Génération automatique de jeux de tests avec analyse symbolique des données pour les systèmes embarqués

Mariem Abdelmoula

UNIVERSITE DE NICE SOPHIA ANTIPOLIS

ECOLE DOCTORALE STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE

Pour l'obtention du grade de

Docteur en Sciences

de l'Université de Nice-Sophia Antipolis

Mention : Informatique

Présentée par

Mariem Abdelmoula

Génération automatique de jeux de tests avec analyse symbolique des données pour les systèmes embarqués

Directeur de thèse : **M. Michel Auguin**

Co-encadrant : **M. Daniel Gaffé**

Soutenue le 18 Décembre 2014

JURY

M. Jean-Claude Fernandez	Professeur, Verimag, France	Rapporteur
M. Yvon Trinquet	Professeur, Université de Nantes, France	Rapporteur
M. Bruno Robisson	Chercheur, Ecole des Mines de St Etienne, France	Examinateur
M. Robert de Simone	Directeur de Recherche, INRIA, France	Examinateur
M. Daniel Gaffé	Maître de conférence, Université de Nice,France	Co-encadrant
M. Michel Auguin	Directeur de recherche, CNRS UMR 7248, France	Directeur de thèse
M. Eric Gerbault	Directeur R&D électronique, ASK, France	Invité

À mes parents,

À mes frères,

À ma sœur,

À mon mari,

À tous les membres de ma famille sans aucune exception.

Et à tous ceux que ma réussite leur tient à cœur.

Je dédie ce travail.

Remerciement

La réalisation de ce mémoire a été possible grâce au concours de plusieurs personnes à qui je voudrais témoigner toute ma reconnaissance. J'aimerai remercier toutes les personnes qui ont contribuées de près ou de loin à l'achèvement de cette précieuse expérience.

Je tiens à exprimer ma profonde reconnaissance envers Monsieur Daniel Gaffé, pour m'avoir encadrée durant mes trois années de thèse. Je le remercie particulièrement pour m'avoir dirigée, guidée et conseillée dans une ambiance, toujours, formidable. Il a su à chaque fois me donner les meilleurs conseils et les bonnes orientations lorsque j'étais en proie du doute ou de l'hésitation. J'apprécie énormément ses hautes qualités scientifiques et ses valeurs humaines exemplaires et pour tout le temps qu'il m'a consacré malgré ses responsabilités. J'ai pris beaucoup de plaisir à travailler avec lui. Les mots ne peuvent exprimer toute ma gratitude.

Je suis aussi profondément reconnaissante à Monsieur Michel Auguin, mon directeur de thèse pour ses orientations et ses judicieux conseils, associés à sa sagesse qui m'ont permis d'améliorer la qualité de mes travaux et du présent mémoire. Je tiens à lui exprimer mon respect, le plus profond et mes remerciements, les plus sincères.

Mes remerciements d'adressent également aux membres de jury : Je tiens à remercier Monsieur Jean-Claude Fernandez, Professeur à l'Université de Grenoble et Monsieur Yvon Trinquet, Professeur à l'Université de Nantes, pour l'honneur qu'ils m'ont fait en acceptant d'être mes rapporteurs de thèse. Je tiens à remercier également Monsieur Bruno Robisson, Chercheur à l'Ecole des Mines de St Etienne, et Monsieur Robert de Simone, Directeur de Recherche à L'INRIA Sophia Antipolis, qui m'ont honorée par leur présence dans le jury de ce travail. Leur participation dans l'évaluation de ce travail est d'une très grande valeur.

Je tiens aussi à remercier Madame Annie Ressouche pour son amabilité, ses conseils, ses remarques et sa contribution durant les réunions.

Ce travail a été réalisé dans le cadre du projet FastPass. Un grand merci à tous les partenaires académiques et industriels de ce projet. Je tiens à remercier en particulier Monsieur Eric Gerbault, Responsable du projet FastPass, pour ses remarques et ses orientations dans le cadre de nos réunions de projet.

Je tiens à remercier tous les membres du laboratoire LEAT, dans lequel j'ai passé mes trois ans de thèse. Merci pour tout ce temps passé ! Merci aussi à tous mes collègues du laboratoire qui se reconnaitront ici pour leur amitié et pour toutes les discussions que nous avons eues durant ces années de thèse.

Un énorme et un grand merci à mes parents : Nejib et Serra. Aucun hommage ne pourrait être à la hauteur de l'amour dont ils ne cessent de me combler. Vous m'avez toujours soutenu tout au long de mon parcours et vous n'avez jamais cessé de m'encourager. Vos conseils ont toujours guidé mes pas vers la réussite. Vous avez su m'inculquer le sens de la responsabilité, de l'optimisme et de la confiance en soi face aux difficultés de la vie. Tout ce que j'ai pu réaliser est certes grâce à vous. Merci pour tous vos sacrifices. Je vous aime beaucoup ! Que Dieu vous procure bonne santé et longue vie.

A mes chers frères Walid et Aymen, vous êtes toujours dans mes pensées et dans mon cœur. Que Dieu vous aide à concrétiser toutes vos envies.

A ma chère sœur Salma, Merci pour ton encouragement, ton soutien moral et ta tendresse. Un grand merci à son mari Nabil pour sa gentillesse, sa grande aide et ses conseils. A ma belle nièce Farah, toujours pleine de joie et d'amour, Tu es vraiment une fille adorable ! A mon beau neveu Adam, Tu es très mignon. Tes éclats de rire égalent la joie que vous me procurez, Que Dieu vous protège.

A mon cher mari Bassem, qui a su me soutenir avec plein d'amour et délicatesse pendant les moments difficiles de stress et de fatigue, merci d'avoir supporté mon stress, merci pour ta présence, ta compréhension, ta patience, ton aide, ton soutien, ton amour...Merci beaucoup ! Que Dieu te protège à mes côtés et nous accorde plein de joie et de succès.

A mes beaux-parents que j'aime Fathi et Rafika, A mon beau-frère Riadh et mes belles-sœurs Rim et Imen. Vous m'avez accueillie dans votre famille avec toute bienveillance. Que Dieu vous bénisse.

A mes chères tantes mon aimable Radia, Monia et Souad, A tous mes cousins et cousines, Que Dieu vous protège et vous prête bonne santé et longue vie.

A toute ma famille, Je vous aime tous !

A ma soeur de cœur, Ahlem, merci pour ta sincère amitié et ta générosité infinie. A son mari Amine. A ma chère Sonia, Nesrine... A tous mes amis en France et en Tunisie. Je vous aime !

Résumé

Un des plus grands défis dans la conception matérielle et logicielle est de s'assurer que le système soit exempt d'erreurs. La moindre erreur dans les systèmes embarqués réactifs peut avoir des conséquences désastreuses et coûteuses pour certains projets critiques, nécessitant parfois de gros investissements pour les corriger, ou même conduire à un échec spectaculaire et inattendu du système. Prévenir de tels phénomènes en identifiant tous les comportements critiques du système est une tâche assez délicate. Les tests en industrie sont globalement non exhaustifs, tandis que la vérification formelle souffre souvent du problème d'explosion combinatoire. Nous présentons dans ce contexte une nouvelle approche de génération exhaustive de jeux de test qui combine les principes du test industriel et de la vérification formelle académique. Notre approche construit un modèle générique du système étudié à partir de l'approche synchrone. Le principe est de se limiter à l'analyse locale des sous-espaces significatifs du modèle. L'objectif de notre approche est d'identifier et extraire les conditions préalables à l'exécution de chaque chemin du sous-espace étudie. Il s'agit ensuite de générer tout les cas de tests possibles à partir de ces pré-conditions. Notre approche présente un algorithme de quasi-aplatissement plus simple et efficace que les techniques existantes ainsi qu'une compilation avantageuse favorisant une réduction considérable du problème de l'explosion de l'espace d'états. Elle présente également une manipulation symbolique des données numériques permettant un test plus expressif et concret du système étudié. Nous avons implémenté notre approche dans un outil appelé GAJE. Afin d'illustrer notre travail, cet outil a été appliqué pour vérifier un projet industriel portant sur la sécurité et la vérification d'une carte à puce sans contact.

Mots clés : Jeux de test, Approche synchrone, Modèle synchrone, Pré-conditions, Couverture de l'espace d'états, Manipulation numérique des données, Backtrack, GAJE.

Abstract

One of the biggest challenges in hardware and software design is to ensure that a system is error-free. Small errors in reactive embedded systems can have disastrous and costly consequences for a project. Preventing such errors by identifying the most probable cases of erratic system behavior is quite challenging. Indeed, tests in industry are overall non-exhaustive, while formal verification in scientific research often suffers from combinatorial explosion problem. We present in this context a new approach for generating exhaustive test sets that combines the underlying principles of the industrial test technique and the academic-based formal verification approach. Our approach builds a generic model of the system under test according to the synchronous approach. The goal is to identify the optimal preconditions for restricting the state space of the model such that test generation can take place on significant subspaces only. So, all the possible test sets are generated from the extracted subspace preconditions. Our approach exhibits a simpler and efficient quasi-flattening algorithm compared with existing techniques and a useful compiled internal description to check security properties and reduce the state space combinatorial explosion problem. It also provides a symbolic processing technique of numeric data that provides a more expressive and concrete test of the system. We have implemented our approach on a tool called GAJE. To illustrate our work, this tool was applied to verify an industrial project on contactless smart cards security.

Keywords : Test Sets, Synchronous Approach, Synchronous Model, Pre-conditions, States Space Covering, Numeric Data Processing, Backtrack, GAJE, Contactless Smart Card.

Table des matières

Table des figures

Liste des algorithmes

Chapitre I

Introduction générale

Ce travail de thèse dans ce mémoire a été réalisé au sein du Laboratoire d'Electronique, Antennes et Télécommunications de l'Université de Nice Sophia Antipolis et du CNRS. Il s'intègre dans le cadre du projet FASTPASS [1]. Un des objectifs de ce projet est de vérifier par l'approche synchrone le fonctionnement du système d'exploitation d'une carte à puce sans contact multi-services. Nous introduisons dans ce chapitre le contexte général de notre étude, la problématique abordée, les principaux objectifs que nous avons fixés ainsi que le plan de ce mémoire.

1 Contexte général

Vérifier automatiquement et de manière formelle le bon fonctionnement d'un logiciel ne parait plus du tout anodin de nos jours vu la complexité croissante des programmes informatiques et leur interaction toujours plus poussée avec leur environnement. Une classe importante de systèmes concernés par de tels problèmes sont les systèmes réactifs : Il s'agit de la classe des systèmes qui réagissent continuellement à leur environnement d'une manière opportune et n'ont pas pour objectif particulier de calculer un résultat définitif. Les systèmes réactifs sont aujourd'hui omniprésents dans notre environnement. Ils varient des simples thermostats au contrôle des centrales nucléaires. La sécurité de tels systèmes est souvent considérée comme critique, cela concerne par exemple des systèmes liés à la santé, des systèmes bancaires, des systèmes militaires, des systèmes de contrôle de vol d'avion, de contrôle dans des centrales nucléaires. La moindre erreur peut donc provoquer des conséquences désastreuses dans ce type de systèmes, nécessitant parfois de grandes sommes d'argents pour les corriger, ou même conduisant à un échec spectaculaire du système.

1. http ://www.pole-scs.org/projet/fastpass

Un exemple bien connu est le blocage du navire "USS Yorktown" de l'armée américaine en septembre 1997. Ce croiseur lance-missiles est équipé d'un système de contrôle et de maintenance automatique qui gère la propulsion des moteurs. Un membre de l'équipage de ce système a entré un "0" par erreur dans la trame de données provoquant une division par zéro. Ceci a entraîné l'immobilisation du navire pendant plusieurs heures au milieu de l'océan coûtant une perte d'environ de 475 millions dollars [Sla98].

