	43	15	41	12	0,51	33
ELEV	11	3	2	3	0	CXP	4	75	27	44,44	0,31	56	13	39	12	0,31	16
						BCI	4	100	27	48,15	0,30	60	15	43	14	0,33	22
		2	0	0	0	CXP	4	50	38	34,21	0,25	67	14	51	13	0,25	32
						BCI	4	50	38	34,21	0,25	67	14	51	13	0,25	32
		3	3	3	0	CXP	5	40	33	24,24	0,20	55	9	41	8	0,20	9
						BCI	5	40	33	24,24	0,18	58	10	45	10	0,22	13
		11	193	22	48	CXP	119	26,89	755	11,92	0,11	1267	99	849	107	0,13	543
						BCI	119	53,78	755	21,85	0,15	1447	112	1117	140	0,13	655
L14	15	3	2	1	0	CXP	4	50	53	7,55	0,07	104	5	57	4	0,07	5
						BCI	4	50	53	7,55	0,07	107	5	60	4	0,07	5
		3	2	1	1	CXP	4	75	50	12	0,11	102	7	56	6	0,11	9
						BCI	4	75	50	12	0,10	106	10	59	9	0,15	13
		5	0	0	0	CXP	25	52	372	6,72	0,06	723	23	397	25	0,06	68
						BCI	25	52	372	6,72	0,06	723	23	397	25	0,06	68
GSM	16	3	0	8	0	CXP	8	25	92	18,48	0,16	157	17	109	17	0,16	38
						BCI	8	25	92	18,48	0,15	164	21	116	20	0,17	42
		3	0	6	0	CXP	6	33,33	72	23,61	0,19	128	16	89	17	0,19	43
						BCI	6	33,33	72	23,61	0,18	132	19	94	19	0,20	44
		16	12	16	3	CXP	18	16,67	153	15,69	0,14	243	20	177	24	0,14	60
						BCI	18	16,67	153	15,69	0,13	247	22	181	26	0,14	62
Moyennes	11,40	4,81	14,69	6,00	4,06	CXP	14,50	51,83	107,82	31,19	0,22	191,82	17,18	124,94	18,35	0,23	56,24
						BCI	14,50	54,88	107,82	31,74	0,20	205,88	19,53	145,47	21,65	0,24	64,94

L'hypothèse ayant mené à la conception de l'algorithme BCI est la suivante : l'instanciation des chaînes de Ball complète potentiellement l'ATS calculé par CXP avec de nouvelles instances de transitions abstraites. Ces nouvelles instances permettraient d'améliorer les taux de couverture des états abstraits et des transitions abstraites à condition, d'une part, qu'elles soient atteintes depuis un état initial du CTS et, d'autre part, qu'elles instancient des transitions abstraites qui n'étaient pas couvertes par CXP.

ANALYSE DU NOMBRE DE *must*-TRANSITIONS IDENTIFIÉES

Le nombre de *must*-transitions identifiées sur les différents cas d'étude (la somme de **#Must**$^-$ et **#Must**$^+$) constitue en moyenne $22,4\%$ du nombre de *may*-transitions (**#AT**) dans le 3MTS. Cette proportion relativement faible se traduit par de faible chances d'identifier une *must*-transition qui n'ait pas encore été couverte par CXP, et des chances encore plus faibles de parvenir à concrétiser une telle $must^-$-transition par une instance atteinte permettant de la couvrir.

Le nombre de *must*-transitions identifiées dépend évidemment des prédicats d'abstraction utilisés et de leur nombre. On note qu'il est d'ailleurs possible de ne trouver aucune *must*-transition comme c'est le cas pour **ELEV** avec le $2^{ème}$ ensemble de prédicats d'abstraction et pour **L14** avec le $3^{ème}$ ensemble de prédicats d'abstraction.

COMPARAISON DES TAUX DE COUVERTURE DES ÉTATS ET DES TRANSITIONS ABSTRAITES ENTRE CXP ET BCI

La table 8.2 montre clairement que les taux de couverture des états abstraits et des transitions abstraites ne sont que très rarement améliorés avec BCI, même lorsque de nouvelles transitions sont instanciées. En effet, les colonnes **%AS** et **%AT** sont identiques avec CXP et BCI pour tous les cas d'étude et tous les ensembles de prédicats d'abstraction mis à part pour le système **ELEV** avec le 1^{er} et le $3^{ème}$ ensemble de prédicats d'abstraction.

Cette amélioration des taux de couverture des états abstraits et des transitions abstraites par BCI peut provenir de deux situations :

1. BCI concrétise par des instances atteintes des *must*-transitions qui n'étaient pas couvertes, et/ou

2. BCI concrétise des $must^-$-transitions par des instances atteintes et qui rejoignent des transitions concrètes dont la source est un état *bleu*. Si ces transitions concrètes sont des instances de transitions abstraites non couvertes par CXP, elles deviennent atteintes par la concrétisation opérée par BCI et sont donc couvertes.

Outre le cas particulier de **ELEV**, même pour les cas où le 3MTS possèdent beaucoup de *must*-transitions (typiquement lorsque les ensembles de prédicats d'abstraction sont les plus grands), BCI n'apporte aucune amélioration des taux de couverture.

COMPARAISON DU NOMBRE D'INSTANCES CONCRÉTISÉES ET ATTEINTES ENTRE CXP ET BCI

Malgré le fait que les améliorations des taux de couverture par BCI soient rares sur les cas d'étude, le nombre de transitions concrétisées (**#CT**) et le nombre de transitions atteintes (**#CT**$_{rchd}$) augmentent presque toujours avec BCI. Plusieurs situations permettent d'expliquer le fait que le nombre de transitions concrètes atteintes augmentent mais que les taux de couverture restent les mêmes :

1. Les transitions ayant la modalité *must* dans le 3MTS sont des transitions déjà couvertes avec CXP, auquel cas BCI ne permet jamais d'améliorer les taux de couverture puisque les transitions qu'il instancie sont justement ces *must*-transitions.
2. Les instances de *must*-transitions calculées par BCI ne sont pas atteintes, c'est-à-dire qu'aucun état vert calculé par CXP ne peut servir de source pour instancier les chaînes de *must*-transitions).
3. Les instances de *must*-transitions calculées par BCI reconnectent des transitions dont l'état source est *bleu* mais instanciant une transition abstraite déjà couverte. Ceci est rendu possible par le fait que lorsque CXP instancie une transition abstraite par une transition atteinte (et donc couvre cette transition), il ajoute également au CTS l'instance témoin dont il disposait au préalable et calculée pour prouver que cette transition abstraite avait la modalité *may*. Cette instance témoin n'est pas nécessairement atteinte car son état source n'est pas forcément un état connu.

En pratique après analyse des CTS obtenus, l'une ou plusieurs de ces situations se produisent sur tous les cas d'étude (sauf pour les cas particuliers de **ELEV** où les taux de couverture sont augmentés par BCI).

BCI instancie une fois toutes les *must*-transitions, c'est-à-dire toutes les $must^-$-transitions dont le nombre est la somme de **#Must**$^-$ et **#Must**$^\#$, et toutes les $must^+$-transitions dont le nombre est la somme de **#Must**$^+$ et **#Must**$^\#$. On rappelle que les colonnes **#Must**$^-$ et **#Must**$^+$ ne comptabilisent pas les $must^\#$-transitions. Ainsi, le nombre d'instances calculées par BCI peut être obtenu par le calcul suivant : **#Must**$^-$ + **#Must**$^\#$ + **#Must**$^+$ + **#Must**$^\#$. Dans le cas de l'exemple fil rouge (voir le premier ensemble de prédicats pour le système **CM** dans la table 8.2), BCI calcule donc $3 + 0 + 5 + 0 = 8$ instances de transitions. Pour le $3^{ème}$ ensemble de prédicats de ce même cas d'étude, BCI calcule $9 + 8 + 19 + 8 = 44$ instances de *must*-transitions.

Il peut paraître surprenant que la différence entre le nombre de transitions dans le CTS calculé par BCI (**#CT** à la ligne de BCI) et le nombre de transitions dans le CTS calculé par CXP (**#CT** à la ligne de CXP) ne soit pas toujours égal à la somme **#Must**$^-$ + **#Must**$^\#$ + **#Must**$^+$ + **#Must**$^\#$. Pour l'exemple fil rouge par exemple (le système **CM** avec le 1^{er} ensemble de prédicats d'abstraction), le CTS calculé par CXP contient 33 transitions, tandis que le CTS calculé par BCI n'en contient que 34, soit seulement une transition de plus au lieu des 8 attendues. Cela vient du fait que les appels au solveur dans BCI peuvent calculer des instances identiques à celles existant déjà dans le CTS calculé par CXP.

ANALYSE DES TEMPS DE CALCUL AVEC CXP ET BCI

Afin d'obtenir le 3MTS (plus précisément les modalités des transitions) CXP est contraint de tester pour chaque *may*-transition si elle possède la modalité $must^-$ et si elle possède la

modalité $must^+$. Pour **ELEV** et **GSM**, le solveur SMT utilisé se révèle assez lent (souvent plusieurs secondes pour **ELEV** et souvent plus de 10 minutes pour **GSM**) à déterminer ces modalités. Dans ces cas là, CXP peut donc prendre plusieurs minutes avant de retourner l'ATS (composé du 3MTS). **ELEV** avec le plus gros ensemble de prédicats d'abstraction est le plus long (environ 45 minutes) car le 3MTS contient de nombreuses transitions.

L'algorithme BCI termine quant à lui instantanément dans la plupart des cas, et en moins de 10 minutes sur tous les exemples. Le cas le plus long est également celui de **ELEV** avec le plus gros ensemble de prédicats d'abstraction et dure environ 8 minutes.

CONCLUSION

Les résultats expérimentaux ne montrent pas que BCI améliore de manière significative CXP en termes de couverture des états et des transitions du MTS. En effet, l'augmentation moyenne des taux de couverture des états et des transitions de l'abstraction avec BCI reste très faible (de $51,83\%$ à $54,88\%$ pour les états et de $31,19\%$ à $31,74\%$ pour les transitions). L'intérêt de BCI pour compléter la sous-approximation calculée par CXP n'est donc pas observé sur nos études de cas.

Une expérimentation plus poussée, qui n'a pas été menée dans le cadre de cette thèse, nécessiterait de faire varier les prédicats d'abstraction dans le but de changer les modalités des transitions du MTS et d'augmenter le nombre de $must$-transitions. Cependant, ces premiers résultats assez négatifs nous ont amenés à nous intéresser en priorité au concept de prédicat de pertinence.

8.3.3/ ANALYSE COMPARATIVE DES RÉSULTATS OBTENUS AVEC CXP, RCXP ET FULL

La table 8.3 présente les résultats obtenus avec CXP, RCXP et FULL. Pour rappel, FULL correspond au calcul de tous les états et de toutes les transitions du système modélisé. Dans le cas du **GSM**, son calcul n'a pas abouti dans la limite de temps fixée, raison pour laquelle certaines cellules de la ligne FULL de ce système ne sont renseignées qu'avec une borne inférieure (notamment sur le nombre d'états concrets et le nombre de transitions concrètes). De plus, puisque la génération de tests n'est pas applicable en temps raisonnable sur un système complet, le nombre de pas de tests (**#Steps**) n'est pas non plus renseigné pour la ligne FULL.

EXPLICATION DES TAUX DE COUVERTURE PARTIELS OBTENUS AVEC FULL

Il arrive que les taux de couverture des états abstraits (**%AS**) ou des transitions abstraites (**%AT**) n'atteignent pas 100% sur la ligne FULL. Cette situation se produit pour **ELEV** avec le $2^{ème}$ et le $4^{ème}$ ensemble de prédicats d'abstraction et pour **L14** avec le 1^{er}, le $2^{ème}$ et le $3^{ème}$ ensemble de prédicats d'abstraction. Cela signifie que certains états ou transitions sont may-atteignables, donc présents dans le MTS, mais ne sont pas atteignables dans le système concret (puisque FULL n'est pas parvenu à les couvrir). Les taux donnés pour dans les colonnes **%AS** et **%AT** portent sur la totalité des états et des transitions abstraites may-atteignables, et non pas sur les états et transitions abstraites effectivement atteignables sur le système. Pour **ELEV** avec le $2^{ème}$ ensemble de prédicats

TABLE 8.3 – Comparaison des résultats entre CXP, RCXP et FULL

Sys.	#Ev	#AP	Alg.	#AS	%AS	#AT	%AT	#AT^{rel}	%AT^{rel}	#CS	#CS$_{rchd}$	#CT	#CT$_{rchd}$	$\mathcal{R}_{CT}$	#Steps
CM	11	3	CXP	5	80	21	57,14	-	-	40	11	33	12	0,36	21
			RCXP	5	100	21	100	9	100,00	377	350	565	544	0,96	1006
			FULL	5	100	21	100	-	-	3586	3586	10952	10952	1,00	-
		3	CXP	4	50	23	30,43	-	-	41	6	30	7	0,23	11
			RCXP	4	100	23	91,3	16	87,50	336	303	405	383	0,95	2385
			FULL	4	100	23	100	-	-	3586	3586	10952	10952	1,00	-
		11	CXP	14	57,14	54	44,44	-	-	87	21	78	27	0,35	53
			RCXP	14	92,86	54	79,63	30	63,33	511	455	771	725	0,94	3149
			FULL	14	100	54	100	-	-	3586	3586	10952	10952	1,00	-
ELEC	4	2	CXP	3	66,67	9	66,67	-	-	23	6	15	6	0,40	11
			RCXP	3	100	9	100	3	100,00	63	55	61	56	0,92	103
			FULL	3	100	9	100	-	-	2048	2048	26112	26112	1,00	-
		2	CXP	3	66,67	11	54,55	-	-	25	6	17	7	0,41	9
			RCXP	3	100	11	100	5	100,00	134	122	154	148	0,96	486
			FULL	3	100	11	100	-	-	2048	2048	26112	26112	1,00	-
		4	CXP	6	66,67	17	70,59	-	-	39	14	29	16	0,55	23
			RCXP	6	100	17	100	5	100,00	115	95	121	111	0,92	165
			FULL	6	100	17	100	-	-	2048	2048	26112	26112	1,00	-
ELEV	11	3	CXP	4	75	27	44,44	-	-	56	13	39	12	0,31	16
			RCXP	4	100	27	74,07	15	53,33	200	168	257	237	0,92	1465
			FULL	4	100	27	100	-	-	18416	18416	101072	101072	1,00	-
		2	CXP	4	50	38	34,21	-	-	67	14	51	13	0,25	32
			RCXP	4	75	38	78,95	20	85,00	569	522	997	962	0,96	5048
			FULL	4	75	38	81,58	-	-	18416	18416	101072	101072	1,00	-
		3	CXP	5	40	33	24,24	-	-	55	9	41	8	0,20	9
			RCXP	5	100	33	63,64	25	52,00	173	133	200	170	0,85	865
			FULL	5	100	33	100	-	-	18416	18416	101072	101072	1,00	-
		11	CXP	119	26,89	755	11,92	-	-	1267	99	849	107	0,13	543
			RCXP	119	36,13	755	18,68	635	8,04	1690	540	1526	794	0,52	5037
			FULL	119	100	755	95,5	-	-	18416	18416	101072	101072	1,00	-
L14	15	3	CXP	4	50	53	7,55	-	-	104	5	57	4	0,07	5
			RCXP	4	100	53	90,57	45	97,78	3120	3033	4040	3993	0,99	15531
			FULL	4	100	53	92,45	-	-	8886	8886	24598	24598	1,00	-
		3	CXP	4	75	50	12	-	-	102	7	56	6	0,11	9
			RCXP	4	100	50	94	42	97,62	3105	3033	4032	3993	0,99	15531
			FULL	4	100	50	96	-	-	8886	8886	24598	24598	1,00	-
		5	CXP	25	52	372	6,72	-	-	723	23	397	25	0,06	68
			RCXP	25	100	372	80,11	338	80,77	3633	3034	4641	4322	0,93	16420
			FULL	25	100	372	97,31	-	-	8886	8886	24598	24598	1,00	-
GSM	16	3	CXP	8	25	92	18,48	-	-	157	17	109	17	0,16	38
			RCXP	8	25	92	39,13	75	25,33	363	222	412	320	0,78	1254
			FULL	8	-	92	-	-	-	-	-	-	-	-	-
		3	CXP	6	33,33	72	23,61	-	-	128	16	89	17	0,19	43
			RCXP	6	100	72	58,33	55	45,45	579	467	1029	957	0,93	3563
			FULL	6	-	72	-	-	-	-	-	-	-	-	-
		16	CXP	18	16,67	153	15,69	-	-	243	20	177	24	0,14	60
			RCXP	18	50	153	39,22	129	27,91	448	225	760	607	0,80	2300
			FULL	18	-	153	-	-	-	-	-	-	-	-	-
Moyennes	11,40	4,81	CXP	14,50	51,83	107,82	31,19	-	-	191,82	17,18	124,94	18,35	0,23	56,24
			RCXP	14,50	87,00	107,82	75,21	100,71	70,40	1 079,94	918,47	1 397,59	1 297,82	0,88	5 222,53
			FULL	14,50	$\geq$ 98,21	107,82	$\geq$ 96,81	-	-	> 9007,86	> 9007,86	> 43848	> 43848	1,00	-

d'abstraction par exemple, les taux obtenus avec FULL sont de 75% et $81,58\%$ pour les états abstraits et les transitions abstraites respectivement. RCXP atteint également 75% de couverture des états abstraits, ce qui signifie, puisque c'est le même taux que celui obtenu avec FULL, que tous les états *may*-atteignables *effectivement* atteignables sur le système réel modélisé sont couverts par RCXP. Pour ce qui est des $78,95\%$ de couverture des transitions abstraites obtenus avec RCXP, contre $81,58\%$ pour FULL, cela signifie que certaines transitions *may*-atteignables *effectivement* atteignables sur le système réel modélisé ne sont pas couvertes par RCXP, mais que ce dernier ne pourra de toute façon pas couvrir plus de $81,58\%$ des transitions *may*-atteignables puisque les autres ne sont pas atteignables sur le système réel.

COMPARAISON DES TAUX DE COUVERTURES ENTRE CXP ET RCXP

Le fait que RCXP couvre toujours au moins autant d'états abstraits et de transitions abstraites que CXP vient évidemment du fait que RCXP n'enlève jamais d'états ni de transitions à l'ATS obtenu avec CXP : il ne fait qu'en rajouter lorsqu'il instancie des transitions pertinentes. Ainsi, les taux de couverture des états abstraits (**%AS**) et des transitions abstraites (**%AT**) peuvent être au pire inchangés et au mieux améliorés. On remarque ce-

pendant que les taux de couverture des transitions abstraites sont toujours meilleurs avec RCXP qu'avec CXP en raison du fait que les prédicats de pertinence ont spécifiquement été conçus pour couvrir des transitions non couvertes par CXP.

Ainsi, la métrique la plus révélatrice de l'effet du prédicat de pertinence est donnée par la colonne **%AT**^{rel} qui indique les taux de couverture des transitions abstraites *cibles* (pertinentes) obtenus avec RCXP.
Dans 4 cas sur 16, toutes les transitions cibles ont été atteintes grâce aux prédicats de pertinence conçus dans ce but (voir les cellules dans la colonne **%AT**^{rel} ayant pour valeur 100). Pour ces 4 cas, le prédicat de pertinence visait (et est parvenu) à couvrir toutes les états abstraits et toutes les transitions abstraites. Ceci est explicité par les colonnes **%AS** et **%AT** qui valent toutes les deux 100% sur la ligne RCXP pour **CM** avec le 1^{er} ensemble de prédicats d'abstraction et pour **ELEC** avec tous les ensembles de prédicats d'abstraction.

TAUX DE COUVERTURE OBTENUS POUR RCXP AVEC DES MTS DE GRANDE TAILLE

En ce qui concerne les autres cas (ceux pour lesquels **%AT**^{rel} n'atteint pas 100%), 4 d'entre eux sont ceux avec le plus grand ensemble de prédicats d'abstraction utilisé pour le système. La taille du MTS calculé devient en effet trop grande pour que le testeur puisse comprendre à quelle situation correspond un état ou dans quelle situation une transition abstraite du MTS pourrait être déclenchée. Ainsi, il devient compliqué non seulement de proposer un ensemble de transitions cibles (que le testeur estime pouvoir être couvertes), mais également de concevoir un prédicat de pertinence adapté à la couverture de ces transitions cibles.

A titre d'exemple, la figure 8.4 montre le MTS obtenu pour **ELEV** avec le $4^{ème}$ ensemble de prédicats d'abstraction. On constate sur cette figure que le nombre d'états abstraits (119) et de transitions abstraites (755) rendent la lecture et la compréhension du MTS très difficile en pratique. La méthode de génération d'un ATS avec RCXP n'est applicable que lorsque la taille du MTS est maîtrisée par le testeur. On entend par là qu'il doit être en mesure de savoir à quelle situation correspondent les états abstraits du MTS, ou de repérer les cycles en amont des transitions ciblées. Ensuite, il doit être capable de concevoir un prédicat de pertinence permettant de guider le système dans une situation où les transitions abstraites non couvertes sont déclenchables.

Dans le cas du distributeur de café, le plus gros ensemble de prédicats d'abstraction contient 11 prédicats mais le MTS correspondant contient 14 états abstraits et 54 transitions abstraites. La figure 8.5 représente ce MTS. L'état q_{128} entouré sur la figure 8.5 constitue un état important pour couvrir de nombreuses transitions : il est la cible de 7 transitions abstraites et la source de 6 transitions abstraites à lui seul. On constate visuellement qu'il représente une situation particulièrement intéressante pour atteindre notamment les états abstraits q_{36}, q_4 par l'événement *powerDown* qui ne sont accessibles que par des transitions dont q_{128} est la source.
L'état q_{128} est atteint par CXP, mais aucun des états abstraits q_{260}, q_{288}, q_{292}, q_{36}, q_4 (donc aucun des états *may*-atteignables à partir de l'état q_{128} par une transition) n'est couvert. Tous ces états sont la cible d'une transition par l'événement *powerDown* dont q_{128} est la source. Malgré la taille du MTS relativement grande, il est possible de concevoir sans trop de difficultés un prédicat de pertinence permettant de guider l'exploration vers des états concrets de l'état q_{128}. Si ces états concrets sont suffisamment nombreux, la dernière