De plus pernicieuses erreurs se sont manifestées comme l'explosion de la fusée Ariane 5 peu de temps après son décollage en 1996 [Dow97]. L'incident fût dû à une exception déclenchée lors de la conversion d'un nombre à virgule flottante en un entier non signé dans son système de référence inertiel (SRI) responsable du calcul du mouvement de la fusée. En effet, le module qui a levé l'exception avait fonctionné d'une manière satisfaisante pendant 10 ans sur Ariane 4 sans qu'il n'y ait de dépassement de mémoire "overflow", mais Ariane 5 a une dynamique différente. Comble d'ironie, le module en question était inutile à cette phase du vol. Cette erreur a provoqué des coûts d'un montant d'environ 500 millions de dollars ce qui en fait le bug informatique le plus coûteux de l'histoire.

Outre des pertes financières, les vies humaines entrent en jeu pour certaines erreurs a priori insolites mais qui se sont révélées au final désastreuses : Jusqu'en 2001, une erreur dans le logiciel de programmation du traitement médical a causé au moins 28 morts à l'Institut national du cancer du Panama. En effet, des doses inappropriées de rayonnement ont été mal calculées et envoyées à des patients atteints de cancer [NÓ1].

Cette liste d'erreurs illustre clairement certains dangers potentiels, pertes financières et humaines, de ne pas déceler et corriger les erreurs lors du développement d'un système. En fait, le cerveau humain n'arrive plus, malgré toute l'expérience qu'il a acquis en conception, à se représenter dans sa globalité toutes les réactions et les évolutions possibles d'un système complexe. L'esprit humain a besoin d'une aide externe la plus automatisée possible pour adresser certaines anomalies :

- Complexité de la tâche de spécification : Le "spécificateur" sait en général ce qu'il veut. Par contre, il a souvent beaucoup plus de mal à caractériser tout ce qu'il ne veut pas ;

- Interprétation erronée des spécifications par l'équipe de conception ou entre les équipes indépendantes participant à la conception. Ce problème est récurrent et peut être amplifié dans les communautés qui n'ont pas le même vocabulaire ou la même culture. Le crash de la sonde "Mars Climate Orbiter" sur Mars en 1999 en est une bonne illustration. Ceci a été dû à une conversion inappropriée des échelles de mesure entre l'orbiteur et la station de contrôle terrestre [Hoe12] ;

-Validation incomplète du produit réalisée par rapport à la spécification d'origine. Le surcoût de cette opération est parfois tellement élevé que les concepteurs préfèrent l'ignorer purement et simplement. Ce fut le cas du télescope spatial Hubble qui est parti myope dans l'espace sans avoir d'abord été testé au sol [G.T10].

Considérant cet ensemble de problèmes, il a donc fallu rechercher et adopter des techniques de vérification pouvant répondre aux impératifs de sécurité et trouver des façons de réaliser des systèmes sûrs. A partir de là, la vérification des systèmes critiques est devenue un sujet de recherche en plein essor, qui se base sur une demande industrielle en forte croissance.

2 Problématique

Des visions variées entre le monde industriel et académique ont donné naissance à différentes techniques de vérification formelle et de test.

2.1 Vérification formelle

La vérification formelle fournit des méthodes et techniques permettant de s'assurer de façon "certaine" du fonctionnement d'un système. Ces méthodes s'appuient le plus souvent sur des outils mathématiques vérifiant que le comportement attendu est bien conforme à sa spécification dans tous les cas de figure. La vérification formelle inclut principalement trois techniques de vérification qui sont le plus souvent complémentaires.

2.1.1 Vérification de modèle "Model Checking"

Cette technique assure la vérification formelle au niveau du modèle de spécification. Elle permet de vérifier d'une façon entièrement automatique que le modèle satisfait sa spécification. Elle s'applique particulièrement à tout système pouvant s'interpréter par une machine d'états tels que les systèmes réactifs. Elle vérifie principalement les propriétés :

- d'Accessibilité : une certaine situation ou un état spécifique peut être atteint ;
- d'Invariance : tous les états du système satisfont une bonne propriété ;
- de Sûreté "Safety" : quelque chose de mauvais n'arrive jamais. Il est fréquent de montrer que la propriété contraire est fausse. Dans ce contexte, les "Sat-solveur" tel que Grasp [MSS96], Sato [Zha97] ou Chaff [MMZ+01] arrivent à trouver très rapidement des contre-exemples par le choix d'heuristiques particulièrement ingénieuses de recherche d'accessibilité ;
- de Vivacité "Liveness" : quelque chose de bon finira par arriver ;

- d'Equité "Fairness" : aucune partie du système n'est abandonnée au profit des autres ;
- d'Equité faible : si un processus demande continuellement son exécution, il finira par l'avoir ;
- d'Equité forte : si un processus demande infiniment son exécution, il finira par l'avoir ;
- d'Absence de blocage "Deadlock" ;
- d'Equivalence de comportement : est-ce que deux systèmes se comportent de la même manière vis à vis des mêmes stimuli ?

Le model checking considère toutes les exécutions du système afin d'assurer une couverture de l'ensemble des états du système pour une certaine propriété. Certes, cette technique souffre d'une explosion combinatoire lorsque les systèmes deviennent trop grands et complexes. Plusieurs travaux en model checking ont été ainsi développés ces dernières années pour réduire ce problème en proposant par exemple des traitements symboliques [McM92] avec des représentations plus compactes des états.

De nombreux outils ont été développés dans l'industrie implémentant cette technologie et ses dérivées comme : Xeve [Bou97], Lesar [HLR92], Nbac [Jea03], Sigali [MBLG00].

2.1.2 Analyse Statique

L'analyse statique assure la vérification formelle du code source implémentant la spécification. Cette technique cherche d'une manière automatique des propriétés sur les variables "intéressantes" du système (signe, intervalle, dépendance). Ces propriétés sont appelées "Invariants". Le but est de prévenir les comportements anormaux tels que le dépassement de capacité ou d'indice et d'optimiser le code en supprimant les parties inutiles, code mort ou redondant, ou en réorganisant les opérations (amélioration du parallélisme, etc). D'un point de vue macroscopique, les concepteurs essaient également de prouver le bon comportement des sous-systèmes offrant un ensemble de services de base ou une abstraction du matériel. Ainsi, l'analyse statique se réfère essentiellement dans ce cas à la théorie de l'interprétation abstraite [Cou78]. Cette dernière est une théorie générale de l'approximation sémantique qui permet de produire des analyses statiques saines par construction (sur-approximation de l'ensemble des comportements du programme). Dans cette catégorie se situent des travaux sur la validation de code noyau très critique [SLQP07], de services transversaux [TWV08] ou de systèmes d'exploitation dans leur globalité [KEH+09].

Ils existent aujourd'hui plusieurs analyseurs statiques, parmi lesquels nous citons ceux portant sur les codes sources en langage C :

– Lint [Dar96] : le premier analyseur statique de code C, toujours d'actualité.
– AiT [FHL+01] : évalue les temps d'exécution maximaux (pour garantir les aspects temps-réel).
– Astrée [BCC+03] : teste l'absence d'erreurs à l'exécution.

2.1.3 Preuve par réécriture

Elle garantit que les compilateurs ou les traducteurs n'altèrent pas les propriétés spécifiées. Dans ce cas, la vérification formelle ne porte plus sur le code source ou exécutable mais sur le compilateur lui-même. Les spécialistes de ce domaine, qualifient cette voie de "Theorem Proving". Des outils tel que Coq [YB04] ou Isabelle [NWP11] permettent d'appliquer symboliquement des théorèmes mathématiques sur les transformations que fait subir le compilateur au code source. Citons par exemple le projet récent CompCert [2] dont l'objectif était de développer un compilateur C réaliste, formellement vérifié par Coq et utilisable dans l'embarqué. Scade [Pap08] est un autre exemple qui présente une version graphique du langage formel Lustre qui a également été prouvée formellement et certifiée.

Tous les outils de preuve ne peuvent pas interpréter les données à la place du programme considéré, car cette action prendrait à elle seule plusieurs centaines d'années (ou plus). Ils donnent des estimations, des intervalles, des sous-espaces où le résultat va se situer : la propriété voulue n'est donc plus cherchée sur le résultat lui-même mais sur le sur-espace qui englobe ce résultat. Il s'agit dans ce contexte de la technique de l'interprétation abstraite.

L'ensemble des techniques de vérification formelle cité ci-dessus est encore en plein développement et ne fournit pas encore de méthodes simples et efficaces dans beaucoup de cas de figure. Certains n'hésitent pas à considérer ce domaine comme "art" autant que "science". Dans certains cas, ces méthodes font face aux problèmes d'indécidabilité et de complexité. Le problème d'indécidabilité est le fait qu'il est parfois mathématiquement impossible de prouver certaines propriétés ou leurs contraires. Avec le problème de complexité, il est possible de prouver une propriété mais la puissance de calcul nécessaire pour y parvenir est hors de portée de nos machines.

En conclusion, toutes ces techniques de vérification formelle présentent un aspect impraticable dû à un manque de mémoire ou de temps. Des solutions alternatives tel que le "Test" sont alors adoptées comme un moyen de vérification de systèmes. Le test est évidemment non exhaustif, mais il permet de découvrir des bugs, et d'accroître la confiance dans un système.

2. http ://compcert.inria.fr/

2.2 Test

Le test est le principal moyen utilisé dans l'industrie pour la validation des comportements des systèmes. Deux types de test se distinguent dans cette optique, à savoir :

- le test fonctionnel (de type boîte noire).
- le test structurel (de type boîte blanche).

Le test fonctionnel examine le comportement d'un programme en le soumettant à des entrées sur lesquelles il vérifie que ce programme correspond aux contraintes définies dans sa spécification. Plus il y a de variables testées, plus grande est la confiance accordée au programme étudié. Le choix des données soumises aux tests est donc crucial pour améliorer la capacité du test à révéler des erreurs. Ce choix est généralement établi en fonction des types d'erreurs recherchés et des caractéristiques des logiciels testés. Citons dans cette philosophie Rational robot [BT00] qui est un exemple d'outil de test fonctionnel pour des applications WEB et ERP (Enterprise Resource Planning).

Par ailleurs, le test structurel vérifie la structure de chaque module composant le programme en examinant son code, afin de générer les jeux de tests associés et de couvrir un maximum de cas possibles en un minimum de cas de tests. QA [QA] est un exemple d'outil de test structurel pour tester rapidement et objectivement la qualité d'un code produit. Dans l'ensemble, le principe de test s'intègre bien dans la logique de vérification modulaire. Un test peut être unitaire s'exécutant sur chaque composant à part, comme il peut être exécuté sur un assemblage de composants testés appelé dans ce cas test d'intégration.

Cependant, il est quasiment impossible de tester et de couvrir la totalité du comportement d'un système complexe. Le fonctionnement n'est jamais garanti et demeure incomplet tant qu'il reste des parties du code non testées. La fin du test est généralement définie par certaines mesures de qualité (détection d'erreurs, critères de couvertures, etc). En réalité, le test est considéré achevé quand les ressources sont épuisées, n'aboutissant à aucune garantie de sûreté. Cette déficience d'exhaustivité a été la cause de plusieurs validations erronées [CGH+95] [CYF94]. Ainsi, le test est une tâche assez complexe et difficile. En outre, cette tâche coûte généralement 50% des coûts totaux d'un projet logiciel [MS04]. Par ailleurs, étant dans le cadre de test des système réactifs qui sont en interaction continue avec leur environnement, la pertinence des entrées de test doit dépendre nécessairement du comportement du système lui-même. Cet aspect de "Feed-back" est très important pour les systèmes réactifs, et il rend la génération de séquences de tests quasiment impossible.

2.3 Manipulation numérique des données

L'application des techniques de vérification formelle ou du test de systèmes complexes est généralement munie d'une représentation booléenne des variables du système comme les Diagrammes de Décision Binaire BDDs [Bry86]. En effet, garder les variables booléennes permet d'exploiter les atouts d'une structure en BDD. S'appuyant sur le succès des BDDs, plusieurs efforts ont été réalisés pour étendre ce concept afin de pouvoir représenter des fonctions à valeurs non-booléennes, telles que les nombres entiers ou réels. Ce type de fonctions se présente en général dans plusieurs applications telles que le calcul des probabilités d'états dans un circuit séquentiel [BFG+93], la programmation linéaire en nombre entiers [LPV06], la vérification des circuits arithmétiques [BC01]. Trouver une façon compacte pour coder les valeurs numériques d'une fonction donnée présente toujours un défi pour les outils existants de test et de vérification.