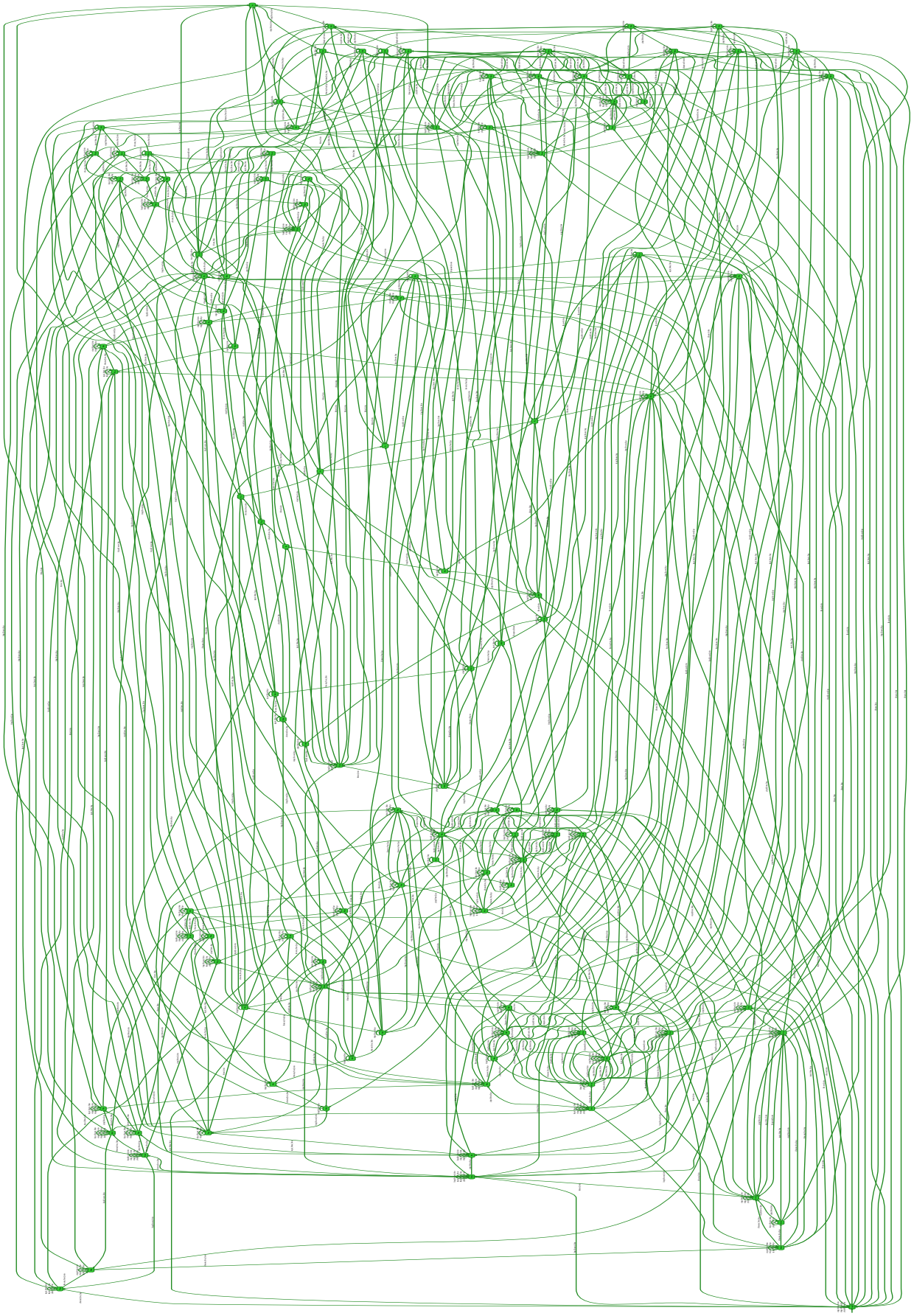


FIGURE 8.4 – MTS obtenu pour **ELEV** avec 11 prédicats d'abstraction

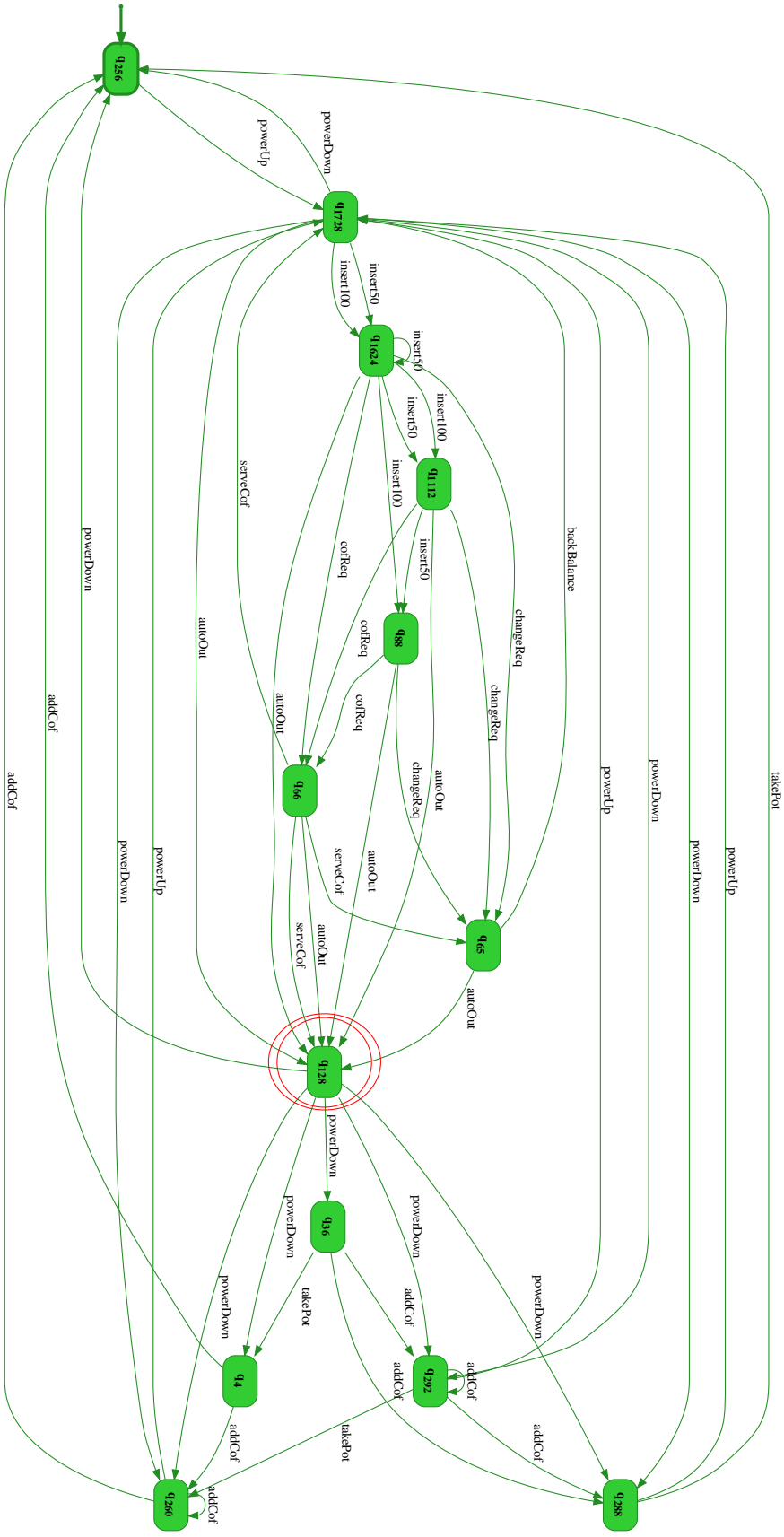


FIGURE 8.5 – MTS obtenu pour **CM** avec 11 prédicats d'abstraction

phase d'instanciation des transitions réalisée par RCXP (c'est-à-dire la concrétisation des transitions à partir des derniers états concrets pertinents vers des états concrets non pertinents) pourrait permettre la couverture de ces transitions par *powerDown*. Le prédicat de pertinence utilisé est le suivant :

$$pert_X \stackrel{def}{=} (\textit{Balance'} > \textit{Balance}) \vee (\textit{CofLeft'} < \textit{CofLeft}) \vee (\textit{AskCof} = \textit{false} \wedge \textit{AskCof'} = \textit{true}) \vee (\textit{Status} = \textit{on} \wedge \textit{Status} = \textit{off})$$

Il permet d'une part le service du dernier café (afin d'atteindre l'état q_{128} par la transition $q_{66} \xrightarrow{serveCof} q_{128}$) et le passage d'un état de fonctionnement de la machine à un état d'erreur (afin d'atteindre q_{128} par l'événement *autoOut* puisqu'il est la cible de 6 transitions dont *autoOut* est l'événement). Avec ce prédicat de pertinence, RCXP, les états abstraits q_{36}, q_{260}, q_{292} et q_{288} sont désormais couverts et 35 transitions abstraites de plus que CXP sont couvertes.
Le taux de couverture des états abstraits passe de $57,14\%$ avec CXP à $92,86\%$ avec RCXP et le taux de couverture des transitions abstraites passe de $44,44\%$ avec CXP à $79,63\%$ avec RCXP. Ces résultats révèlent qu'il est parfois possible de maîtriser relativement la lecture du MTS même lorsqu'il est de relativement grande taille. Lorsque ce n'est pas possible, des outils d'analyse de graphes permettraient peut-être de révéler les particularités qui ne seraient pas facilement identifiables par un être humain.

Le système **L14** est particulier en ce sens que chaque événement est nécessaire pour qu'une rame puisse passer d'une station à une autre : l'ouverture puis la fermeture des portes à chaque station, le départ de la station courante et l'arrivée à la station suivante, et enfin le changement de direction lorsque toutes les stations ont été parcourues. Pour couvrir les transitions non couvertes par CXP (quel que soit l'ensemble de prédicats d'abstraction utilisé) il est donc généralement nécessaire que le prédicat de pertinence autorise tous les événements à se produire. Un tel prédicat de pertinence irait à l'encontre du principe de RCXP qui est justement de restreindre l'exploration à seulement une partie des événements. RCXP a donc été appliqué à trois reprises (pour les trois trains), chacun des trois prédicats de pertinence utilisés ayant pour objectif de faire avancer un train en particulier (le premier, le deuxième, puis le troisième, dans l'ordre). La situation initiale du système (pour la première application de RCXP) doit donc être favorable à l'avancée du train numéro 1, c'est-à-dire qu'aucune autre rame ne doit se trouver sur son chemin avant qu'elle ait pu effectuer un changement de direction RCXP est ensuite de nouveau appliqué à l'ATS obtenu par la première application de RCXP, avec cette fois-ci le prédicat de pertinence faisant avancer le second train. Enfin, RCXP est appliqué une troisième fois à l'ATS obtenu par la deuxième application de RCXP, avec le prédicat de pertinence faisant avancer le troisième train.

Les expériences ont également été menées avec un prédicat de pertinence autorisant la progression de tous les trains. Cependant, le nombre de transitions pertinentes (donc instanciées) devient vite trop grand et le calcul de la valeur du variant sur chaque état cible de ces transitions rend le calcul d'un ATS avec RCXP plus lent que le calcul du système complet (FULL). En effet, dans ce cas, RCXP calcule la totalité du LTS (comme FULL) mais ajoute en plus le calcul du variant. L'utilisation de RCXP avec un prédicat de pertinence autorisant tous les événements est donc contre productif.

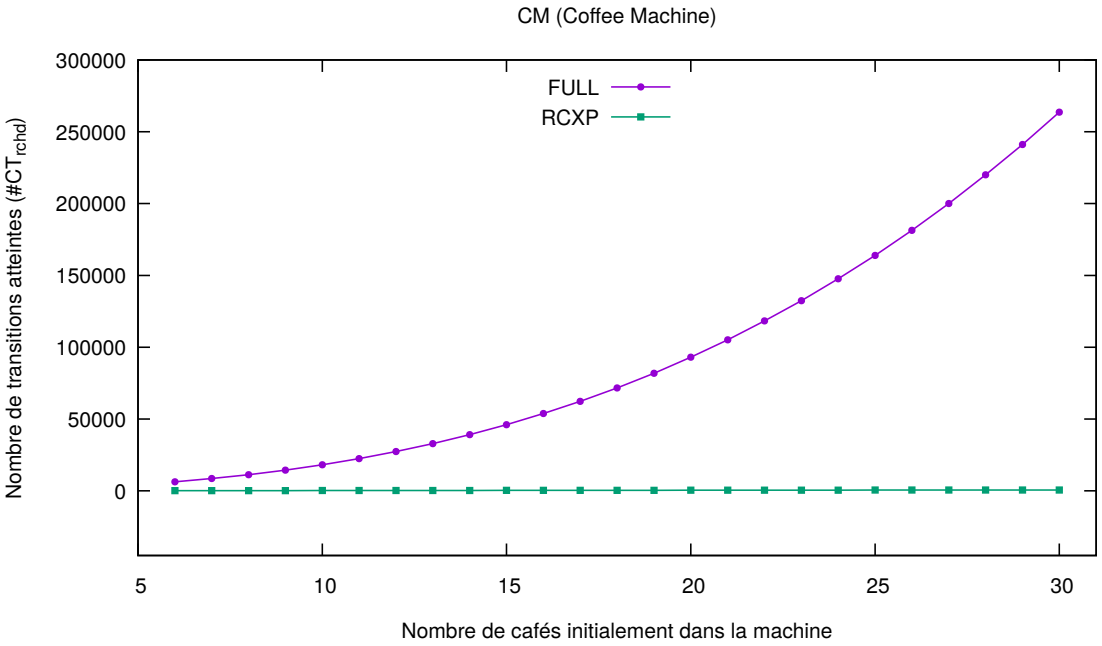


FIGURE 8.6 – Croissance du nombre de transitions concrétisées et atteintes avec RCXP et FULL en fonction du nombre de cafés initialement présents dans la machine

ANALYSE DU NOMBRE DE TRANSITIONS CONCRÉTISÉES PAR RCXP

Un autre constat intéressant pouvant être apprécié sur la table 8.3 est le fait que RCXP, malgré le fait qu'il permette généralement d'augmenter de manière significative les taux de couverture des états et des transitions, instancie peu de transitions concrètes par rapport à ce que le calcul du système complet aurait nécessité. Pour l'exemple du distributeur de café avec le 1^{er} ensemble de prédicats d'abstraction par exemple, on constate que le nombre de transitions instanciées (**#CT**) passe de 33 avec CXP à 350 avec RCXP, soit plus de 10 fois plus, pour une augmentation de 20% du taux de couverture des états abstraits et de $42,86\%$ du taux de couverture des transitions abstraites. Pour calculer le système complet, il aurait fallu instancier 10952 transitions (voir valeur de la colonne **#CT** pour la ligne FULL), soit plus de 30 fois le nombre de transitions instanciées avec RCXP.

La courbe de la figure 8.6 montre, pour l'exemple fil rouge du distributeur de café (**CM**), la croissance du nombre de transitions concrétisées et atteintes lors du calcul de RCXP et de FULL en fonction du nombre de cafés initialement dans le distributeur (qui varie ici de 6 à 30).

Le prédicat de pertinence utilisé par RCXP est celui de l'exemple fil rouge, quel que soit le nombre de cafés initialement présents dans le distributeur. Il vise donc à couvrir le service du dernier café. Bien que le nombre de cafés initialement dans la machine augmente, le MTS reste le même que pour l'exemple fil rouge (voir figure 6.1 page 93). Les taux de couverture des états abstraits (**%AS**) et des transitions abstraites (**%AT**) restent également inchangés par rapport aux résultats présentés pour le premier prédicat d'abstractions dans la table 8.3.
On constate sur la figure 8.6 que le nombre de transitions atteintes par RCXP croît linéairement (avec une pente quasiment nulle à l'échelle de la figure) lorsque le nombre

de cafés initialement présents dans la machine augmente tandis que le nombre de transitions atteintes par FULL croît exponentiellement. RCXP calcule 133 transitions atteintes avec 6 cafés initialement et augmente très exactement ce nombre de 21 transitions à chaque incrément du nombre de cafés initial. Avec 30 cafés initialement, il calcule donc 637 transitions atteintes. FULL passe quant à lui de 6306 transitions atteintes avec 6 cafés à 263634 avec 30 cafés.

La croissance linéaire observée avec RCXP est un résultat très positif de cet algorithme. Cela montre en effet qu'il est capable d'obtenir les mêmes taux de couverture des états abstraits et des transitions abstraites, quelle que soit la taille du système, tout en évitant l'exploration très coûteuse qui devrait être effectuée pour calculer le système complet.

CONCLUSION

L'algorithme RCXP permet de couvrir tout ou partie des états et transitions non couvert(e)s par CXP, notamment lorsque l'exécution répétée de transitions est nécessaire pour améliorer les taux de couverture. Pour tous les cas d'étude, après avoir identifié les transitions à cibler, il est relativement aisé de concevoir un prédicat de pertinence permettant de les couvrir avec RCXP. Pour tous les cas étudiés, RCXP se révèle largement plus efficace que FULL puisque le nombre de transitions concrètes calculées est toujours beaucoup moins grand avec RCXP qu'avec FULL (plus de 10 fois moins grand en moyenne sur les cas d'étude, sans compter le système **GSM** pour lequel FULL consomme trop de mémoire). Moyennant un effort supplémentaire de la part du testeur donc, il est possible d'obtenir la couverture des transitions ciblées en minimisant la taille de l'ATS obtenu (en termes d'états concrets et de transitions concrètes).