3 Contribution

En pratique, les industriels procèdent le plus souvent au test de façon pragmatique, tandis que la recherche se concentre plutôt sur la vérification formelle. L'idée dans ce mémoire est de tirer le meilleur profit de ces deux approches qui sont considérées aujourd'hui complémentaires. Notre objectif est d'automatiser la procédure de test en tirant profit des avantages de l'approche formelle. Notre travail se concentre donc sur l'automatisation de la génération des séquences de test à partir d'un modèle spécifié. Ce modèle est présenté comme une entrée du test et il est généré sous forme de machines d'états [Zah80] à partir de la spécification formelle du système étudié. Le résultat du test sera présenté par la suite par des cas de test symboliques qui sont moins complexes que les tests concrets et plus faciles à appréhender .

En outre, automatiser les différentes étapes du test permet de réduire considérablement l'effort requis et d'élever le nombre de tests pouvant être réalisés. L'intérêt majeur de l'intégration des spécifications formelles pour le test est d'assurer une meilleure automatisation dans toutes les phases de conception, d'exécution ou d'évaluation du test. Cet aspect formel permet également de réduire l'intervention humaine qui est la cause d'erreurs involontaires dans plusieurs cas. Le principe d'automatisation apporte plusieurs avantages, il garantit une meilleure efficacité du test, il permet ainsi de réduire ses coûts, et de le rendre par conséquence plus sûr et fiable.

Bien que le test à partir de modèles avec des diagrammes d'états, ou des machines d'états ait été étudié depuis plusieurs décennies, il reste encore de nombreux aspects inexplorés et

des problématiques à résoudre. Les outils existants de génération de jeux de test automatiques (voir section 4 du chapitre III) souffrent encore du problème de l'explosion de l'espace des états, en particulier si le système étudié intègre du traitement numérique des données. Ce problème est encore accentué quand ce traitement amène à des résultats qui ont eux-même un effet significatif sur le comportement du système. A cet effet, l'énumération exhaustive de toutes les situations possibles, en particulier celles non souhaitées est presque impossible. A ces problèmes, les langages synchrones tel que Esterel [BGGL92] sont susceptibles d'apporter une réponse : ils sont capables de comparer symboliquement le comportement logique de différents systèmes. Nous exploitons cette capacité de manipulation symbolique pour générer la liste exhaustive des comportements attendus. Pour cela, nous apportons une contribution dans cette thèse sur différents axes :

- Nous avons utilisé la robustesse des langages synchrones pour concevoir un modèle bien structuré et facile à analyser du système étudié ;
- Nous avons développé par la suite un algorithme de quasi-aplatissement qui permet de rendre le modèle déterministe et facile à exploiter par les outils d'analyse et de vérification formelle. Cet algorithme assure également une génération simple du code ;
- Nous avons établi le traitement numérique des données sans provoquer un dépassement de mémoire (ce qui est l'enjeu majeur de génération automatique de jeux de tests). Ce traitement a permis d'une part le calcul symbolique des données du système et d'autre part la vérification du système (déterminisme, séquences impossibles, identification des cas de tests possibles, etc ;)
- Nous avons élaboré un second algorithme d'exploration symbolique de tout l'espace des états accessibles. Cet algorithme s'appuie sur la représentation efficace du modèle et sur la puissance du traitement numérique pour générer la liste exhaustive de toutes les séquences possibles ;
- Enfin, nous avons mis en place un algorithme de marche en arrière "Backtrack" qui permet de générer tous les cas de tests possibles du comportement d'un système testé à partir des états critiques finaux jusqu'à l'état initial racine.

L'utilisation du test basé sur un modèle n'est pas encore très fréquente dans les projets logiciels industriels. Cette thèse va montrer dans la suite que le test basé sur un modèle formel peut être appliqué avec succès à des applications de taille industrielle.

4 Plan du mémoire

Ce mémoire est organisé en quatre principales parties. Nous présentons dans le chapitre II un rappel sur l'approche synchrone introduisant les systèmes réactifs, la spécification de ces systèmes par les langages synchrones, la notion de compilation de ces langages en machines explicites et implicites, la modélisation des systèmes réactifs à l'aide des machines à états finis et les techniques d'exploration explicites et implicites de l'espace d'états. Nous concluons ce chapitre par la présentation des techniques existantes d'analyse avec et sans aplatissement des machines à états finis.

Nous étudions dans le chapitre III les différentes techniques utilisées pour la représentation des structures de données utilisées dans les tests. Le but de cette étude est d'aboutir à une représentation efficace des données numériques en plus d'une représentation des données booléennes. Nous concluons ce chapitre par une étude comparative de différents outils existants de génération de jeux de tests automatiques y compris ceux qui assurent ou pas le traitement numérique des données.

Nous présentons dans le chapitre IV notre approche de génération automatique de jeux de tests. Nous développons ainsi une représentation efficace du modèle par des machines à états finis, une technique efficace d'analyse des automates (quasi-aplatissement), une génération exhaustive de toutes les séquences possibles, une représentation rigoureuse des données numériques et l'intérêt de cette représentation pour la vérification du système et la génération des cas de test intéressants (opération de marche en arrière).

Le chapitre V présente la mise en œuvre de notre approche qui a donné naissance à un nouvel outil appelé GAJE (Générateur Automatique de jeux de Tests). Nous illustrons notre travail en appliquant l'outil GAJE dans le cadre du projet industriel FASTPASS à la vérification du système d'exploitation d'une carte à puce sans contact. Le résultat de cette application par rapport à celui établi à travers des méthodes classiques a démontré l'efficacité de notre approche et la possibilité de l'exploiter dans des applications de taille industrielle.

Enfin, le dernier chapitre conclut cette thèse et identifie des perspectives de futurs travaux de recherche.

Chapitre II

Conception des systèmes réactifs par l'approche Synchrone

1 Introduction

Nous introduisons dans ce chapitre le principe de l'approche synchrone adaptée à la description des systèmes réactifs critiques. Cette description est basée sur la capacité des langages synchrones à fournir une présentation commode à analyser et à vérifier formellement par la suite. Ce chapitre introduit également la notion de machines à états finis explicites et implicites. Ces machines permettent une modélisation simple et facile des systèmes décrits suivant n'importe quel langage synchrone. Nous décrivons à la fin de ce chapitre les techniques existantes d'exploration d'espace des états et d'analyse avec et sans aplatissement des structures hiérarchiques.

2 Introduction à l'approche Synchrone

2.1 Systèmes réactifs temps réels

Un système temps réel est un système informatique qui doit répondre sous un délai spécifié (temps fini connu ou borné) à des stimuli générés par le monde extérieur. La validité (correction) d'un système temps réel dépend non seulement de la justesse de ses calculs mais aussi du temps auquel les résultats sont produits (contraintes temporelles). Un résultat produit hors échéance devient faux même s'il s'avère logiquement correct.

Les systèmes temps réel se décomposent en trois principales classes d'applications nommées respectivement transformationnelles, interactives et réactives. La première concerne des systèmes qui renvoient des résultats à partir de données en entrée selon un processus de calcul

indépendant de l'environnement puis se terminent (par exemple compilateur). La seconde classe prend en compte, au cours de son exécution, des données produites par l'environnement et réagit à son propre rythme (par exemple un shell). La dernière classe concerne des systèmes qui interagissent continuellement au stimuli de l'environnement. En d'autres termes, le système n'effectue aucun traitement relatif à l'application s'il n'est pas stimulé par son environnement (par exemple un ascenseur). Dans ce mémoire, nous nous intéresserons particulièrement aux tests des systèmes réactifs temps-réel. Finalement, on parle de système réactif temps réel, lorsque le concepteur est capable de prévoir les temps de réactions maximales du système. Ces temps d'exécution notés "T" dans la figure II.1 devront alors être inférieurs ou égaux aux contraintes de temps définies pour l'application (temps borné).

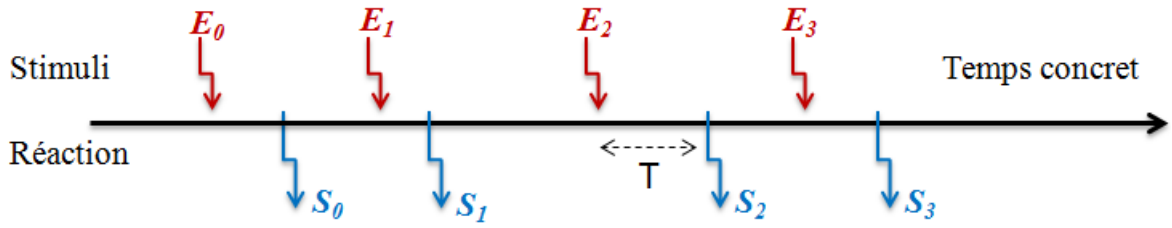


Figure II.1 – Système réactif temps réel.

Très souvent, ces systèmes ont pour objectif le contrôle d'un processus qui communique avec son environnement par l'intermédiaire de signaux, capteurs, thermomètres, claviers, etc. Ces logiciels n'ont pas pour but de calculer un résultat spécifique, mais d'assurer le fonctionnement permanent du processus contrôlé. Une autre caractéristique fréquente de ces systèmes concerne leur comportement global qui dépend de l'interaction de plusieurs sous-systèmes évoluant en parallèle : ceux sont des systèmes distribués. Ceci impose la mise en place de mécanismes de synchronisation et d'ordonnancement spécifiques aux contraintes temps réel. En ce qui concerne l'aspect temps réel, le paramètre temps intervient généralement de manière explicite : ces systèmes ne peuvent pas attendre ! Aussi, le déterminisme est une propriété souvent voulue pour les systèmes réactifs permettant ainsi un "débogage" plus aisé si le programme réagit de la même manière aux mêmes séquences d'entrées. Finalement, un système réactif doit être sûr, fiable et robuste du fait que ce type de système intervient généralement dans des domaines ou applications critiques. En d'autres termes, toutes les évolutions de leur état ainsi que leurs conséquences sur les données fournies doivent être prévisibles. Le risque de panne doit être le plus faible possible. Un système réactif doit garantir un mode de fonctionnement sécurisé assurant les fonctions essentielles en cas de panne ou d'imprévus externes.

2.2 Systèmes réactifs temps réels asynchrones

Le monde concret est totalement asynchrone dans le sens où il est peuplé de "retards", "délais", "dérives", etc. Modéliser et réaliser les systèmes réactifs suivant une approche asynchrone est donc une tâche assez compliquée. Il est incontestablement nécessaire d'inclure des mécanismes appropriés afin de pallier à ses problèmes. Il faut par exemple, dater les signaux, ordonner les exécutions des actions, attendre les signaux dans une plage de temps. D'autre part, réaliser un système réactif complexe, intégrant beaucoup d'entrées/sorties, en adaptant une conception "mono-bloc" devient alors une tâche très contraignante. Une solution classique est de concevoir ce système de façon parallèle et hiérarchique. La figure II.2 montre un exemple de réalisation de systèmes réactifs suivant cette approche.

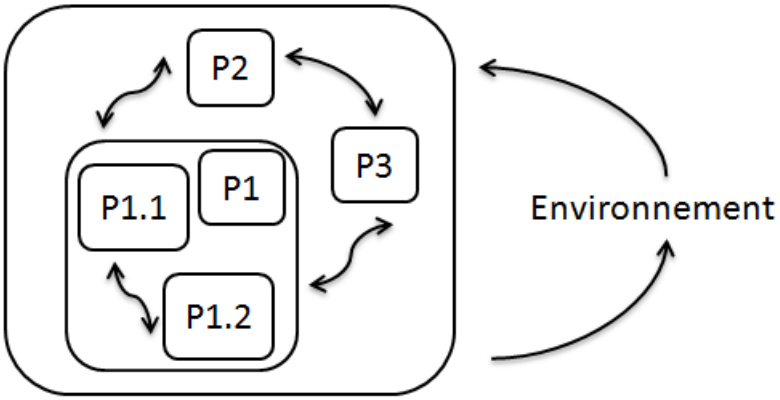


Figure II.2 – Systèmes réactifs parallèles et hiérarchiques.

En effet, le fonctionnement souhaité consiste à considérer localement chaque sous-système comme un système temps-réel. Cependant, avec l'absence de mécanismes adéquats comme "sémaphore" et "rendez vous", le parallélisme rend les exécutions non déterministes. Ceci peut causer un risque de blocage, un ordonnancement des actions inadéquat, des temps d'exécutions imprévisibles, etc.

La conception des systèmes réactifs temps réels est donc très largement influencée par les problèmes liées a l'asynchronisme du monde extérieur et au parallélisme. Même s'il existe des solutions théoriquement efficaces pour pallier à ces problèmes, elles rajoutent néanmoins une grande complexité au système réalisé et accroissent considérablement les temps de conception et les risques de bug. Cependant, dans le cas où les systèmes imposent une conception asynchrone, les concepteurs n'ont d'autres choix que de s'attaquer à ces difficultés.

2.3 L'approche synchrone : le synchronisme fort

Dans ce contexte, une nouvelle approche de conception ("l'Ecole synchrone de conception") est apparue dans les années 80 portée en particulier par M. Gérard Berry, aujourd'hui membre de l'Académie des Sciences, et soutenue à l'époque par de grands centres de recherche dont INRIA, Verimag, IRISA ainsi que de grandes industries de l'aéronautique comme Dassault et Airbus industrie. Que ce soit pour les problèmes liés a l'environnement ou au parallélisme, la difficulté à appréhender est liée au temps. L'idée de l'approche synchrone s'appuie sur le synchronisme fort. Il s'agit principalement d'abstraire le temps afin d'en simplifier tous les problèmes qui en découlent.

Hypothèse synchrone forte : L'hypothèse de synchronisme fort suppose l'existence d'un temps global (une discrétisation du temps réel). Le "temps physique", qui joue un rôle si particulier dans les approches asynchrones, n'a plus de raison d'être privilégié : pour le manipuler il suffit d'être sensible à un signal logique d'entrée qui caractérise un état important du système. Cette hypothèse suppose également que les actions (calcul et communication) soient atomiques et durent un temps nul. Chaque réaction d'un système réactif est donc instantanée. Cela implique que deux actions exécutées à la même date (du temps global) seront considérées comme simultanées comme l'illustre la figure II.3. Ainsi, le système produit ses sorties dans le même instant logique que ses entrées.

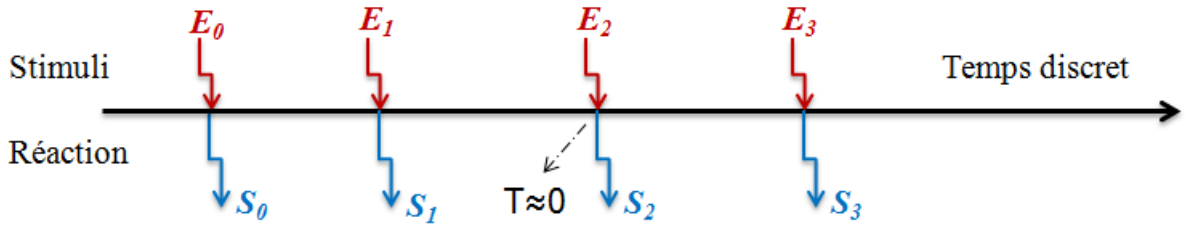


Figure II.3 – Hypothèse synchrone.

L'hypothèse synchrone ne peut être effectuée que dans le cas ou les temps d'exécution du système sont courts devant la dynamique de l'environnement observé. C'est à dire que le temps minimal entre deux stimuli extérieurs doit être supérieur au temps maximal d'exécution du système. Cette condition doit être impérativement vérifiée pour que cette abstraction reste valide. Elle garantit aussi que le système assure des évolutions de type atomique à chaque arrivée de nouvel évènement. L'atomicité rend le système insensible aux nouvelles entrées tant qu'il n'a pas fini de réagir aux entrées précédentes. La figure II.4 montre un exemple de mise en défaut de l'approche synchrone.

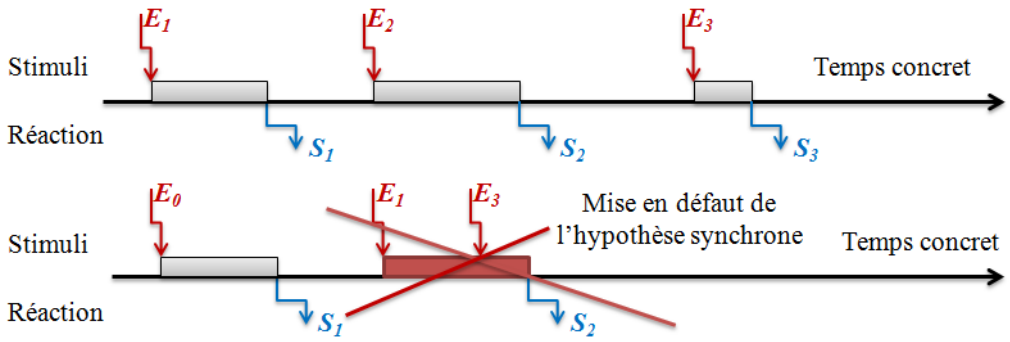


Figure II.4 – Vérification de l'hypothèse synchrone.

Cette vision comporte principalement deux intérêts : La complexité de programmation liée à la notion de temps est réduite. Cette approche suppose que le temps est discret et tous les évènements intervenants entre les dates t1 et t2 seront considérés comme être intervenu à la même date t2. En effet, les différentes parties du système communiquant par diffusion instantanée d'information, s'interrompent instantanément pour prendre en compte les nouveaux évènements. La notion de temps multiforme introduit une souplesse de programmation et une possibilité d'abstraction sur les applications à développer. Pour ce faire, il existe de nombreuses primitives temporelles dans l'approche synchrone qui permettent de manipuler les exceptions, les délais, ainsi que la suspension d'exécution. Ainsi dans l'approche synchrone il est possible de supprimer l'indéterminisme lié aux exécutions parallèles. En effet, tous les processus débutent à la même date et leurs temps d'exécution sont identiques et égaux à zéro. Un seul résultat est alors possible si le système est constructif. La disparation des phénomènes transitoires garantit le déterminisme.

Cette approche apporte une grande expressivité ainsi qu'une sémantique formalisée et permet de programmer réellement les systèmes réactifs et de simplifier considérablement leur réalisation, leur mise au point et l'étude de leur comportement. Elle favorise aussi l'analyse statique et la vérification des propriétés du système. De même, la compilation d'un programme synchrone génère un automate à états finis séquentiel, déterministe et en général minimal. L'avantage est double. D'une part, le code produit est efficace. En effet, le programme parallèle initial est traduit en un programme équivalent purement séquentiel et l'exécution d'un tel programme n'introduit pas de gestion de processus et les communications internes sont compilées et codées dans l'automate. Comme le temps maximal de la réaction est prévisible, la validité de l'hypothèse synchrone peut être réellement testée. D'autre part, une fois l'automate construit, il est possible d'utiliser des outils qui travaillent sur les automates pour pouvoir procéder à des vérifications formelles et s'assurer que le programme implémente bien les spécifications voulues.

En conclusion, le déroulement du temps n'est pas lié à l'exécution réelle d'un programme mais uniquement à l'occurrence du flux d'évènements qu'il traite. Néanmoins cette vision nécessite une machine d'exécution support afin de mettre en forme les stimuli en entrée et en sortie.

3 Les langages synchrones

3.1 Introduction aux langages synchrones

Les langages synchrones sont apparus avec l'approche synchrone de conception. Ce sont des langages de programmation dédiés à la réalisation des systèmes réactifs embarqués à la fois complexes et critiques. Ils sont essentiellement conçus pour être simples et permettent une vérification aisée. Ils se fondent ainsi sur un formalisme mathématique spécifique. Il existe deux grandes familles qui se distinguent par la façon d'appréhender le temps discret. En effet, elles nécessitent un gestionnaire dédié, connu sous le nom de "machine d'exécution" [AB00] afin de gérer les aspects temps réel et construire des évènements logiques qui vont servir d'horloge pour le système synchrone.Les deux familles de langages sont :
- Langages déclaratifs et orientés flot de données dont les plus connus sont Lustre [HCRP91] et Signal [GLGB87]. Ils dérivent principalement des techniques de traitement du signal représentant les systèmes sous forme équationnelle ;
- Langages impératifs orientés vers les systèmes de contrôle. Ils proposent des opérations en termes de séquences d'instructions, principalement concurrentes, exécutées pour changer l'état du système. Esterel [BGGL92] est le langage synchrone impératif par excellence. Son formalisme graphique SyncCharts [And96] permet de présenter d'une manière graphique tout système synchrone en produisant un code Esterel équivalent.

Tout ces langages assurent la modélisation du comportement d'un système synchrone de façon déterministe. Ils permettent de construire par la suite la partie contrôle sous forme d'automates à états finis ou d'équations logiques. Ils définissent également des sémantiques de comportement rigoureuses et vérifient que le code produit au niveau implantation sur la machine cible aura rigoureusement le même comportement que le modèle de départ. L'intérêt d'une sémantique formelle permet non seulement de lever toute ambigüité d'interprétation comportementale, mais de plus elle facilite la mise en œuvre d'outils de preuves formelles de comportement. Le compilateur d'un langage synchrone doit préserver par construction les propriétés importantes pour le bon fonctionnement des systèmes critiques : absence d'états non voulus, exécution déterministe du code, génération de code parallèle, génération de code sé-

quentiel s'exécutant en temps et mémoire bornés, etc. En somme, les langages synchrones sont en premier lieu des langages de spécification. Leur sémantique leur permet ainsi de rendre ces spécifications directement compilables. Ces langages ont connu un grand succès dans divers domaines industriels, pour la programmation de contrôle des avions, des trains, des éoliennes. Par exemple, l'outil SCADE [Ber07] issu du langage de programmation universitaire synchrone Lustre a développé le système de commande de vol des Airbus.

3.2 Lustre

Lustre [HCRP91] est un langage synchrone déclaratif de flot de données développé par Nicolas Halbwachs et Paul Caspi au laboratoire VERIMAG depuis 1984. Une modélisation d'un système en Lustre est basée sur la notion de nœuds réactifs qui communiquent de manière instantanée entre eux. Lustre est un langage fonctionnel : un nœud est caractérisé par un ensemble d'équations qui gèrent l'ensemble des sorties et des variables locales à partir des entrées du nœud et de son état interne courant. De ce point de vue, Lustre présente un vrai langage de "flux de données" puisque les flux d'entrées déterminent complètement le comportement du programme. Ce formalisme mathématique a une vision graphique par schémas de blocs dans SCADE. Le langage Lustre offre une description déclarative par un ensemble d'équations qui doivent être toujours vérifiées. Son aspect synchrone offre à ces opérateurs une capacité de répondre instantanément à ces entrées. Il fournit un ensemble limité de constructeurs mais très puissant pour décrire toutes les caractéristiques du système étudié et suffisamment restrictif pour répondre aux contraintes de criticité. Dans le contexte de l'aviation, certains compilateurs SCADE ont été prouvés formellement pour vérifier l'absence de dépassement de capacité et l'absence d'accès mémoire interdit. Ceci a été assuré par la description à la fois du programme et de son cahier des charges dans le même langage en adoptant au même temps des sémantiques mathématiques qui permettent l'élaboration de preuves ou la réécriture formelle. Lesar [Ray08] est un outil de vérification de programmes Lustres spécifiées de cette manière.