Les résultats moyens en termes de taux de couverture des états et des transitions de l'abstraction obtenus avec CXP et ceux obtenus avec RCXP sont très différents. Cela montre clairement qu'il sont difficilement comparables entre eux. En effet, CXP n'atteint que $51,83\%$ de couverture des états abstraits et $31,19\%$ de couverture des transitions abstraites en moyenne, alors que RCXP atteint respectivement 87% et $75,21\%$. Cela constitue une augmentation moyenne de plus de 35% des états abstraits et de près de 45% des transitions abstraites.

En moyenne, le nombre de pas de test est quant à lui presque multiplié par 100 avec RCXP, ce qui montre bien que les deux méthodes de calcul de sous-approximation ne sont pas comparables. Cependant, l'augmentation du nombre de transitions concrètes, donc de pas de test, est linéaire avec RCXP et exponentielle avec FULL pour peu que le prédicat de pertinence soit suffisamment fort, comme le montre la figure 8.6.

8.3.4/ ANALYSE COMPARATIVE DES RÉSULTATS OBTENUS AVEC RCXP ET RCXP_{ASO}

La table 8.4 présente les résultats obtenus avec RCXP et RCXP_{ASO} afin de déterminer expérimentalement quel algorithme devrait être utilisé.
N.B. Les valeurs dans la colonne **#AT**^{rel} sont parfois différentes entre RCXP et RCXP_{ASO}. En effet, certaines transitions sont couvertes par CXP_{ASO} et ne le sont pas par CXP (ou l'inverse), et ces transitions ne font donc pas partie des transitions ciblées.

TABLE 8.4 – Comparaison des résultats entre RCXP et RCXP_{ASO}

Sys.	#Ev	#AP	Alg.	#AS	%AS	#AT	%AT	$\#AT^{rel}$	$\%AT^{rel}$	#CS	$\#CS_{rchd}$	#CT	$\#CT_{rchd}$	$\mathcal{R}_{CT}$	#Steps
CM	11	3	RCXP	5	100	21	100	9	100,00	377	350	565	544	0,96	1006
			RCXP_{ASO}	5	100	21	100	8	100,00	371	350	556	544	0,98	1006
		3	RCXP	4	100	23	91,3	16	87,50	336	303	405	383	0,95	2385
			RCXP_{ASO}	4	100	23	91,3	6	66,67	326	308	404	394	0,98	2399
		11	RCXP	14	92,86	54	79,63	30	63,33	511	455	771	725	0,94	3149
			RCXP_{ASO}	14	92,86	54	79,63	30	63,33	492	456	751	727	0,97	3179
ELEC	4	2	RCXP	3	100	9	100	3	100,00	63	55	61	56	0,92	103
			RCXP_{ASO}	3	100	9	100	3	100,00	56	46	57	52	0,91	81
		2	RCXP	3	100	11	100	5	100,00	134	122	154	148	0,96	486
			RCXP_{ASO}	3	100	11	100	3	100,00	67	61	71	67	0,94	112
		4	RCXP	6	100	17	100	5	100,00	115	95	121	111	0,92	165
			RCXP_{ASO}	6	100	17	100	5	100,00	89	80	93	87	0,94	141
ELEV	11	3	RCXP	4	100	27	74,07	15	53,33	200	168	257	237	0,92	1465
			RCXP_{ASO}	4	100	27	85,19	7	42,86	200	175	258	243	0,94	1307
		2	RCXP	4	75	38	78,95	20	85,00	569	522	997	962	0,96	5048
			RCXP_{ASO}	4	75	38	78,95	17	76,47	681	631	1098	1062	0,97	4610
		3	RCXP	5	100	33	63,64	25	52,00	173	133	200	170	0,85	865
			RCXP_{ASO}	5	100	33	69,7	16	37,50	194	159	252	227	0,90	1545
		11	RCXP	119	36,13	755	18,68	635	8,04	1690	540	1526	794	0,52	5037
			RCXP_{ASO}	119	48,74	755	32,72	515	6,03	1656	672	1552	923	0,59	4870
L14	15	3	RCXP	4	100	53	90,57	45	97,78	3120	3033	4040	3993	0,99	15531
			RCXP_{ASO}	4	100	53	90,57	40	95,00	3115	3033	4364	4322	0,99	15878
		3	RCXP	4	100	50	94	42	97,62	3105	3033	4032	3993	0,99	15531
			RCXP_{ASO}	4	100	50	94	36	97,22	3098	3033	4356	4322	0,99	15878
		5	RCXP	25	100	372	80,11	338	80,77	3633	3034	4641	4322	0,93	16420
			RCXP_{ASO}	25	100	372	90,59	203	85,70	4539	4189	6527	6339	0,97	22905
GSM	16	3	RCXP	8	25	92	39,13	75	25,33	363	222	412	320	0,78	1254
			RCXP_{ASO}	8	25	92	39,13	74	24,32	308	174	338	250	0,74	922
		3	RCXP	6	100	72	58,33	55	45,45	579	467	1029	957	0,93	3563
			RCXP_{ASO}	6	100	72	58,33	54	44,44	565	458	1026	957	0,93	3334
		16	RCXP	18	50	153	39,22	129	27,91	448	225	760	607	0,80	2300
			RCXP_{ASO}	18	92,59	153	55,12	118	42,15	989	647	1971	1753	0,89	7671
Moyennes	11,40	4,81	RCXP	14,50	87,00	107,82	76,36	87,76	71,87	1090,35	928,82	1 412,41	1312,65	0,90	5 284,65
			RCXP_{ASO}	14,50	90,25	107,82	79,75	69,12	69,12	1 168,29	1 029,71	1 649,29	1564,18	0,92	5 983,29

8.3.4.1/ COMPARAISON DES TAUX DE COUVERTURE ENTRE RCXP ET RCXP_{ASO}

La couverture des états abstraits est identique pour la plupart des cas et est en moyenne très similaire entre les deux algorithmes : 87% avec RCXP et $90,25\%$ avec RCXP_{ASO}. Dans tous les cas en revanche, RCXP_{ASO} permet de couvrir autant ou plus de transitions que RCXP ($3,39\%$ de plus en moyenne). Les taux de couverture des états et des transitions identiques entre RCXP et RCXP_{ASO} s'expliquent par le fait que les ATS calculés par CXP et CXP_{ASO} sont identiques, ou couvrent en tout cas les mêmes états abstraits et les mêmes transitions abstraites, même si les instances peuvent être différentes. Lorsque les taux de couverture entre RCXP et RCXP_{ASO} sont différents en revanche, la raison est que les ATS calculés par CXP et CXP_{ASO} sont différents. Dans cette situation, deux cas sont possibles :

1. soit l'un des algorithmes (CXP ou CXP_{ASO}) couvre plus d'états abstraits ou de transitions abstraites que l'autre et cet avantage en termes de taux de couverture reste présent après avoir appliqué RCXP ou RCXP_{ASO},
2. soit les états concrets des ATS calculés par CXP et CXP_{ASO} sont différents et, certains de ces états constituant des points de départ différents, ils permettront à RCXP ou RCXP_{ASO} d'effectuer une exploration couvrant des états abstraits différents ou des transitions abstraites différentes.

8.3.4.2/ INCONVÉNIENTS POTENTIELS PROPRES À RCXP ET À RCXP_{ASO}

RCXP et RCXP_{ASO} possèdent tous les deux des inconvénients pouvant survenir lors de leur utilisation, raison pour laquelle choisir le meilleur d'entre eux dépend de l'exemple.

L'ensemble de transitions ciblées par RCXP est un sous-ensemble des transitions non

couvertes par CXP. Cet ensemble de transitions ciblées par RCXP contiendra donc généralement plus de transitions que celui des transitions ciblées par RCXP_{ASO} car l'ATS calculé avec CXP_{ASO} couvre généralement plus de transitions que celui calculé avec CXP, comme cela a été vu en section 8.3.1. Le fait de devoir identifier plus de transitions à cibler est en soit un premier inconvénient pour le testeur car cela nécessite une analyse du MTS plus approfondie de sa part. De plus, plus le nombre de transitions ciblées est grand, plus le prédicat de pertinence autorisera un grand nombre d'événements à se déclencher, et donc plus les performances de RCXP risquent de se dégrader.

A contrario, puisque RCXP_{ASO} se base sur l'ATS calculé par CXP_{ASO}, l'ensemble des transitions ciblées sera généralement moins grand que celui utilisé par RCXP. Cependant, cet ATS possédera généralement plus d'états concrets atteints que celui calculé avec CXP. Or, plus le nombre d'états concrets atteints est grand, plus le nombre d'états pertinents, et donc de points de départ des chaînes qui seront instanciées par RCXP_{ASO}, est grand. Les performances de RCXP_{ASO} peuvent donc être fortement dégradées si l'ATS calculé par CXP_{ASO} crée trop de points de départ pour ces chaînes.

8.3.4.3/ COMPARAISON DES ÉTATS CONCRETS ET DES TRANSITIONS CONCRÈTES CALCULÉES PAR RCXP ET RCXP_{ASO}

Dans 4 cas sur 16, RCXP_{ASO} calcule plus d'états concrets (voir la colonne **#CS**) que RCXP et, dans 8 cas sur 16, il calcule plus de transitions concrètes (voir la colonne **#CT**). Dans la plupart des cas, la différence entre le nombre d'états concrets calculés par RCXP et RCXP_{ASO} est relativement faible. Il en va de même pour la différence entre le nombre de transitions concrètes calculées par RCXP et RCXP_{ASO}, sauf pour quelques cas particuliers.
Par exemple, dans le cas du système **GSM** avec le $3^{ème}$ ensemble de prédicats d'abstraction, le nombre de transitions calculées par RCXP_{ASO} (1971) est plus de deux fois plus élevé que le nombre de transitions calculées par RCXP.
Les cas particuliers pour lesquels RCXP_{ASO} calcule beaucoup plus de transitions concrètes sont liés au fait que l'ATS calculé par CXP_{ASO} calcule soit plus d'états concrets, soit des états concrets qui serviront de points de départ pour des chaînes de transitions plus longues que celles calculées par CXP. On retrouve donc dans les résultats les inconvénients potentiels de RCXP_{ASO} évoqués en section 8.3.4.2.

CONCLUSION

Dans certains cas particuliers, RCXP_{ASO} pourrait explorer beaucoup plus de chaînes de transitions que RCXP, dans le cas notamment où CXP_{ASO} calculerait un ATS avec beaucoup plus d'états concrets que celui calculé avec CXP. Le nombre d'instanciations réalisées durant l'exploration effectuée par RCXP_{ASO} pourrait donc théoriquement être beaucoup plus grand que pour RCXP. Les cas où cette situation se produit sont cependant rares, en tout cas sur l'ensemble des cas étudiés ici, et le nombre d'états concrets et de transitions instancié(e)s reste en pratique tout à fait acceptable.

La différence dans les moyennes des taux de couverture obtenus avec RCXP_{ASO} et RCXP_{ASO} sont faibles (87% pour RCXP contre 90,25% pour RCXP_{ASO} pour les états abstraits et 76,36% pour RCXP contre 79,75% pour RCXP_{ASO} pour les transitions abstraites). Cela indique que le choix de l'utilisation de CXP_{ASO} plutôt que CXP dans le cas où la

sous-approximation sera complétée par RCXP ou RCXP_{ASO} n'est pas crucial. Il est dans ce cas plus important de concevoir un prédicat de pertinence adapté aux objectifs de couverture du testeur.

Par ailleurs, RCXP_{ASO} requiert généralement de cibler moins de transitions que RCXP et tend donc à alléger le travail du testeur, non seulement lorsqu'il s'agit d'identifier les transitions à cibler, mais également lorsqu'il s'agit d'exprimer le prédicat de pertinence. Moins le nombre de transitions à cibler est grand, plus le nombre d'événements à autoriser avec le prédicat de pertinence aura tendance à être petit, ce qui permet de meilleures performances lors de la phase d'exploration.

Ces observations semblent indiquer que RCXP_{ASO} est donc à préférer à RCXP dans le cas général. Si en revanche RCXP_{ASO} se révèle vraiment trop long à l'exécution, ou instancie trop de transitions, RCXP peut se révéler tout aussi efficace en termes de couverture des transitions ciblées tout en calculant potentiellement moins d'instances.

8.4/ BILAN SUR LES RÉSULTATS ET CONCLUSIONS

Les résultats présentés précédemment ont permis d'évaluer expérimentalement la capacité des algorithmes CXP, CXP_{ASO}, BCI et RCXP à couvrir les états abstraits et les transitions abstraites d'abstraction de modèles (MTS ou 3MTS).

CONCLUSIONS SUR CXP ET CXP_{ASO}

Les avantages de CXP_{ASO} par rapport à CXP sont facilement identifiables, notamment grâce aux résultats. D'abord, CXP_{ASO}, contrairement à CXP, ne requiert pas de la part du testeur d'exprimer une fonction d'ordonnancement des états abstraits, ce qui allège la tâche du testeur. Ensuite, les résultats montrent clairement l'avantage que représente la fonction d'ordonnancement dynamique utilisée par CXP_{ASO} : les taux de couverture des états abstraits et des transitions abstraites ne sont jamais moins bons avec CXP_{ASO} qu'avec CXP, et sont même souvent meilleurs. Enfin, CXP_{ASO} calcule très souvent moins de transitions qui ne seront pas atteintes dans les tests que CXP, tout en améliorant les taux de couverture. Le CTS qu'il génère constitue donc généralement une sous-approximation plus complète du système que celui généré par CXP.