3.3 Signal

Signal [GLGB87] est un langage de flux de données développé à l'IRISA Rennes par une équipe dirigée par Albert Benveniste et Paul le Guernic. L'industrialisation de Signal a été assurée par la société TNI à travers l'environnement Sildex [1]. Comme Lustre, Signal est un langage déclaratif, dans lequel un programme exprime les relations entre les séquences temporisées de

1. *http* : *//www.tni − world.com/*

valeurs. Les signaux sont définis par des équations qui spécifient leurs propriétés. L'ordre des équations n'est pas pertinent. Les signaux se présentent sous trois formes possibles : présent, absent et bot (non défini). Contrairement à Lustre qui est un langage fonctionnel, Signal est plutôt un langage relationnel : de façon générale, un programme en Signal définit une relation entre les flux d'entrées et de sorties. La façon dont un flux de sortie est utilisé peut limiter le flux d'entrées de l'opérateur qui le produit. Le style de programmation en Signal est alors similaire à la "programmation par contraintes" : Chaque élément d'un programme induit ses propres contraintes limitant ainsi le non-déterminisme du programme. La conjonction de toutes ces contraintes doit aboutir à une description déterministe : ceci est vérifié par le compilateur. Signal est un langage synchrone. Ceci veut dire que tous les évènements, c'est à dire les valeurs de tous les signaux, sont totalement ordonnées. Par conséquent, la façon dont un signal est utilisé peut avoir des conséquences sur le signal lui-même autrement dit sur son horloge. Contrairement à Lustre, il n'y a pas d'horloge globale. Au lieu de cela, Signal manipule des arbres d'horloges [Fri95]. Si deux signaux ont la même horloge, alors toute évaluation de l'un évoque l'évaluation de l'autre. De même, la non-évaluation (bot) de l'un implique la non-évaluation de l'autre. Certes, Signal offre un traitement simple des données mais la programmation d'une simple séquence devient rapidement très complexe. Sigali [MBLG00] est l'outil de vérification de modèle conçu pour prouver les propriétés statiques et dynamiques de programmes en Signal.

3.4 Esterel

Esterel [BGGL92] est un langage synchrone impératif développé principalement dans l'équipe de recherche INRIA dirigée par Gérard Berry en 1980 et également au centre des mathématiques appliquées CMA de l'Ecole des Mines de Paris. Son style impératif permet principalement une expression simple du parallélisme. Étant donné deux programmes parallèles P1 et P2 en Esterel, leur mise en parallèle assure les caractéristiques suivantes :

- Tout les évènements émis par P1 ou P2 sont pris en compte dans le même instant par P1 et P2. En particulier, toutes les sorties générées par P1 (ou P2) sont disponibles dans le même instant de P2 (ou P1).
- L'opération P1 $\|$ P2 termine dans l'instant logique où les deux programmes P1 et P2 se terminent.
- Les données et variables de P1 et P2 ne sont pas partagées.

La mise en en œuvre du parallélisme dans le langage Esterel évite tout indéterminisme du programme. Tous les comportements en Esterel doivent être forcément déterministes. D'ailleurs,

la sémantique du langage garantit ce déterminisme pour tous les opérateurs. Cette hypothèse de déterminisme simplifie considérablement les spécifications de comportement d'un programme en Esterel.

Ce déterminisme est assuré au niveau implémentation par l'hypothèse d'atomicité. C'est à dire une réaction particulière du système n'interfère jamais avec les autres réactions évitant ainsi tout chevauchement possible. En effet, Esterel en tant que modèle possède une vitesse de réaction infiniment rapide en même temps qu'une connaissance exacte de l'environnement. Il permet grâce au parallélisme de gérer plusieurs entrées à la fois. Ceci a servi à définir la notion de modules dans Esterel. Un module présente une unité de comportement, ou plutôt un sous-programme indépendant ayant son propre comportement et des données locales. Par la suite, la construction de systèmes réactifs complexes peut être assurée par une combinaison des différents modules. En fait, un programme en Esterel est typiquement un réseau interconnecté de modules. L'échange entre les différents modules d'un système est assuré par la diffusion de signaux purs. Cette communication est réalisée de la même façon à l'intérieur d'un module. Un signal durant une telle communication est reçu à l'instant déterminé où il est émis assurant ainsi une communication instantanée. Dans ce cas, une réaction peut être considérée comme une propagation des entrées vers les sorties. A chaque instant, un signal peut être présent, donc émis, comme il peut être absent autrement dit non émis. L'absence d'un signal peut être testée explicitement mais son émission est implicite. A la réception d'un signal, différentes réactions du programme peuvent être établies suivant les mécanismes implémentés. Dans ce contexte la préemption forte est définie, elle interrompt instantanément un bloc du programme "P" dès la réception d'un signal "S". La préemption faible est un autre mécanisme qui permet de terminer les réactions en cours d'un bloc du programme "P" à la réception d'un signal "S" avant d'interrompre son exécution. Le mécanisme de suspension "suspend" est aussi un mécanisme important d'Esterel assurant la suspension de l'exécution d'un bloc "P" à chaque fois qu'un signal "S" est reçu.

Contrairement au mécanisme du rendez-vous (hand shaking, en anglais) qui permet une seule communication à la fois d'une entité avec une autre, la diffusion instantanée des signaux en Esterel peut être considérée comme une technique de mains levées (hand raising, en anglais). Si un élément souhaite communiquer avec les autres, il lève sa main pour que tout le monde puisse le voir. Ceci est analogue au principe de la communication radio où tout les récepteurs sont tous sensibles à la même information. Donc, tous les signaux sont diffusés et chaque module peut écouter un signal émis. La diffusion en Esterel apporte une caractéristique importante : il peut y avoir plusieurs émissions et réceptions de signaux en séquence dans le même instant. Cette caractéristique permet de programmer ce qu'on appelle des "diffusions instantanées" qui sont

spécifiques à Esterel. L'émission d'une requête, la réception de la réponse et son interprétation au même instant est un exemple concret de diffusion instantanée.

Dans sa version complète, Esterel intègre le traitement des données : booléens, entiers, flottants et chaînes de caractères. La manipulation des données est gérée globalement par un langage hôte tel que C ou Java suivant la cible recherchée. En outre, Esterel offre un moyen de créer et de manipuler ses propres structures de données par l'intermédiaire d'une interface avec le langage C. Ces données peuvent être manipulées par le biais de simples variables ou bien être transmises par des signaux. Dans ce second cas, un signal est caractérisé à la fois par son statut de présence ou d'absence et par la valeur qu'il transporte. La valeur d'un signal demeure indéfinie ou bien initialisée à une valeur par défaut jusqu'à ce que le signal soit émis pour la première fois. A la différence d'une variable, un signal ne peut pas prendre plusieurs valeurs successives au cours d'un même instant.

Esterel intègre une sémantique mathématique rigoureuse. Il génère du code effectif sous forme d'automates finis ou de système d'équations. Le langage Esterel favorise très bien l'application des tests de compilation et des techniques de vérification. Esterel a connu un succès assez considérable dans plusieurs entreprises et universités. Esterel ainsi que son formalisme graphique SyncCharts développé par Charles André ont été commercialisés par la société Synopsys. Ils ont été utilisés par des sociétés comme Dassault Aviation pour l'avionique, Thomson pour les télécommunications, Texas Instruments pour le développement de circuits.

3.5 StateCharts et Argos

Argos [MR01] développé par Florence Maraninchi, est un langage synchrone impératif qui permet la description de systèmes réactifs sous forme de compositions de machines de Mealy. Argos diffère des StateCharts [Har87] par l'application de l'hypothèse du synchronisme fort. StateCharts développé par David Harel, de son côté présente une large extension du formalisme classique des machines et des diagrammes d'états. Ce langage fournit une flexibilité au niveau de la conception. Il est principalement dédié pour la spécification et la conception de systèmes à évènements discrets complexes.

3.6 SyncCharts

SyncCharts [And96] est un formalisme graphique dédié à la modélisation des systèmes réactifs. Il est fortement inspiré des StateCharts et d'Argos. Le langage SyncCharts s'apparente à la classe des StateCharts par la définiton de la notion des états, des états initiaux et finaux, des transitions, des signaux et des évènements, la hiérarchie, la modularité et le parallélisme. Par

contre, il se diffère de son aîné par l'introduction d'opérateurs synchrones. Il ne permet aucune interprétation ambiguë et aucun comportement caché. En cela, il garantit le déterminisme du modèle. Il intègre des évènements multiples ainsi que la négation des évènements et omet toute transition entre plusieurs niveaux hiérarchiques à la fois.

Comparé à ses prédécesseurs, SyncCharts traite aussi la préemption d'une manière plus rationnelle en introduisant la notion d'avortement et de suspension. Il utilise un ensemble restreint de primitives graphiques assez puissantes. Sa sémantique est fondée sur un calcul de processus permettant la traduction systématique en un programme Esterel, de sorte que l'utilisateur peut profiter de l'environnement logiciel développé pour la programmation synchrone. SyncCharts intègre aussi la simultanéité des évènements ainsi que la diffusion instantanée des signaux durant la communication.

3.7 Quartz

Quartz [Sch09] est un langage de programmation synchrone impératif dérivé d'Esterel dédié à la spécification, la vérification et la mise en œuvre de systèmes réactifs. Il a été développé à l'Université de Kaiserslautern (Allemagne). Quartz est assez similaire à Esterel. En particulier, il ajoute des instructions pour l'exécution parallèle asynchrone des tâches. En plus des opérateurs connus du flux de contrôle (séquences et itérations conditionnelles), Quartz présente la notion de concurrence et de préemption. Quartz est étendu avec l'intégration des affectations de données retardées. La manipulation de ces données permet de décrire convenablement de nombreux algorithmes (séquentiels) et aussi des circuits matériels. De plus, Quartz décrit une sorte de consommation quantitative du temps, où "quantitative" signifie que deux unités logiques de temps sont consommées. Quartz permet également de gérer les données analogiques afin de s'adapter aux systèmes hybrides. Les programmes Quartz peuvent être convertis en relations de transitions symboliques équivalentes dans le but de fournir une base opérationnelle pour la vérification formelle. Le déterminisme de ce langage est également très important pour la simulation, car il permet de reproduire les comportements observés avec plus de précision. Contrairement à Esterel et d'autres langages synchrones, les variables en Quartz sont toujours présentes : Une variable peut prendre une seule valeur à chaque instant.

3.8 Light Esterel

Light Esterel [RGR08] est un nouveau langage de programmation synchrone inspiré en particulier de la syntaxe classique d'Esterel V5. Il a été développé entre les équipes INRIA, LEAT et CMA par Annie Ressouche, Daniel Gaffé et Valerie Roy. Ce langage est déterministe

et dédié comme Esterel à la spécification et la vérification des systèmes de contrôle réactifs. Cependant, il se distingue de son aîné par plusieurs aspects :

1. En plus d'une syntaxe proche d'Esterel, il offre une représentation native des automates hiérarchiques. Par contre, ces automates n'ont pas toute la puissance de description de SyncCharts. Il possède également une représentation directe des machines de Mealy ;
2. Il permet une véritable compilation modulaire et il est capable d'intégrer les instances de modules déjà compilées ;
3. Durant la phase de compilation les signaux comprennent quatre valeurs (présent, absent, final ou erreur) contrairement à deux en Esterel (présent ou absent). L'algèbre utilisée pour manipuler ces signaux [GD13] a été définie à l'origine par Benalp puis reprise par Ginsberg et Mobasher. Par exemple, l'opérateur AND quadri-valué est défini par la table de vérité II.5. Les quatres valeurs sont ensuite codés sur 2 bits (Buttom :00, 0 :01, 1 :10, Erreur :11). Avec ce codage adapté, l'opérateur logique quadri-valué se réécrit sous la forme de deux équations booléennes comme le montre la figure II.5. A la fin de ce processus, les signaux sont re-projeté dans le monde booléen. Cette opération s'appelle la "Finalisation". Un de ces rôles est de gérer la réaction à l'absence d'un signal. Elle est chargée également de générer des codes cibles : C, VHDL, blif [Uni98] ;
4. Etant encore en cours de développement, Light Esterel ne supporte pas la suspension.