CONCLUSIONS SUR CXP ET BCI

Les taux de couverture des états abstraits et des transitions abstraites obtenus avec l'algorithme BCI ne sont que très rarement meilleurs que ceux obtenus avec CXP. Ils ne sont d'ailleurs jamais meilleurs que ceux obtenus avec CXP_{ASO}. Le problème de BCI est qu'il dépend fortement du nombre de *must*-transitions dans le MTS, le calcul des modalités étant d'ailleurs déjà coûteux en lui-même. Si ce nombre est faible, et encore plus si les *must*-transitions du 3MTS sont des transitions déjà couvertes par CXP, BCI ne peut pas permettre d'obtenir de meilleurs taux de couverture. En l'occurrence, ce problème survient sur tous les exemples utilisés pour mener les expérimentations. Par conséquent, pour que BCI soit éventuellement viable pour améliorer les taux de couverture, il faudrait modifier les prédicats d'abstraction pour que les transitions non couvertes par CXP aient

la modalité *must*, et espérer que les instances de ces transitions soient atteintes. Il n'est donc pas recommandé d'utiliser BCI car cet algorithme, en plus d'être coûteux, n'apporte que très peu d'amélioration des taux de couverture.

CONCLUSIONS SUR CXP ET RCXP

Contrairement à BCI, le recours au prédicat de pertinence et à l'algorithme RCXP s'est montré très efficace dans nos expériences. En effet, même si la conception du prédicat de pertinence requiert l'intervention du testeur et repose sur sa connaissance du système, les résultats obtenus avec RCXP montrent une nette amélioration des taux de couverture. Sur tous les exemples traités, il est assez facile, pour une personne initiée aux méthodes formelles en tout cas, d'énoncer ce prédicat de pertinence pour couvrir les transitions ciblées. L'utilisation de RCXP résout le problème majeur de CXP et de CXP_{ASO} qui est qu'ils ne concrétisent qu'une à trois fois chaque transition abstraite. RCXP permet, comme cela a été mis en évidence par les résultats expérimentaux, d'appliquer les cycles nécessaires à la couverture de certaines transitions abstraites nécessitant l'exécution de ces cycles.

De plus, l'exploration effectuée par RCXP présente l'avantage de toujours terminer à l'aide d'une borne, le variant, dépendante du modèle lui-même, puisque le variant est calculé à partir de la taille des ensembles des variables. RCXP est ainsi utilisable en pratique lorsque l'explosion combinatoire rend l'exécution de FULL extrêmement longue.

RCXP n'est en revanche viable que lorsque le prédicat de pertinence ne vise que quelques transitions cibles pour éviter l'explosion. Il est évident que si le prédicat de pertinence autorise presque tous les événements, RCXP rencontre les mêmes difficultés que FULL. Il est possible de l'appliquer plusieurs fois sur des sous-ensembles de transitions cibles disjoints. Cette expérience a été réalisée sur le système **L14** et a permis de montrer que l'utilisation successive de RCXP pour plusieurs sous-ensembles de transitions cibles permet d'explorer une partie plus restreinte que celle explorée avec un prédicat de pertinence autorisant tous les trains à se déplacer en même temps.

CONCLUSIONS SUR RCXP VS RCXP_{ASO}

RCXP_{ASO} est à privilégier par rapport à RCXP en raison du fait que l'ATS calculé avec CXP_{ASO} couvre généralement autant ou plus d'états et de transitions que celui calculé avec CXP. Cet avantage de couverture obtenu avec CXP_{ASO} implique que le testeur aura moins de transitions cibles à identifier que s'il utilisait CXP, et que le prédicat de pertinence autorisera également l'application de moins d'événements. Cela se traduit non seulement par des performances généralement accrues mais aussi par un effort moins important de la part du testeur. RCXP se montre malgré tout viable et même utile dans le cas où RCXP_{ASO} explore trop de chaînes de transitions à cause de l'ATS calculé par CXP_{ASO}.

CONCLUSION GÉNÉRALE SUR LES ALGORITHMES

Les résultats montrent que CXP_{ASO} est généralement plus intéressant à utiliser que CXP car il produit des taux de couverture au moins aussi bons que CXP. Il est possible, et

même probable, que $\mathrm{CXP}_{\mathrm{ASO}}$ ne couvre malgré tout pas tous les états abstraits ou toutes les transitions abstraites. Pour couvrir ces états et ces transitions, BCI ne répond pas au problème, bien qu'il présente l'avantage de ne pas nécessiter d'intervention de la part du testeur. Généralement, les transitions abstraites non couvertes par CXP ou $\mathrm{CXP}_{\mathrm{ASO}}$ sont celles qui ne peuvent être atteintes qu'après l'exécution répétée d'un ou plusieurs cycle(s) de transitions. RCXP et $\mathrm{RCXP}_{\mathrm{ASO}}$ permettent tous les deux cette exécution répétée si la fonction de pertinence autorise l'exécution des événements des transitions de ce(s) cycle(s). Par conséquent, ils permettent de couvrir les transitions non couvertes par CXP ou $\mathrm{CXP}_{\mathrm{ASO}}$ et sont utilisables en pratique sur des systèmes de grande taille, à condition de limiter l'ensemble de transitions ciblées à chaque application. $\mathrm{RCXP}_{\mathrm{ASO}}$ est à préférer à RCXP en raison du fait que l'ATS calculé par $\mathrm{CXP}_{\mathrm{ASO}}$ et utilisé par $\mathrm{RCXP}_{\mathrm{ASO}}$ produit de meilleurs taux de couverture que CXP. Ainsi, le prédicat de pertinence a tendance à nécessiter l'application de moins d'événements puisque certaines transitions sont déjà couvertes.

III

CONCLUSION ET TRAVAUX FUTURS

9

CONCLUSION

Nos travaux proposent plusieurs méthodes de génération automatique de tests à partir d'abstraction de modèles de systèmes événementiels. L'objectif principal de nos méthodes est d'obtenir des tests couvrant suffisamment (évaluation laissée à l'appréciation du testeur) les états et les transitions de cette abstraction. Les tests prennent la forme de séquences d'événements effectivement réalisables sur le système et constituent donc une sous-approximation de celui-ci. Afin de maîtriser l'explosion combinatoire qui émergerait du calcul de l'ensemble des exécutions possibles du système, les méthodes proposées procèdent au calcul de son abstraction par prédicats. Cette abstraction peut être représentée sous la forme d'un graphe d'états et de transitions. Ce graphe est garanti d'être fini et est généralement plus petit que celui représentant les comportements du modèle complet. Manipuler cette abstraction pour générer des tests est ainsi nettement plus efficace qu'une méthode parcourant une structure modélisant le système de manière exhaustive.

Cependant, la concrétisation immédiate des transitions d'un système abstrait donne généralement lieu à une approximation dont les transitions ne sont pas connectées entre elles ou ne sont pas atteignables sur le système. L'absence de flot de contrôle et du non-déterminisme inhérents aux systèmes événementiels contribuent fortement à l'apparition de ce problème compliquant le calcul d'une sous-approximation. Les méthodes proposées contribuent donc à résoudre ce problème en favorisant la connectivité et l'atteignabilité des transitions instanciées et en guidant le calcul de ces instances pour couvrir des séquences pertinentes vis-à-vis d'un objectif de test.

9.1/ GÉNÉRATION AUTOMATIQUE DE TESTS À PARTIR D'UNE ABSTRACTION DE MODÈLES

Nous avons proposé un ensemble de méthodes pouvant être combinées entre elles et permettant le calcul automatique d'une sous-approximation de modèles à partir de leur abstraction par prédicats. La sous-approximation calculée prend la forme d'un système d'états concrets et de transitions concrètes constituant des instances des états et des transitions de l'abstraction.

Les prédicats d'abstraction utilisés pour réaliser l'abstraction du modèle sont sélectionnés automatiquement à partir d'un objectif de test exprimant une propriété dynamique exprimée sur les événements du système. La garde de chacun des événements apparaissant dans cet objectif de test est ainsi utilisée comme prédicat d'abstraction. Les

propriétés dynamiques auraient tout à fait pu être exprimées sur les états du système auquel cas chaque contrainte d'état aurait constitué un prédicat d'abstraction. Le principal dans le cadre de notre démarche est de pouvoir extraire de manière automatique ces prédicats afin que l'abstraction calculée permette d'observer les comportements du système vis-à-vis de cette propriété. A partir de ces prédicats d'abstraction, l'ensemble des états abstraits non vides, c'est-à-dire ceux qui ne contredisent pas l'invariant et pour lesquels les prédicats d'abstraction n'entrent pas en contradiction, peut être calculé de manière automatisée à l'aide d'un solveur de contraintes. Cet ensemble est indispensable à l'utilisation des méthodes que nous proposons.

Lorsque l'ensemble des états abstraits est déterminé, les deux algorithmes CXP et $\mathrm{CXP}_{\mathrm{ASO}}$ peuvent être appliqués au système. Ces deux méthodes permettent de calculer l'ensemble des états abstraits et de transitions abstraites pouvant être atteints depuis l'état abstrait initial (il s'agit des états et des transitions *may*-atteignables). En même temps qu'elles calculent ce système abstrait, ces deux méthodes calculent des instances concrètes des états et des transitions qui le composent. Ce faisant, les deux méthodes conservent une information précieuse sur l'atteinte (ou non) des instances depuis un état concret initial du système.
Dans le cas de CXP, l'idée est d'essayer de connecter les nouvelles instances à d'autres instances atteintes depuis un état concret initial. L'objectif de cet algorithme est d'une part d'essayer de maximiser la couverture, par la sous-approximation, des états abstraits et transitions abstraites instancié(e)s, et d'autre part de garantir qu'une instance de chaque état abstrait et de chaque transition abstraite soit calculée.
Les objectifs de l'algorithme $\mathrm{CXP}_{\mathrm{ASO}}$ sont identiques à ceux de CXP : seule la démarche employée diffère. En effet, CXP calcule une instance d'une transition abstraite pour un état abstrait source et un état abstrait cible donnés garantis d'être *may*-atteignables. Cependant, rien ne garantit qu'un état concret atteint déjà instancié au préalable puisse être la source d'une instance de cette transition en particulier. L'idée derrière $\mathrm{CXP}_{\mathrm{ASO}}$ est de ne pas contraindre l'état abstrait cible de la transition à instancier. S'il existe un état concret connu et atteint appartenant à l'état abstrait source de la transition à instancier et que l'événement peut être appliqué à cet état, l'état concret ainsi atteint instancie un état abstrait *may*-atteignable. L'algorithme ne contraint donc pas l'état abstrait cible à atteindre mais laisse plutôt le solveur en trouver un, s'il existe.
CXP et $\mathrm{CXP}_{\mathrm{ASO}}$ calculent donc tous les deux des instances de transitions abstraites dont l'atteinte depuis un état initial est connu. Si une transition instanciée n'est pas atteinte depuis un état initial, on ne peut pas conclure sur son atteignabilité sur le système. Afin de garantir que les tests soient instanciables sur le système, ces transitions ne seront pas conservées. Si en revanche une transition est atteinte par une séquence de transitions depuis un état initial, la séquence complétée par cette transition constitue bien une sous-approximation atteignable du système. Les tests générés à partir de l'approximation calculée par CXP et $\mathrm{CXP}_{\mathrm{ASO}}$ seront donc exclusivement composés de l'ensemble des transitions garanties d'être atteintes.

L'algorithme BCI requiert le calcul au préalable d'une sous-approximation du système et des modalités des transitions du système abstrait, ce qui peut-être réalisé par CXP ou $\mathrm{CXP}_{\mathrm{ASO}}$. A partir de ces informations, il instancie des arbres de $must^{-}$-transitions et de $must^{+}$-transitions tout en essayant de maximiser leur taille. Chaque $must^{-}$-transition et chaque $must^{+}$-transition n'est instanciée qu'une seule fois afin de garantir une complexité linéaire de l'algorithme. Lorsque c'est possible, la première instance de ces séquences de transitions abstraites est calculée à partir d'un état concret atteint.

Puisque les chaînes de $must^-$ et les chaînes de $must^+$ sont garanties d'être instanciables si l'état source est atteint, l'algorithme BCI complète donc l'approximation obtenue avec CXP et CXP_{ASO}. Puisqu'il maintient également l'information d'atteinte des séquences instanciées, les transitions constituant une sous-approximation du système, et donc appartenant aux tests, restent distinguables de celles constituant potentiellement une sur-approximation.

L'algorithme RCXP à également pour objectif de compléter la sous-approximation obtenue à partir de CXP (ou CXP_{ASO} pour RCXP_{ASO}). En effet, ni l'algorithme CXP, ni l'algorithme CXP_{ASO} ne garantissent que les tests couvriront bien l'ensemble des états abstraits et des transitions abstraites pouvant être couvertes par des tests instanciables. Le testeur peut donc souhaiter compléter la sous-approximation obtenue dans le but de remplir ce critère minimal de couverture.
Dans ce but, RCXP donne au testeur la possibilité de guider une exploration concrète du système à partir des états atteints dans la sous-approximation obtenue par CXP. Cette exploration est automatiquement effectuée mais exige tout de même de la part du testeur de concevoir un prédicat de pertinence qui guidera cette exploration concrète. Si ce prédicat de pertinence est conçu pour viser un objectif de test particulier, son utilisation avec RCXP peut permettre d'obtenir une sous-approximation couvrant effectivement plus d'états et de transitions abstraites que celle obtenue avec CXP et CXP_{ASO}.
Rappelons toutefois que l'atteignabilité d'au moins une instance concrète des *may*-transitions étant un problème indécidable, aucune méthode ne saura garantir au testeur la couverture de toutes les transitions abstraites de son modèle.

Toutes ces méthodes permettent donc d'obtenir ou de compléter une approximation d'un système à partir d'une abstraction par prédicats du système modélisé. Les états concrets et les transitions concrètes appartenant à cette approximation et pouvant être atteints depuis au moins l'un des états concrets initiaux du système constituent une sous-approximation de ce système. Les états et transitions appartenant à cette approximation mais n'étant pas atteint(e)s ne doivent pas apparaître dans les tests générés car pourraient constituer une sur-approximation du système. Même si certains états ou transitions sont effectivement atteignables sur l'implémentation du système, la sous-approximation ne permet pas de déterminer quelle séquence de transitions déclencher pour les atteindre. On ne peut donc pas générer de tests composés de ces états et de ces transitions.
Un simple parcours en avant des transitions constituant la sous-approximation à partir des états concrets initiaux du système peut ensuite faire office de génération de tests : chaque séquence de transitions parcourue constitue un cas de test. Cependant, on préfère généralement limiter le nombre de cas de test. En effet, pour une même couverture, moins les cas de test sont nombreux, plus leur instanciation sur l'implémentation du système aura de chance d'être efficace. Par conséquent, on préfère l'utilisation de l'algorithme du postier chinois qui effectue un parcours en avant des transitions tout en maximisant la taille des cas de test et donc en limitant leur nombre.

Les méthodes proposées permettent donc de générer de manière entièrement automatisée des tests issus d'une abstraction de modèles.

9.2/ GÉNÉRATION DE TESTS GUIDÉE ET LIMITÉE SELON LEUR PERTINENCE VIS-À-VIS D'UN OBJECTIF DE TEST

Une autre contribution apportée par nos travaux concerne la génération efficace de tests pertinents vis-à-vis d'un objectif de test.

Les algorithmes CXP et CXP_{ASO} guident le calcul de la sous-approximation pour favoriser la couverture des états et des transitions de l'abstraction d'un système. L'abstraction du système est réalisée à partir de prédicats issus d'un objectif de test particulier. Elle permet donc d'observer, dans une certaine mesure, les différents comportements possibles du système vis-à-vis de cet objectif de test. Globalement, on peut ainsi qualifier de pertinents tous les tests générés à partir de la sous-approximation obtenue avec CXP et CXP_{ASO}.
Il n'est cependant pas garantit que CXP ou CXP_{ASO} couvrent l'ensemble des états et l'ensemble des transitions de l'abstraction. Pour résoudre ce problème, nos recherches se sont ainsi basées sur l'idée de compléter la sous-approximation obtenue avec ces algorithmes par de nouvelles séquences de transitions pertinentes d'une part selon le testeur et d'autre part par leur apport en termes de couverture des états abstraits et des transitions abstraites.

Au même titre que CXP et CXP_{ASO}, on peut considérer que l'algorithme BCI guide la génération des tests vis-à-vis de l'objectif de test. En effet, les modalités des transitions d'un système abstrait dépendent des prédicats d'abstraction utilisés qui, comme cela à déjà été évoqué, sont directement extraits de l'objectif de test. Comme expliqué dans [Pasareanu et al., 2007, Ball, 2005, Bride et al., 2016a], l'utilisation de la garde des événements apparaissant dans l'objectif de test tend à favoriser l'apparition de modalités *must* sur les transitions appliquant ces événements. Puisque les *must*-transitions sont toutes de nouveau instanciées par BCI, toutes les séquences de $must^-$-transitions et $must^+$-transitions atteintes amélioreront potentiellement la couverture de l'abstraction.

L'algorithme RCXP est quant à lui spécifiquement conçu pour permettre au testeur de déterminer les états et les transitions qu'il souhaite impérativement couvrir. Pour cela, nous proposons que le testeur étudie l'abstraction du système calculée par CXP ou CXP_{ASO}, plutôt que le système concret qui, rappelons-le, peut être très grand voir infini.
Ainsi, si le testeur estime que tous les états et toutes les transitions du système abstrait doivent être couverts, alors celles et ceux qui ne sont pas couverts par CXP ou CXP_{ASO} seront considérées comme pertinents. S'il souhaite plutôt faire en sorte que les tests placent le système dans une situation particulière, les transitions abstraites constituant les séquences permettant de mettre le système dans cette situation seront considérées comme pertinentes. Les transitions pertinentes seront celles explorées et instanciées par RCXP, les autres ne le seront pas.
Afin de caractériser ces transitions pertinentes de manière exploitable par RCXP, nous proposons de faire appel à l'expertise du testeur et à sa connaissance du système. À partir du système abstrait (le MTS) calculé par CXP et CXP_{ASO}, le testeur détermine les transitions abstraites devant être instanciées lors de l'exploration pour mener le système dans l'état souhaité.
Une fois ces séquences de transitions déterminées, le testeur conçoit un prédicat de pertinence qui permet à RCXP d'évaluer la pertinence des transitions qu'il instancie. Si une transition est pertinente, elle est instanciée et l'exploration se poursuit depuis celle-

ci. L'exploration s'arrête alors lorsque plus aucune transition n'est évaluée comme pertinente, raison pour laquelle le prédicat de pertinence doit restreindre le nombre de transitions concernées.
Afin de garantir la terminaison du processus d'exploration, nous proposons également d'estimer une valeur de variant qui diminue lorsqu'une transition pertinente est instanciée. Lorsque la valeur de ce variant atteint zéro, l'exploration s'arrête même si la dernière transition instanciée est pertinente. Le calcul de la valeur du variant est automatisée et dépend d'une part de la taille des domaines des variables présentes dans le prédicat de pertinence et d'autre part du nombre d'événements susceptibles de générer des transitions pertinentes.

9.3/ IMPLÉMENTATION ET EXPÉRIMENTATIONS

Nous avons également développé un outil appelé STRATEST permettant d'appliquer tous les algorithmes proposées dans cette thèse à un modèle de système événementiel, un ensemble de prédicats d'abstraction et éventuellement un prédicat de pertinence. Cet outil automatise complètement le calcul des états abstraits non vides, l'application des algorithmes CXP, CXP_{ASO}, BCI, RCXP et RCXP_{ASO}, la génération de tests par l'algorithme du postier chinois ainsi que la production de statistiques diverses concernant ces tests. Il peut également être utilisé sous la forme d'une bibliothèque logicielle permettant de manipuler des systèmes événementiels pour une utilisation différente et l'application d'autres algorithmes à ceux-ci.
Les expérimentations réalisées sur l'ensemble d'exemples de systèmes événementiels ont été menées à l'aide de STRATEST. Les résultats obtenus sont donc en pratique dépendants de l'implémentation de cet outil et du solveur SMT qu'il utilise c'est-à-dire Z3 [de Moura et al., 2008]. Ces résultats montrent les avantages et les inconvénients des méthodes que nous avons proposées, notamment en termes de qualité de couverture des états et des transitions d'une abstraction de systèmes.
Nous avons montré la capacité des algorithmes à produire une sous-approximation et des tests à partir de cette abstraction de manière efficace tout en améliorant la connectivité des transitions qui le composent par rapport à un algorithme de concrétisation *naïf*. Nous avons également montré la capacité de ces algorithmes à générer une sous-approximation composée de transitions pertinentes vis-à-vis d'un objectif de test défini par le testeur. Les méthodes ainsi que leur implémentation possèdent cependant certaines limites mises en évidence par les expérimentations. En effet, la couverture de tous les états et de toutes les transitions n'est pas toujours obtenue en raison des déconnexions, entre les transitions instanciées, qui sont liés à l'abstraction réalisée du système. L'outil STRATEST ne prend en charge qu'une partie du pouvoir d'expression du langage B. Nous présentons donc dans le chapitre 10 les perspectives futures qui pourraient permettre de dépasser ces limitations.

10

TRAVAUX FUTURS

Ce chapitre présente les perspectives liées aux travaux présentés dans cette thèse : différentes pistes de recherche futures et améliorations possibles à envisager sont proposées.

10.1/ EXTENSION DE LA PRISE EN CHARGE DU LANGAGE DE MODÉLISATION

Le langage de modélisation utilisé par l'outil STRATEST est le langage B événementiel. Ce langage permet l'expression de contraintes complexes et de réaliser des opérations aussi bien sur des fonctions partielles que sur des listes ou des ensembles. STRATEST s'appuyant sur l'utilisation d'un solveur SMT, il est nécessaire de traduire les contraintes exprimées dans le modèle par des contraintes exprimées dans le langage du solveur. Cette étape de traduction est délicate (parfois même impossible selon le solveur utilisé) et la preuve de correction de cette traduction n'est pas toujours évidente à produire.
Ainsi, STRATEST permet à ce jour l'utilisation des objets mathématiques suivants :

- les ensembles finis d'entiers et les ensembles énumérés,
- les constantes et variables de type entier ou prenant leur valeur dans un ensemble énuméré,
- les fonctions renvoyant un entier et acceptant un unique paramètre de type entier dont le domaine de définition est fini (les règles de traduction utilisées l'imposent),
- les formules booléennes et arithmétiques usuelles incluant les quantificateurs,
- toutes les substitutions primitives ainsi que la substitution conditionnelle **if** ... **then** ... **else** ....

Une extension de la part du langage prise en charge pourrait permettre l'application et l'évaluation des différents algorithmes sur de nouveaux modèles d'échelle industrielle. Un analyseur syntaxique et sémantique plus complet ainsi que les règles de traduction associées constitueraient donc une amélioration importante de l'outil et permettrait une évaluation plus précise du passage à l'échelle des contributions présentées.

10.2/ ÉTUDE DE L'IMPACT DES MÉTHODES D'EXTRACTION DES PRÉDICATS D'ABSTRACTION

Les résultats obtenus expérimentalement dépendent des prédicats d'abstraction utilisés. Ceux-ci ont été choisis comme étant la garde des événements apparaissant dans la formule temporelle constituant l'objectif de test.

D'autres méthodes possibles, proposées par exemple dans [Bride et al., 2016a], consisteraient à utiliser la post-condition des événements (c'est-à-dire la contrainte la plus forte possible sur les variables d'états après application d'un événement). Il serait également envisageable d'utiliser à la fois la garde et la post-condition des événements apparaissant dans l'objectif de test comme prédicats d'abstraction. Enfin, des prédicats d'abstraction n'étant constitués que de parties de la garde ou de la post-condition pourraient également être étudiés.

Il conviendrait alors de mener de nouvelles expériences et d'observer l'impact de ces changements de couverture structurelle de l'abstraction obtenus avec les tests.

10.3/ ÉTUDES DE NOUVEAUX CRITÈRES DE COUVERTURE

Les critères de couverture analysés dans le cadre des expérimentations sont la couverture des états abstraits et des transitions abstraites. L'analyse des taux de couverture des chemins des modèles est rendue difficile par la taille de ces derniers et n'a donc pas été effectuée.

Un critère ayant été envisagé durant cette thèse est la couverture des chemins du MTS. Cependant, lorsque des cycles sont présents dans un MTS, le nombre de chemins à considérer peut être infini et il sera nécessaire de se limiter aux k-chemins pour des valeurs de k choisies judicieusement. Une autre possibilité serait d'étudier la couverture par les tests des chemins du système de transitions issu de l'objectif de test. En effet, à toute formule de la logique temporelle linéaire peut être associé un système fini d'états et de transitions appelé automate de Büchi [Büchi, 1990, Somenzi et al., 2000].

10.4/ AMÉLIORATION DES ALGORITHMES DE CALCUL D'UNE SOUS-APPROXIMATION

Les expérimentations ont montré que la sous-approximation calculée par $\mathrm{CXP}_{\mathrm{ASO}}$ couvre généralement plus d'états et de transitions abstraites que CXP. Cela provient du fait que l'état abstrait cible rejoint par la transition à instancier n'est pas déterminé à l'avance. Le solveur SMT se charge en effet de trouver un état abstrait pouvant être rejoint par une instance atteinte dans la sous-approximation. Cette amélioration par rapport à CXP ne porte dans nos travaux que sur les états abstraits cible.
Il pourrait être intéressant d'observer comment se comporte $\mathrm{CXP}_{\mathrm{ASO}}$ lorsque l'événement à déclencher n'est pas non plus imposé. L'idée est de débuter l'instanciation à partir des états abstraits may-atteignables (initialement les états abstraits initiaux) et de laisser le solveur déterminer un événement, parmi l'ensemble des événements du système, pouvant être appliqué aux états concrets atteints connus. L'état concret cible obtenu par appli-

cation de cet événement serait ainsi nécessairement atteint dans la sous-approximation et la transition abstraite correspondante serait couverte par les tests. La différence avec l'implémentation actuelle de $\mathrm{CXP}_{\mathrm{ASO}}$ est que l'événement à appliquer à un instant donné lors de l'exploration n'est pas contraint, de même que l'état abstrait cible à rejoindre n'est pas contraint contrairement à ce qui est fait avec CXP.
Au vu des résultats prometteurs de $\mathrm{CXP}_{\mathrm{ASO}}$ par rapport à CXP, il est probable que cette amélioration supplémentaire produise des résultats encore plus intéressants en terme de couverture de l'abstraction. Une limite de cette amélioration est la complexité des formules à résoudre pour le solveur. En effet la méthode correspondrait davantage à une méthode d'exploration symbolique car elle manipulerait les événements et les états abstraits cible de manière symbolique.