⊡	1	0	⊤	⊥
1	1	0	⊤	⊥
0	0	0	0	0
⊤	⊤	0	⊤	0
⊥	⊥	0	0	⊥

Table de vérité

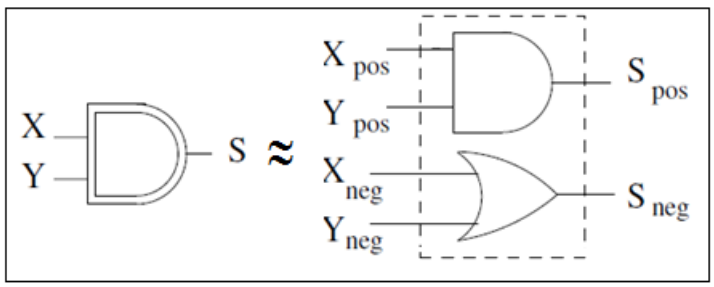


Figure II.5 – L'opérateur AND quadri-valué en Light Esterel.

Nous focalisons notre attention spécifiquement dans ce mémoire sur le langage Light Esterel qui nous paraît être le bon candidat pour utiliser cette approche synchrone dans notre contexte vu les points 1 et 2 ci-dessus. De plus, l'accès aux sources du compilateur a été un avantage indéniable pour générer du code cible plus adapté à nos travaux.

4 Compilation d'Esterel vs Light Esterel

4.1 Compilation Esterel

4.1.1 Compilation en Machine de Mealy explicite

La compilation d'un programme Esterel avec les versions v1, v2 et v3 génère directement un automate d'états finis "FSM" (Finite State Machine, en anglais) qui sera détaillé dans la suite ou plus précisément une machine de Mealy, par expansion puis aplatissement des comportements. Ces machines possèdent un nombre fini d'états pour représenter différents systèmes (logiciels, électroniques, etc). Partant de l'état initial, ces systèmes changent d'un état à un autre suivant les règles de transitions qui les caractérisent, en tenant compte de leur état courant et des stimuli externes. Cette représentation en automates permet d'obtenir une description concise et complètement explicite du graphe de transitions du système. Dans chaque automate de cette représentation, les transitions sont étiquetées par les entrées et les sorties du programme. Une telle représentation offre une exécution extrêmement rapide assurant ainsi une connaissance directe de toutes les configurations. Elle permet également une interprétation ou traduction facile de l'automate explicite en un langage cible tel que ADA, C et Java. Jusqu'à une certaine dimension, les automates explicites sont très lisibles pour un être humain.

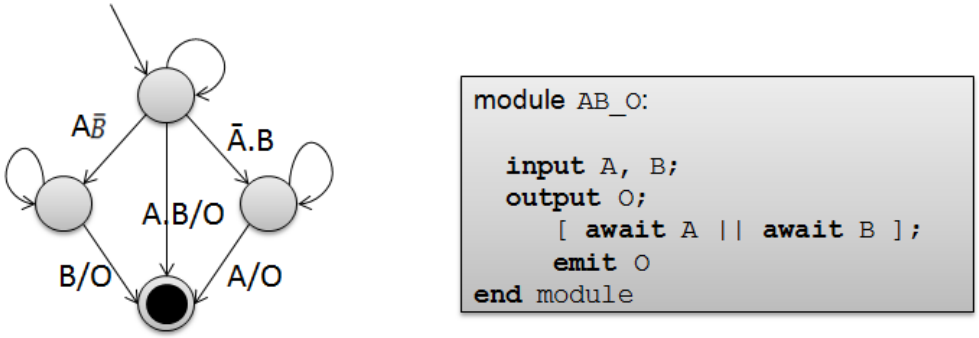


Figure II.6 – Programme ABO.

Néanmoins, une compilation en automates explicites présente deux principaux désavantages : Elle nécessite un espace de stockage et un temps de génération exponentiels en terme de taille du code. Ceci peut être dû en partie au phénomène de parallélisme qui mène a la duplication des sous-modèles en parallèle. Ceci s'illustre clairement à partir du programme ABO en Esterel (version restreinte du "fameux" [2] programme ABRO [Ber99]) représenté dans la figure II.6. Ce programme émet un signal de sortie O à la réception des signaux d'entrées A et B. A

2. Systématiquement mis en avant pour illustrer le langage Esterel

et B peuvent être émis en même temps ou à des instants différents.

Cet automate correspond au produit cartésien des deux sous graphes associés aux instructions "await A" et "await B". Le test de présence de ces deux instructions ainsi que l'émission du signal O sont itérés à plusieurs reprises, tandis qu'il n'est jamais fait plus d'une fois. En rajoutant à cet exemple un troisième signal d'entrée C, l'automate devra attendre en plus la réception du signal C, "await C" avant d'émettre le signal de sortie O, cet automate s'avère clairement plus complexe par l'augmentation du nombre d'états et d'arcs. Ainsi, l'automate aura 2^n états pour un nombre n de signaux d'entrée en parallèle. La complexité d'un tel automate peut mener à des entrelacements de comportements initialement indépendants et définis en parallèle dans un programme, ce qui complique encore l'analyse de cette machine. En effet, ces automates ou plus précisément ces machines de Mealy se présentent par un modèle "plat" non hiérarchique. Cette présentation ne permet pas d'introduire la notion de préemption, de parallélisme et manque de structuration. Afin de remédier à ces problèmes, une nouvelle version "4" du compilateur Esterel a été créée pour passer de la compilation en machine de Mealy explicite à la compilation en circuits logiques implicites.

4.1.2 Compilation en circuits logiques

A partir des versions v4 jusqu'à v7, la complexité de la compilation Esterel est devenue pratiquement linéaire avec la taille du code. Dans sa version 4, le compilateur Esterel génère des équations de changement d'états. Ces équations pouvaient être produites d'une manière facultative à partir des automates explicites. Ceci permet de construire une structure aplatie facile à exploiter par les outils de calculs.

Contrairement aux machines de Mealy, les circuits sont plus faciles à générer et possèdent une taille linéaire par rapport aux programmes d'origine. En effet, les circuits offrent une présentation implicite des états atteignables par un vecteur d'états ce qui réduit nettement l'explosion combinatoire due à une représentation explicite. Ils permettent aussi une utilisation facile du parallélisme sans aucune contrainte. Ceci offre une meilleure lisibilité des informations concernant par exemple les dépendances de données qui sont introduites d'une manière dispersée dans le cas d'une compilation en automates.

Une compilation d'un programme Esterel en circuits repose sur une interprétation dénotationnelle du langage. Elle consiste essentiellement à traduire d'une façon récursive chaque instruction en un circuit logique en se basant sur une sémantique constructive. Cette traduction associe un registre booléen à chaque instruction. En conséquence, un état du programme peut être codé par un nombre bien défini de registres. Il s'agit ensuite de lier ces registres par

rapport aux équations combinatoires du circuit.

Un circuit est dans l'ensemble un réseau de portes logiques appelé "netList". Dans ce contexte, quatre types de portes logiques sont définis : les trois portes ET, OU et NON qui établissent la partie combinatoire du système et les registres booléens qui mémorisent l'état du système. Les entrées et sorties du système sont transmises par des fils portant une valeur booléenne égale à 1 lorsque le signal est présent et égale à 0 lorsque le signal est absent. Ainsi, un système synchrone peut être représenté par une liste triée d'équations qui manipule les variables et les registres booléens en fonction logique d'autres valeurs et qui permet d'assurer le caractère constructif du système.

A l'exécution, une notion d'instant est définie à travers les registres booléens. Un circuit synchrone réagit alors à chaque instant. Au début, tout les registres sont initialisés. Par la suite, l'état du système est calculé et mémorisé à chaque instant. Enfin, la valeur suivante des registres est déterminée à partir des entrées et des valeurs courantes comme l'illustre la figure II.7.

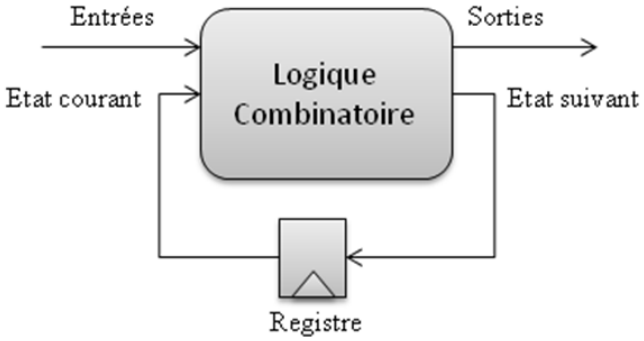


Figure II.7 – Circuit séquentiel synchrone.

La figure II.8 montre la compilation du programme ABO en circuits. Les deux cadres en pointillés représentent les deux instructions "await A" et "await B". Le bloc suivant n'a pas été détaillé dans la figure, il permet la synchronisation entre les différents blocs à chaque instruction. Il permet de gérer en particulier le niveau de l'instruction de préemption (Abort) en dessus. Comme il est construit à la compilation, il rend le programme non modulaire. Ce circuit parait assez complexe par rapport à une représentation en automates. Néanmoins, l'ajout d'un signal en attente, par exemple "await C", ne fait qu'additionner un bloc supplémentaire à ce schéma. Ceci est valide pour toutes les instructions du langage Esterel telle que les préemptions, les exceptions, les boucles. L'impact de l'ajout d'une de ces instructions est linéaire sur le circuit, ceci permet d'avoir autant de blocs que d'instructions tandis qu'il pourrait avoir un impact exponentiel à l'égard d'un automate.

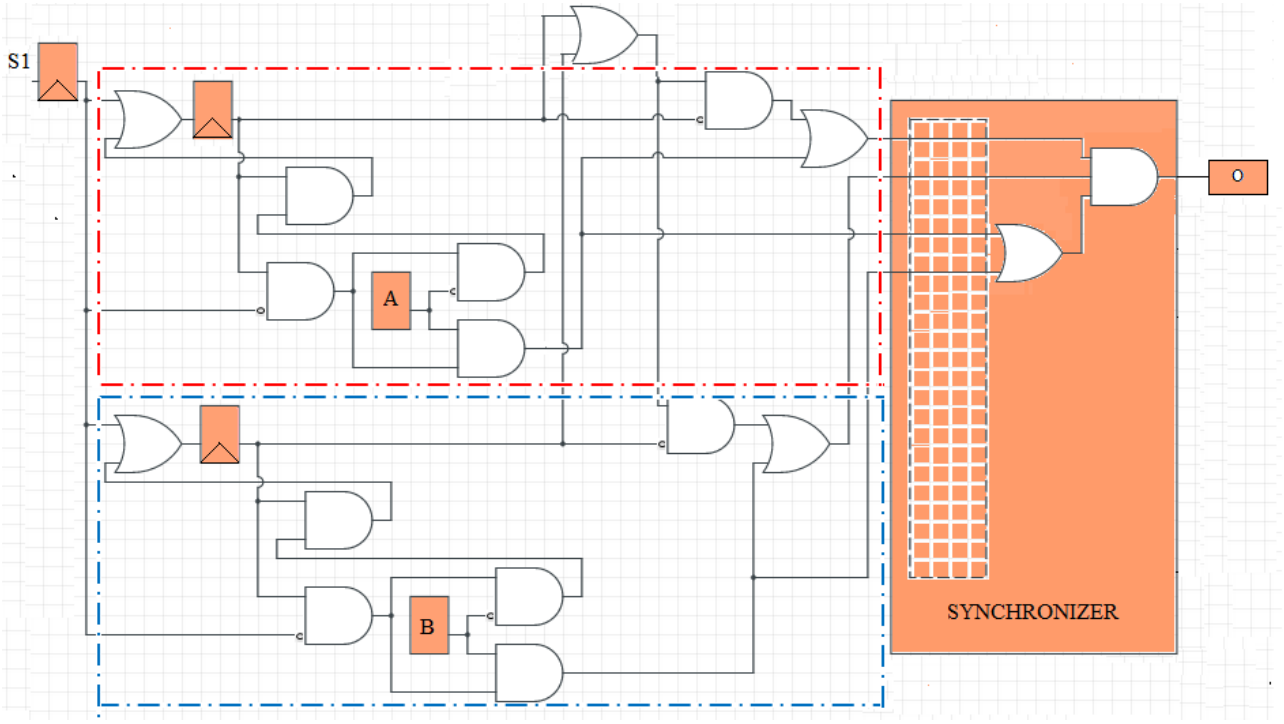


Figure II.8 – Compilation en circuit du programme ABO.