L'algorithme BCI n'a pas donné lieu à des résultats suffisamment prometteurs pour poursuivre les recherches concernant les modalités des transitions abstraites et l'instanciation des chaînes de Ball. Une amélioration souhaitable de l'algorithme dans son état actuel pourrait consister à augmenter le nombre de $must^{-}$-transitions instanciées et effectivement atteintes. Pour cela, il faut augmenter le nombre d'instances de séquences de $must^{-}$-transitions dont l'état source est atteint ($vert$). Pour optimiser ce principe, l'algorithme de concrétisation des $must^{-}$-transitions essaie déjà de mémoriser au cours de l'exploration en arrière les états verts permettant de concrétiser la séquence de transitions abstraites en avant par des transitions concrètes atteintes. Lorsque c'est possible, toute la séquence est instanciée à partir d'un état concret atteint, mais il est possible qu'une instance calculée ne puisse pas avoir pour source un état atteint (par exemple si aucun état concret atteint n'est connu pour l'état abstrait source).
En l'état, l'algorithme de concrétisation des $must$-transitions n'instancie qu'une fois les $must$-transitions. Si une instance de $must$-transition calculée n'est pas atteinte, BCI ne cherchera plus d'autres instances de cette transition. L'idée que nous suggérons est alors de mémoriser les $must$-transitions dont les instances ne peuvent pas être atteintes au moment où elles sont calculées. Lorsqu'une autre chaîne appartenant à la même $must$-structure est instanciée, les états concrets instanciés et atteints pourraient servir de point de départ pour instancier de nouveau les $must$-transitions ainsi mémorisées. Cela pourrait théoriquement augmenter le nombre de $must$-transitions couvertes par les tests, au prix d'une complexité un peu plus grande car certaines $must$-transitions pourraient être instanciées plusieurs fois.

L'algorithme RCXP peut faire l'objet de plusieurs pistes de recherche. D'une part, le calcul du variant, qui permet de limiter l'exploration et garantir sa finitude, est effectué à partir de la taille du domaine de définition des variables composant le prédicat de pertinence. Le variant peut donc initialement être très grand (et même non calculable dans le cas d'un domaine de définition infini), et ainsi ne limiter que très peu le phénomène d'explosion combinatoire due à l'exploration réalisée. En effet, si le variant permet l'exploration de $n \in \mathbb{N}$ transitions et que toutes les transitions pertinentes peuvent être couvertes par des séquences de transitions de taille inférieure à n, l'exploration n'est pas limitée par le variant mais par le prédicat de pertinence. Le prédicat de pertinence peut cependant sur-approximer les transitions pertinentes et le problème d'explosion du nombre de transitions instanciées peut de nouveau surgir. De même, le variant décroît toujours de 1 après la concrétisation d'une transition indépendamment de l'effet de cette transition. Or si cette transition fait progresser le système de manière significative vers les transitions pertinentes à couvrir, une décroissance plus forte du variant pourrait être utilisée. Ainsi, une amélioration souhaitable consisterait à mieux approximer la valeur initiale du variant

et à corréler sa décroissance à la *pertinence* des transitions instanciées.
D'autre part, la conception du prédicat de pertinence est une tâche manuelle dont il est difficile d'évaluer la faisabilité sur des abstractions de systèmes de taille industrielle. Un outil d'analyse de graphes pourrait identifier automatiquement les chemins abstraits susceptibles de mener aux transitions ciblées par RCXP et laisser au testeur la tâche de déterminer lequel (ou lesquels) pourrai(en)t convenir à son objectif. Pour se rapprocher d'une méthode complètement automatisée, l'outil pourrait également déduire des prédicats de pertinence à partir des événements apparaissant dans les chemins identifiés par le testeur.
Enfin, l'évaluation de la pertinence est pour l'instant réalisée par le biais d'un prédicat de pertinence, c'est-à-dire qu'une transition peut être considérée soit comme pertinente, soit comme non pertinente. Il serait envisageable d'évaluer cette pertinence de manière moins binaire. Les transitions ciblées peuvent parfois être atteintes par des séquences relativement courtes de transitions et par d'autres séquences nettement plus longues. Une évaluation de la pertinence de la transition en fonction de sa capacité à rapprocher rapidement le système d'un état susceptible de couvrir les transitions ciblées pourrait donc être considérée. L'algorithme RCXP pourrait être modifié pour favoriser la poursuite de l'exploration à partir des transitions évaluées comme les plus pertinentes et éventuellement s'arrêter dès que les transitions cibles sont couvertes.

PUBLICATIONS

PUBLICATIONS EN LIEN AVEC LA THÈSE

Dans le cadre de cette thèse, nous avons réalisé cinq publications de recherche :

1. [Julliand et al., 2017a] : cette publication à la conférence internationale PSI, LNCS 10742 présente l'algorithme CXP ainsi que les heuristiques qui lui sont rattachées et fournit des résultats expérimentaux. **N.B.** CXP_{ASO} n'a pas fait l'objet d'une publication.
2. [Julliand et al., 2017b] : les travaux de [Julliand et al., 2017a] sur CXP ont été présentés lors des journées francophones sur les Approches Formelles dans l'Assistance au Développement de Logiciels de 2017.
3. [Julliand et al., 2017c] : cette publication à la conférence TASE 2017 présente l'algorithme BCI et détaille particulièrement l'algorithme d'exploration symbolique en arrière réalisée pour concrétiser les $must^{-}$-transitions.
4. [Julliand et al., 2018a] : cette publication est une extension de l'article [Julliand et al., 2017a] sur l'algorithme CXP pour le journal russe PCS de 2018.
5. [Julliand et al., 2018b] : cette publication à la conférence TAP 2018 présente l'algorithme RCXP ainsi que les règles de conception d'un prédicat de pertinence et de calcul de variant associées.

PUBLICATIONS EN DEHORS DU CADRE DE LA THÈSE

Suite à des travaux effectués avant cette thèse, nous avons également réalisé une autre publication sur le thème de la vérification de propriétés modales sur les réseaux de Petri :

1. [Bride et al., 2016c] : cet article publié à la conférence FMICS-AVoCS 2016 étudie expérimentalement les capacités respectives des solveurs SMT et de la programmation par contraintes appliqués à la vérification de propriétés modales sur les réseaux de Petri.
2. [Bride et al., 2016b] : les travaux de [Bride et al., 2016c] ont été présentées lors des journées francophones sur les Approches Formelles dans l'Assistance au Développement de Logiciels de 2016.
3. [Bride et al., 2018] : l'article [Bride et al., 2016c] à fait l'objet d'une extension et à été publié dans le journal international Software Tools for Technology Transfer de 2018.

RÉSUMÉ

EXPLORATION CONCRÉTISÉE ET PERTINENTE DE SYSTÈMES D'ÉVÉNEMENTS ABSTRAITS EN VUE DE LA GÉNÉRATION AUTOMATIQUE DE TESTS

Les travaux présentés dans cette thèse constituent une contribution aux méthodes de génération automatique de sous-approximations en vue de la génération de tests à partir de modèles. Le test à partir de modèle a pour objectif de garantir la conformité d'une implémentation vis-à-vis d'un modèle, tous les deux conçus à partir des spécifications par deux équipes différentes.

Dans cette thèse, nous proposons l'utilisation des techniques connues d'abstraction à partir de prédicats de modèles comportementaux qui permettent de réduire à un ensemble fini et restreint l'espace d'états manipulé. Nous proposons d'extraire les prédicats d'abstraction à partir de l'objectif de test afin que les tests générés couvrent les comportements ciblés par ce dernier. Cependant, le calcul d'une abstraction entraîne une perte d'information de l'atteignabilité par rapport au modèle initial. Nos objectifs sont donc dans un premier temps de calculer efficacement une abstraction de modèle aussi représentative que possible d'un objectif de test. Dans un second temps, nous cherchons à extraire à partir de cette abstraction des exécutions instanciables sur le modèle avant abstraction, ciblant les comportements à tester, et visant la couverture des états et des transitions du modèle abstrait.

Nos contributions sont les suivantes. Nous définissons une méthode de génération de tests combinant plusieurs algorithmes qui permettent d'obtenir une bonne couverture structurelle d'une abstraction de modèle comportemental non déterministe. Nous proposons dans un premier temps un algorithme calculant une abstraction de modèle par prédicats issus d'un objectif de test exprimé sous la forme d'une propriété temporelle. Cet algorithme calcule une sous-approximation du modèle en couvrant les états et les transitions abstraits du modèle. Il applique plusieurs heuristiques et diverses techniques d'exploration ayant pour but d'augmenter le nombre d'instances effectivement atteintes. Dans un second temps, nous proposons d'améliorer par le biais de deux autres algorithmes la couverture structurelle obtenue par cette première sous-approximation. Le premier, entièrement automatisé, tire parti des modalités des transitions abstraites qui fournissent des propriétés d'atteignabilité. Le second algorithme d'extension de la sous-approximation fait appel à l'expertise du testeur qui doit, à partir des transitions non couvertes, énoncer un prédicat de pertinence qui guide et limite l'exploration et l'instanciation réalisées. Nous définissons un ensemble de règles permettant d'énoncer ce prédicat de pertinence et de calculer un variant garantissant la terminaison de l'algorithme d'exploration. Ces deux algorithmes complètent la sous-approximation obtenue auparavant par des exécutions instanciables. Enfin, nous mettons en œuvre une démarche expérimentale pour l'évaluation de la qualité de la méthode, portant sur cinq études de cas.

Mots-clés : Systèmes réactifs, Abstraction de modèles, Test à partir de modèles, Génération de tests pertinents, Couverture structurelle

BIBLIOGRAPHIE

[Abrial, 1996] Abrial, J.-R. (1996). **The B-book : Assigning Programs to Meanings**. Cambridge University Press, New York, NY, USA.

[Abrial, 2010] Abrial, J.-R. (2010). **Modeling in Event-B : system and software engineering**. Cambridge University Press.

[Baldoni et al., 2018] Baldoni, R., Coppa, E., D'elia, D. C., Demetrescu, C., et Finocchi, I. (2018). **A survey of symbolic execution techniques**. *ACM Computing Surveys (CSUR)*, 51(3) :50.

[Ball, 2005] Ball, T. (2005). **A theory of predicate-complete test coverage and generation**. Dans de Boer, F. S., Bonsangue, M. M., Graf, S., et de Roever, W.-P., éditeurs, *Formal Methods for Components and Objects*, pages 1–22, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Ball et al., 2005] Ball, T., Kupferman, O., et Yorsh, G. (2005). **Abstraction for falsification**. Dans *CAV*, pages 67–81.

[Barnett et al., 2005] Barnett, M., Leino, K. R. M., et Schulte, W. (2005). **The spec# programming system : An overview**. Dans *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 de *Lecture Notes in Computer Science*, pages 49–69. Springer.

[Barrett et al., 2017] Barrett, C., Fontaine, P., et Tinelli, C. (2017). **The SMT-LIB Standard : Version 2.6**. Rapport technique, Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[Bernard et al., 2004] Bernard, E., Legeard, B., Luck, X., et Peureux, F. (2004). **Generation of test sequences from formal specifications : Gsm 11-11 standard case study**. *Softw. Pract. Exper.*, 34(10) :915–948.

[Bert et al., 2000] Bert, D., et Cave, F. (2000). **Construction of finite labelled transition systems from b abstract systems**. Dans Grieskamp, W., Santen, T., et Stoddart, B., éditeurs, *Integrated Formal Methods*, pages 235–254, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Blackburn et al., 2001] Blackburn, M., Busser, R., Nauman, A., Knickerbocker, R., et Kasuda, R. (2001). **Mars polar lander fault identification using model-based testing**. Dans *Proceedings 26th Annual NASA Goddard Software Engineering Workshop*, pages 128–135.

[Blackburn et al., 1996] Blackburn, M. R., et Busser, R. D. (1996). **T-vec : A tool for developing critical systems**. Dans *In Proceedings of the 1996 Annual Conference on Computer Assurance (COMPASS 96)*, pages 237–249. IEEE Computer Society Press.

[Bradley et al., 2010] Bradley, A. R., et Manna, Z. (2010). **The Calculus of Computation : Decision Procedures with Applications to Verification**. Springer Publishing Company, Incorporated, 1ère édition.

[Bride et al., 2016a] Bride, H., Julliand, J., et Masson, P.-A. (2016a). **Tri-modal under-approximation for test generation**. *Sci. Comput. Program.*, 132(P2) :190–208.

[Bride et al., 2016b] Bride, H., Kouchnarenko, O., Peureux, F., et Voiron, G. (2016b). **Comparing clp(fd) and smt approaches applied to workflow nets verification**. Dans Hurault, A., et Stouls, N., éditeurs, *15èmes Journées Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL 2016)*, pages 11 – 12, Besançon, France. FEMTO-ST.

[Bride et al., 2016c] Bride, H., Kouchnarenko, O., Peureux, F., et Voiron, G. (2016c). **Workflow nets verification : Smt or clp ?** Dans *Critical Systems : Formal Methods and Automated Verification, FMICS-AVoCS 2016*, volume 9933 de *Lecture Notes in Computer Science (LNCS)*, pages 39 – 55, Pisa, Italy. Springer.