4.2 Compilation Light Esterel

Le langage Light Esterel a été crée principalement dans le but de permettre la compilation modulaire. En effet, l'intérêt majeur de la compilation modulaire est de pouvoir compiler séparément chaque sous système (parallèle et dépendant) et permettre ainsi de modifier ou d'ajouter un module sans recompiler le système complet. Cela permettra également dans l'avenir de réaliser une simulation modulaire. Le compilateur du langage Light Esterel (LE) appelé "CLEM" s'appuie bien entendu sur la sémantique équationnelle du langage LE. Il met en œuvre les principales règles sémantiques afin de compiler chaque programme en un système d'équations correspondant. Le principe de cette compilation est similaire à celui du langage Esterel par la mise en œuvre d'une sémantique constructive basée sur les circuits. Par contre, cette sémantique permet une interprétation en un circuit séquentiel quadri-valué en Light Esterel plutôt qu'un circuit tri-valué dans le cas d'Esterel. Ceci permet un calcul plus simple des valeurs des signaux dans LE car l'algèbre quadri-valué choisie est bien adaptée pour gérer les sur-spécifications tel que le cas où un signal est vrai et faux au même temps. La figure II.9 montre la réécriture de l'instruction "await" en LE.

La figure II.10 montre la réécriture du programme ABO en Light Esterel. Cette réécriture, contrairement à Esterel, garantit une partie synchronisation fixe qui ne dépend pas du nombre

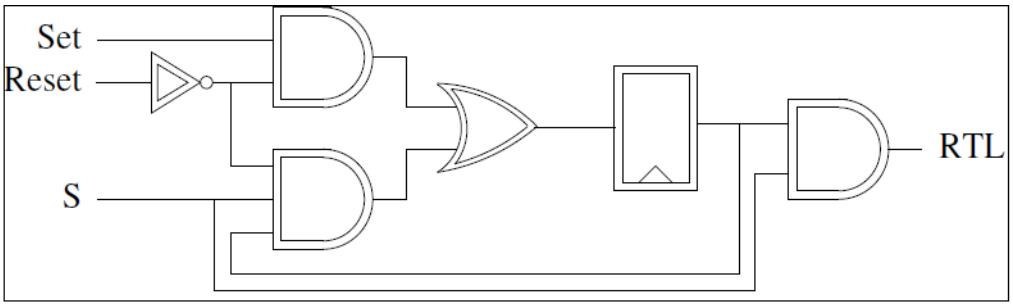


Figure II.9 – Réécriture de l'instruction "await" en Light Esterel.

de préemptions imbriquées. Le prix à payer est par contre l'ajout d'un registre de plus par branche parallèle pour conserver l'état présent. Ce choix est justifié par la volonté de conserver le caractère incrémental de la compilation Light Esterel. A l'encontre de la compilation Esterel, aucune retouche n'est nécessaire par la suite et les différents blocs peuvent être compilés séparément.

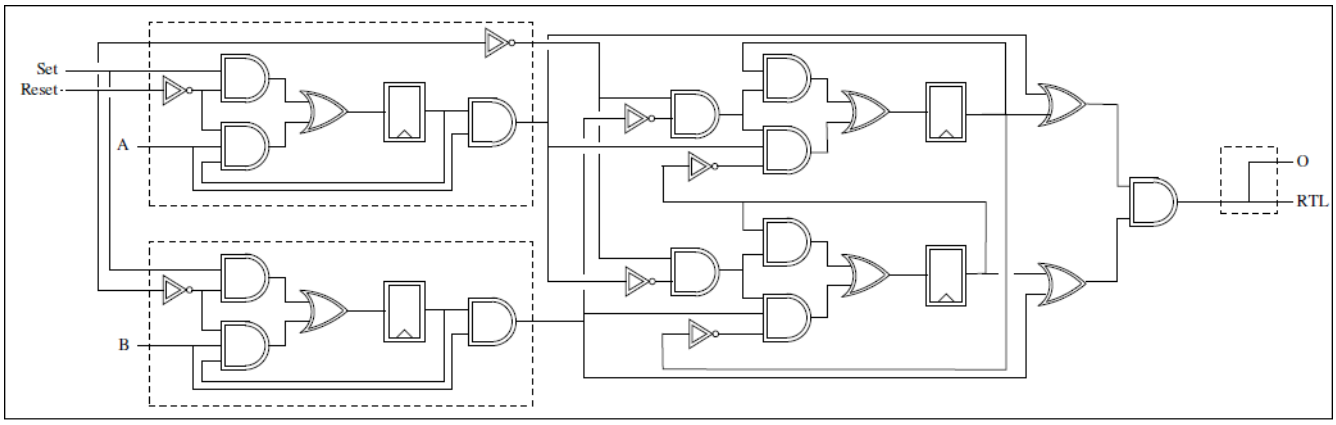


Figure II.10 – Compilation du programme ABO en Light Esterel

Assurer une compilation séparée d'un langage synchrone nécessite absolument l'utilisation d'un ordre d'évaluation valable pour tous les instants. Généralement, cet ordre est statique dans la plupart des langages synchrones populaires, évitant toute compilation séparée. Ce problème a été résolu par la mise au point d'un ordre partiel incrémentiel afin de rompre la liaison entre deux signaux indépendants. A cet effet, deux principales variables entières ont été définies pour chaque équation, à savoir (CanDate , MustDate) pour enregistrer la date de l'évaluation d'une équation. La "CanDate" caractérise la première date à laquelle l'équation peut être évaluée. Tandis que, La "MustDate" caractérise la dernière date à laquelle l'équation doit être évaluée. Ces dates varient dans une échelle de temps discrets appelés niveaux. Le niveau 0 caractérise les premières équations à évaluer puisqu'elles dépendent de variables libres, alors que le niveau

n +1 caractérise les équations qui nécessitent l'évaluation des variables de niveaux inférieurs (de n à 0). Les équations de même niveau sont indépendantes et donc peuvent être évaluées quel que soit l'ordre choisi. En effet, un couple de niveaux est associé à chacune des équations. Cette méthode est inspirée de la méthode PERT [KC66] qui permet de traiter les ordres partiels dans des systèmes d'équations au lieu de s'aligner avec un ordre arbitraire total. La figure II.11 [RG11] montre le graphe de dépendance des variables correspondant aux équations II.1 et II.2 dans lesquelles chaque variable X est reliée à toute variable X' dont elle dépend pour être évaluée. Ce graphe montre les deux dépendances can(X) et must(X) de chaque variable X, où les nombres 0, 1, 2 et 3 présentent les dates logiques. Par exemple, en dépendance MustDate, l'action "t" ne peut se faire qu'à l'instant 2 car elle dépend de l'action "c" qui se fait à l'instant 1. Tandis qu'en dépendance CanDate, les dates sont inversées et "t" peut se faire plus "tard".

$$must(can(a)) = a, t, c \tag{II.1}$$

$$can(must(a)) = a, b \tag{II.2}$$

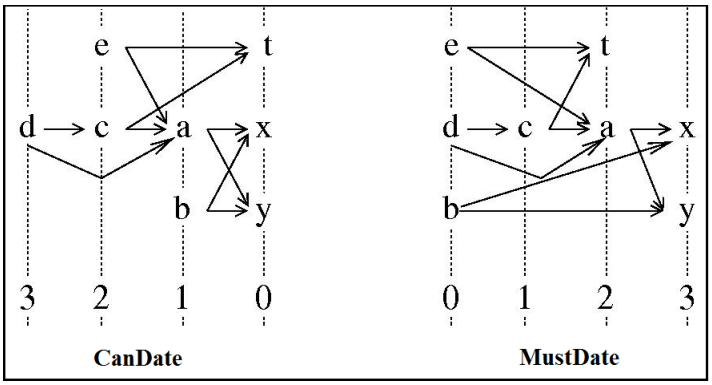


Figure II.11 – CanDate et MustDate

La figure II.12 [RG11] montre la chaine de compilation CLEM. A partir du code en Light Esterel, CLEM génère un format d'enregistrement appelé format "LEC" pour enregistrer le système d'équations booléennes sous forme de BDD généré à partir du programme Light Esterel ainsi que leurs dates d'évaluation des variables (CanDate et MustDate). Dans la pratique, le compilateur CLEM génère le format LEC dans le but de pouvoir réutiliser d'une manière plus efficace le code déjà compilé grâce à la méthode PERT citée précédemment. Dans la phase finale de compilation, le format "LEC" peut être traduit vers différentes cibles logicielles et

matérielles(code C, code Vhdl, synthétiseurs FPGA, Blif). Le format BLIF "Berkeley Logic Interchange [Uni98] est un format bien adapté pour l'application des d'outils de vérification par la suite. C'est un format très compact pour représenter les netLists (citcuits logiques) auxquels CLEM a ajouté des moyens syntaxiques pour pouvoir enregistrer la "CanDate" et "MustDate" de chaque variable.

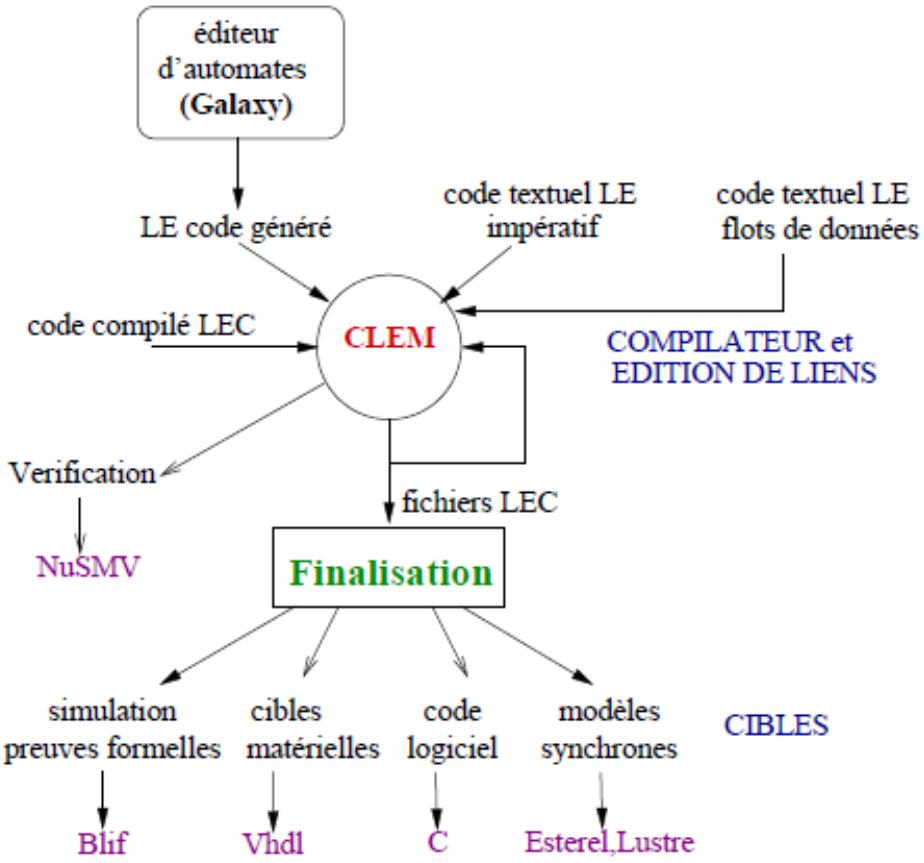


Figure II.12 – Chaîne de compilation Light Esterel .