[Bride et al., 2018] Bride, H., Kouchnarenko, O., Peureux, F., et Voiron, G. (2018). **Assessing SMT and CLP approaches for workflow nets verification**. *STTT*, 20(4) :467–491.

[Broy et al., 2005] Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., et Pretschner, A. (2005). **Model-based testing of reactive systems**. Dans *Volume 3472 of Springer LNCS*. Springer.

[Büchi, 1990] Büchi, J. R. (1990). **On a Decision Method in Restricted Second Order Arithmetic**, pages 425–435. Springer New York, New York, NY.

[Bue, 2011] Bue, P.-C. (2011). **Contributions à la génération automatique de tests à partir de critères de sélection dynamique par abstraction de modèles**. PhD thesis, Université de Franche-Comté.

[Chaki et al., 2003] Chaki, S., Clarke, E., Groce, A., Jha, S., et Veith, H. (2003). **Modular verification of software components in c**. Dans *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 385–395, Washington, DC, USA. IEEE Computer Society.

[Chan et al., 1998] Chan, W., Anderson, R. J., Beame, P., Burns, S., Modugno, F., Notkin, D., et Reese, J. D. (1998). **Model checking large software specifications**. *IEEE Transactions on Software Engineering*, 24(7) :498–520.

[Clarke et al., 2003] Clarke, E., Grumberg, O., Jha, S., Lu, Y., et Veith, H. (2003). **Counterexample-guided abstraction refinement for symbolic model checking**. *J. ACM*, 50(5) :752–794.

[Clarke et al., 1982] Clarke, E. M., et Emerson, E. A. (1982). **Design and synthesis of synchronization skeletons using branching-time temporal logic**. Dans *Logic of Programs, Workshop*, pages 52–71, Berlin, Heidelberg. Springer-Verlag.

[Clarke et al., 1999] Clarke, Jr., E. M., Grumberg, O., et Peled, D. A. (1999). **Model Checking**. MIT Press, Cambridge, MA, USA.

[Dams et al., 2004] Dams, D., et Namjoshi, K. S. (2004). **The existence of finite abstractions for branching time model checking**. Dans *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.*, pages 335–344.

[de Alfaro et al., 2004] de Alfaro, L., Godefroid, P., et Jagadeesan, R. (2004). **Three-valued abstractions of games : uncertainty, but with precision**. Dans *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.*, pages 170–179.

[de Moura et al., 2008] de Moura, L., et Bjørner, N. (2008). **Z3 : An efficient smt solver**. Dans Ramakrishnan, C. R., et Rehof, J., éditeurs, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Dijkstra et al., 1990] Dijkstra, E. W., et Scholten, C. S. (1990). **Predicate Calculus and Program Semantics**. Springer-Verlag New York, Inc., New York, NY, USA.

[Dillon et al., 1994] Dillon, L. K., Kutty, G., Moser, L. E., Melliar-Smith, P. M., et Ramakrishna, Y. S. (1994). **A graphical interval logic for specifying concurrent systems**. *ACM Transactions on Software Engineering and Methodology*, 3 :131–165.

[Dwyer et al., 1999] Dwyer, M. B., Avrunin, G. S., et Corbett, J. C. (1999). **Patterns in property specifications for finite-state verification**. Dans *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 411–420, New York, NY, USA. ACM.

[Dwyer et al., 1997] Dwyer, M. B., Carr, V., et Hines, L. (1997). **Model checking graphical user interfaces using abstractions**. Dans *In Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 244–261. Springer-Verlag.

[El-Far et al., 2002] El-Far, I. K., et Whittaker, J. A. (2002). **Model-based software testing**. *Encyclopedia of Software Engineering*.

[Emerson et al., 1985] Emerson, E., et Halpern, J. Y. (1985). **Decision procedures and expressiveness in the temporal logic of branching time**. *Journal of Computer and System Sciences*, 30(1) :1 – 24.

[Fantechi et al., 2012] Fantechi, A., Fokkink, W., et Morzenti, A. (2012). **Some trends in formal methods applications to railway signaling**. Dans *FMICS 2012*.

[Fitzgerald et al., 2007] Fitzgerald, J. S., Larsen, P. G., et Verhoef, M. (2007). **Vienna development method**. *Wiley Encyclopedia of Computer Science and Engineering*, pages 1–11.

[Godefroid et al., 2001] Godefroid, P., Huth, M., et Jagadeesan, R. (2001). **Abstraction-based model checking using modal transition systems**. Dans *Proceedings of the 12th International Conference on Concurrency Theory*, volume 2154 de *CONCUR '01*, pages 426–440, Berlin, Heidelberg. Springer-Verlag.

[Graf et al., 1997] Graf, S., et Saidi, H. (1997). **Construction of abstract state graphs with pvs**. Dans Grumberg, O., éditeur, *Computer Aided Verification*, volume 1254 de *Lecture Notes in Computer Science (LNCS)*, pages 72–83, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Grieskamp et al., 2002] Grieskamp, W., Gurevich, Y., Schulte, W., et Veanes, M. (2002). **Generating finite state machines from abstract state machines**. Dans *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 112–122, New York, NY, USA. ACM.

[Henzinger et al., 2002] Henzinger, T. A., Jhala, R., Majumdar, R., et Sutre, G. (2002). **Lazy abstraction**. *SIGPLAN Not.*, 37(1) :58–70.

[Hoare, 1969] Hoare, C. A. R. (1969). **An axiomatic basis for computer programming**. *Commun. ACM*, 12(10) :576–580.

[Holzmann, 2003] Holzmann, G. (2003). **Spin Model Checker, the : Primer and Reference Manual**. Addison-Wesley Professional, 1ère édition.

[Howden, 1977] Howden, W. E. (1977). **Symbolic testing and the dissect symbolic evaluation system**. *IEEE Transactions on Software Engineering*, SE-3(4) :266–278.

[Huizinga et al., 2007] Huizinga, D., et Kolawa, A. (2007). **Automated defect prevention : best practices in software management**. John Wiley & Sons.

[Jaffuel et al., 2006] Jaffuel, E., et Legeard, B. (2006). **Leirios test generator : Automated test generation from b models**. Dans *Proceedings of the 7th International Conference on Formal Specification and Development in B*, B'07, pages 277–280, Berlin, Heidelberg. Springer-Verlag.

[Julliand et al., 2017a] Julliand, J., Kouchnarenko, O., Masson, P.-A., et Voiron, G. (2017a). **Approximating event system abstractions by covering their states and transitions**. Dans Petrenko, A. K., et Voronkov, A., éditeurs, *PSI'17, A.P. Ershov Informatics Conference (the PSI Conference Series, 11th edition)*, volume 10742 de *Lecture Notes in Computer Science (LNCS)*, pages 211 – 226, Moscow, Russia.

[Julliand et al., 2017b] Julliand, J., Kouchnarenko, O., Masson, P.-A., et Voiron, G. (2017b). **Approximer des abstractions de systèmes d'événements en couvrant leurs états et leurs transitions**. Dans IDANI, A., et KOSMATOV, N., éditeurs, *16ème Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL 2017)*, pages 93 – 94, Montpellier, France.

[Julliand et al., 2017c] Julliand, J., Kouchnarenko, O., Masson, P.-A., et Voiron, G. (2017c). **Two under-approximation techniques for 3-modal abstraction coverage of event systems : Joint effort ?** Dans *11th IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2017)*, Nice, France. IEEE.

[Julliand et al., 2018a] Julliand, J., Kouchnarenko, O., Masson, P.-A., et Voiron, G. (2018a). **Test generation from event system abstractions to cover their states and transitions**. *Programming and Computer Software*, 44(1) :1 – 14.

[Julliand et al., 2018b] Julliand, J., Kouchnarenko, O., Masson, P.-A., et Voiron, G. (2018b). **Under-approximation generation driven by relevance predicates and variants**. Dans *12th International Conference on Tests & Proofs (TAP 2018)*, volume 10889 de *Lecture Notes in Computer Science (LNCS)*, pages 63 – 82, Toulouse, France.

[King, 1975] King, J. C. (1975). **A new approach to program testing**. *SIGPLAN Not.*, 10(6) :228–233.

[King, 1976] King, J. C. (1976). **Symbolic execution and program testing**. *Commun. ACM*, 19(7) :385–394.

[Knight, 2002] Knight, J. C. (2002). **Safety critical systems : challenges and directions**. Dans *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 547–550.

[Lee et al., 1992] Lee, D., Yannakakis, M., et Yannakakis, M. (1992). **Online minimization of transition systems (extended abstract)**. Dans *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 264–274, New York, NY, USA. ACM.

[Legeard et al., 2004] Legeard, B., Peureux, F., et Utting, M. (2004). **Controlling test case explosion in test generation from b formal models : Research articles**. *Softw. Test. Verif. Reliab.*, 14(2) :81–103.

[Lougee, 2001] Lougee, H. (2001). **Software considerations in airborne systems and equipment certification**.

[Manna et al., 1992] Manna, Z., et Pnueli, A. (1992). **The Temporal Logic of Reactive and Concurrent Systems**. Springer-Verlag, Berlin, Heidelberg.

[N. Naumovich et al., 1996] N. Naumovich, G., Clarke, L., et J. Osterweil, L. (1996). **Verification of communication protocols using data flow analysis**. *ACM SIGSOFT Software Engineering Notes*, 21.

[Pasareanu et al., 2007] Pasareanu, C. S., Pelánek, R., et Visser, W. (2007). **Predicate abstraction with under-approximation refinement**. *CoRR*, abs/cs/0701140.

[Păsăreanu et al., 2009] Păsăreanu, C. S., et Visser, W. (2009). **A survey of new trends in symbolic execution for software testing and analysis**. *International journal on software tools for technology transfer*, 11(4) :339.

[Philipps et al., 2003] Philipps, J., Pretschner, A., Slotosch, O., Aiglstorfer, E., Kriebel, S., et Scholl, K. (2003). **Model-based test case generation for smart cards**. *Electronic Notes in Theoretical Computer Science*, 80 :170–184.

[Pnueli, 1981] Pnueli, A. (1981). **The temporal semantics of concurrent programs**. *Theoretical Computer Science*, 13(1) :45 – 60. Special Issue Semantics of Concurrent Computation.

[Pressman, 2005] Pressman, R. S. (2005). **Software engineering : a practitioner's approach**. Palgrave Macmillan.

[S. Boyer et al., 1975] S. Boyer, R., Elspas, B., et Levitt, K. (1975). **Select—a formal system for testing and debugging programs by symbolic execution**. *ACM SIGPLAN Notices*, 10 :234–245.

[Sen et al., 2006] Sen, K., et Agha, G. (2006). **Cute and jcute : Concolic unit testing and explicit path model-checking tools**. Dans *Computer Aided Verification - 18th International Conference, CAV 2006, Proceedings*, volume 4144 LNCS de *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 419–423, Germany. Springer Verlag.

[Shoham et al., 2007] Shoham, S., et Grumberg, O. (2007). **A game-based framework for ctl counterexamples and 3-valued abstraction-refinement**. *ACM Trans. Comput. Logic*, 9(1).

[Somenzi et al., 2000] Somenzi, F., et Bloem, R. (2000). **Efficient büchi automata from ltl formulae**. Dans Emerson, E. A., et Sistla, A. P., éditeurs, *Computer Aided Verification*, volume 1855 de *Lecture Notes in Computer Science (LNCS)*, pages 248–263, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Spivey, 1989] Spivey, J. M. (1989). **The z notation - a reference manual**. Dans *Prentice Hall International Series in Computer Science*.

[Tarjan, 1971] Tarjan, R. (1971). **Depth-first search and linear graph algorithms**. Dans *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 114–121.

[Thimbleby, 2003] Thimbleby, H. (2003). **The directed chinese postman problem**. *Software : Practice and Experience*, 33(11) :1081–1096.

[Utting, 2005] Utting, M. (2005). **Position paper : model-based testing**. *Verified Software : Theories, Tools, Experiments. ETH Zürich, IFIP WG*, 2.

[Utting et al., 2012] Utting, M., Pretschner, A., et Legeard, B. (2012). **A taxonomy of model-based testing approaches**. *Softw. Test., Verif. Reliab.*, 22 :297–312.

[Veanes et al., 2003] Veanes, M., et Yavorsky, R. (2003). **Combined algorithm for approximating a finite state abstraction of a large system**. Dans *ICSE'03/Scenarios Workshop*, pages 86–91.

[Visser et al., 2005] Visser, W., et Mehlitz, P. (2005). **Model checking programs with java pathfinder**. Dans Godefroid, P., éditeur, *Model Checking Software*, volume 3639 de *Lecture Notes in Computer Science (LNCS)*, pages 27–27, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Wing et al., 1997] Wing, J. M., et Vaziri-Farahani, M. (1997). **A case study in model checking software systems**. *Science of Computer Programming*, 28(2) :273 – 299. Formal Specifications : Foundations, Methods, Tools and Applications.

TABLE DES FIGURES

LISTE DES TABLES

LISTE DES DÉFINITIONS

Document réalisé avec LaTeX et :
le style LaTeX pour Thèse de Doctorat créé par S. Galland — http://www.multiagent.fr/ThesisStyle
la collection de paquets `tex-upmethodology` — http://www.arakhne.org/tex-upmethodology/