Pour conclure, la compilation Light Esterel offre deux principaux avantages :

- Elle garantit la modularité en assurant une liaison efficace de deux ordres partiels, sans avoir besoin de s'engager dans un re-calcul de l'ordre partiel global. En effet, il suffit de relier le module principal du système d'équations avec un sous-module déjà compilé (appelé à partir d'une instruction d'exécution), dans ce cas l'opération de liaison est assurée avec le simple ajustement des dates de leurs variables communes.

- Elle offre également la possibilité de vérifier le comportement d'un système étudié. Comme il a été décrit dans le début de ce chapitre, vérifier formellement la correction d'un système consiste à s'assurer que la modélisation de sa structure satisfait la formule décrivant le compor-

tement attendu. Cette formule est généralement écrite en se basant sur une logique temporelle dans la plupart des techniques existantes de vérification de modèle. Dans ce contexte, la sémantique équationnelle du langage Light Esterel "LE" offre un système d'équations qui encode l'ensemble des comportements du programme. Par la suite, une translation est réalisée du programme LE compilé en un format SMV [CCGR00] qui est l'entrée du model checker "NuSMV" [CCGR00] assurant ainsi la vérification des propriétés du langage LE. En effet, il existe peu d'approches qui considèrent la compilation modulaire à cause d'un désaccord profond entre la causalité et la modularité. La causalité signifie que pour chaque évènement généré dans une réaction, il existe une chaîne causale d'évènements amenant à cette génération. La plupart des langages synchrones s'intéressent attentivement à la vérification de la causalité de leurs programmes. Ainsi, s'appuyer sur une sémantique pour compiler un langage assure une approche modulaire, mais nécessite de compléter le processus de compilation avec une vérification globale de causalité. L'originalité de la compilation Light Esterel repose sur la façon de vérifier la causalité à partir des sous-programmes déjà vérifiés et sur l'approche modulaire adaptée.

5 Modélisation des systèmes réactifs

5.1 Machines à états finis

Les machines à états finis (FSMs) [Zah80] appelées aussi automates à états finis sont largement utilisées dans la modélisation des programmes informatiques et des circuits logiques séquentiels, en particulier pour la modélisation des systèmes réactifs. Elles offrent une description précise pour représenter le flux de contrôle (par opposition à la manipulation de données) et se prêtent bien à l'analyse formelle, comme le "model checking". Une machine à états finis est vue comme une machine abstraite dans laquelle il n'existe qu'un seul état actif à la fois. Il existe dans l'ensemble deux types de FSMs : Le premier type concerne des FSMs explicites qui se présentent par un graphe étiqueté dont les sommets correspondent aux états du système et les arcs correspondent aux transitions du système. A partir de l'état initial, la transition d'un état à un autre se fait selon les règles de transitions qui les caractérisent par la présence d'une condition ou d'un évènement déclencheur. Dans le cadre de l'approche synchrone, un deuxième type, celui des FSMs implicites est apparu, les machines à états finis implicites permettent de présenter les systèmes réactifs programmés à partir des différents langages synchrones. L'état initial présente le premier état activé. L'état interne est encodé dans des registres ou vecteur de variables d'états. Les stimuli et les réactions du système à ces stimuli externes sont fournis par des signaux d'entrées/sorties.

Dans la pratique, il existe deux principales méthodes pour générer les sorties d'une machine implicite à états finis : les machines de Moore et les machines de Mealy. Les sorties dans une machine de Mealy sont générées à partir de l'état existant et des entrées données comme l'illustre l'équation II.3. En revanche, les sorties dans une machine de Moore présentée par l'équation II.4 ne sont générées qu'à partir de l'état existant. Une machine de Mealy ajoute aux machines d'états finis la notion d'actions à effectuer lors des transitions d'états. Contrairement à une machine de Moore qui est toujours synchrone, une machine de Mealy peut être asynchrone a cause de la dépendance de ses sorties par rapport à ses entrées.

$$Prochainetat = F(etatpresent, entrees); Sortie = G(etatpresent, entrees) \tag{II.3}$$

$$Prochainetat = F(etatpresent, entrees); Sortie = G(etatpresent) \tag{II.4}$$

Une autre distinction peut être réalisée au niveau déterminisme. En effet, dans un automate déterministe, chaque état a exactement une seule transition possible pour chaque entrée donnée. En d'autres termes, une transition doit être clairement prédictible à partir de l'état courant et de l'entrée donnée. Par ailleurs, dans un automate non-déterministe, une entrée peut conduire à une, plus qu'une ou aucune transition pour un état donné. Dans ce cas, une transition de cet état ne peut pas être prévisible. Dans la suite de ce mémoire, nous ne considérerons que des automates déterministes pour représenter le comportement de nos systèmes.

L'utilisation des FSMs au niveau modélisation apporte plusieurs avantages à savoir :

- En raison de leur simplicité, les FSMs sont d'une très grande aide pour la conception des systèmes, faciles à implémenter et rapides en exécution. Ceci permet aux développeurs de les implémenter aisément ;
- En raison de leur prévisibilité (dans le cas des FSMs déterministes), les transitions sont facilement prédictibles permettant ainsi une application adéquate pour le test ;
- Les FSMs sont relativement souples. Ceci autorise plusieurs façons de les implémenter ainsi qu'une transformation directe de la représentation abstraite à l'implémentation du code ;
- Les FSMs sont bien adaptées aux systèmes où le temps d'exécution est commun entre les différents modules ou les sous-systèmes. Seul le code de l'état courant est à exécuter dans ce cas ;
- La représentation des FSMs sous une forme abstraite permet de déterminer facilement l'accessibilité de l'état donné ainsi que les conditions nécessaires pour l'atteindre.

Cependant, ces automates présentent quelques inconvénients :

- Les FSMs dans le cas explicite, sont limitées en mémoire et difficiles à gérer et à maintenir

pour les systèmes qui possèdent un grand nombre d'états ;
- Les FSMs ne sont utiles que dans le cas où le comportement du système peut être décomposé en états séparés avec une définition précise des conditions de transitions. Cela signifie que tous les états, les transitions et les conditions doivent être bien définis et connus à l'avance.

Analyse des FSMs : Exploration (explicite/implicite) de l'espace d'états

Une manière classique de tester les programmes synchrones consiste à l'exploration de l'espace des états accessibles de l'automate, c'est à dire générer par un parcours exhaustif de l'espace d'états du système toutes les séquences possibles. Cette exploration permet une vérification de l'automate, c'est à dire prouver qu'une propriété donnée est toujours vraie sur un ou plusieurs chemins parcourus. La propriété à prouver peut être présentée par une formule logique temporelle ou par une machine à états finis. Le résultat de l'exploration renvoie généralement un contre-exemple sous forme de trace si la propriété est fausse. Nous nous intéressons particulièrement dans cette thèse au premier cas de génération exhaustive de jeux de tests possibles des systèmes plutôt qu'à la vérification des propriétés.

A leur apparition, les méthodes d'exploration d'espaces d'états atteignables s'appuyaient sur une énumération explicite de tous les états du système. Cette technique explicite assure une exploration approfondie de l'espace d'états mais elle souffre nettement d'une explosion en temps face à une combinatoire très importante d'expressions. En effet, l'exploration explicite présente l'espace d'états par des arbres, des tables de hachages ou des graphes. Ainsi, la mémoire nécessaire pour stocker l'espace d'états d'un système augmente rapidement avec le nombre d'états définis pour le système. De plus, consommer un bit pour chaque variable d'état pendant l'énumération explicite des états atteignables est une technique très coûteuse en mémoire.

A ce propos, une deuxième approche a été introduite au début des années 90 réalisant une réduction considérable des besoins en mémoire produits par les techniques explicites. Cette approche repose sur une description implicite de l'ensemble des états plutôt que sur une énumération explicite. Dans ce cadre, l'exécution symbolique [BEL75] présente une méthode très populaire d'exploration d'espace d'états implicites . Elle assure l'automatisation des tests par la production automatique des cas de test possibles dans le but d'atteindre le maximum de couverture des exécutions du programme. En effet, l'exécution symbolique est une technique d'analyse de programmes avec des propriétés d'invariance spécifiques aux entrées plutôt qu'avec des entrées concrètes. Elle présente les variables du programme par des expressions symboliques. En conséquence, les sorties calculées sont exprimées en tant que fonctions des invariants don-

nés. Elle attache une condition à chaque chemin et détermine à la fin les contraintes sur les entrées qui aboutissent à un tel point du programme. Ces contraintes seront résolues par la suite à l'aide d'un solveur générant ainsi les cas de tests possibles du programme. Par ailleurs, l'exécution symbolique peut être utilisée pour identifier les bugs du système : elle permet de vérifier les erreurs d'exécution ou de violations d'assertions en identifiant les entrées de test qui déclenchent ces erreurs. Dans ce contexte, de nouveaux algorithmes tels que les solveurs SMT (Satisfability Modulo Theories) ont été appliquées sur des machines assez puissantes pour la résolution des bugs de systèmes critiques.

Les travaux dans [BEL75],[Cla76] présentent les premières approches d'exécution symbolique. Ils adressent les programmes séquentiels simples avec un nombre fixe de données d'entrée de type primitifs. Les approches plus récentes, telles que l'exécution symbolique généralisée "GSE" [KPV03] et "jCUTE" [SA06], traitent les programmes multithread ayant comme entrées des structures de données complexes et récursives. Par la suite, une extension de l'exécution symbolique pour les grands systèmes a été introduite suite aux progrès récents dans la génération dynamique des tests [CE05],[GKS05]. Une technique populaire de l'exécution de la génération symbolique est d'utiliser les diagrammes de décision binaires BDDs (Binary Decision Diagram, en anglais) que nous détaillerons dans la section III. 2.1. D'une manière globale, les BDDs sont des structures de données pour la représentation des fonctions booléennes. La représentation en BDD de l'espace d'états étudié est en fait calculée en appliquant d'une manière itérative la fonction BDD d'état suivant sur le BDD représentant les états initiaux, jusqu'à atteindre un point fixe. Contrairement à la génération explicite dont les exigences en mémoire augmentent linéairement avec le nombre d'états explorés, la génération symbolique agrandit l'espace d'états accessibles durant la phase de stockage mais le réduit pendant l'exécution.

Néanmoins, l'exécution symbolique souffre encore des problèmes de passage à l'échelle en raison de l'augmentation exponentielle du nombre de chemins à analyser et la complexité des contraintes générées pour les systèmes de grande dimension. D'ailleurs, le nombre de chemins à exécuter peut être infini dans le cas de programmes avec des itérations de boucles. D'autres difficultés se manifestent également en cas de parallélisme. La synchronisation entre les blocs de deux programmes parallèles P1 et P2 par des signaux internes fait que l'ensemble des états atteignables d'un programme parallèle P1 $\|$ P2 n'est plus le simple produit cartésien des comportements des deux programmes P1 et P2. Les solutions proposées à ce problème d'explosion de chemins utilisent généralement soit des heuristiques pour assurer le maximum de couverture dans l'exploration des chemins [MPFH11], soit la mise en parallèle des chemins indépendants [SP10].

5.2 Machines à états finis hiérarchiques

Avec l'apparition de StateCharts, les FSMs deviennent hiérarchiques et leur pouvoir de modélisation augmente considérablement pour décrire les systèmes complexes. Une machine hiérarchique est une extension de la classe des FSMs souvent appelées FSMs imbriquées. Cette classe d'automates est fortement employée dans le contexte de génération de tests et d'analyse formelle. Par exemple, l'outil commercial TestMaster [3] a été basé sur un modèle hiérarchique d'automates pour le test d'un grand commutateur de l'entreprise Teradyne. Les sommets d'une machine hiérarchique peuvent être ordinaires ou des macro-états qui sont eux-mêmes des FSMs. D'une manière plus détaillée, une machine à états finis hiérarchique abrégée HSM est définie comme suit : une HSM de niveau i>j est tout simplement une FSM ordinaire dont certains états sont des macro-états représentant à leur tour des HSMs de niveau inférieur j. L'activation d'un macro-état entraîne automatiquement l'activation de la HSM associée. De même, la sortie d'un macro-état entraîne la préemption de la HSM sous-jacente. StateCharts autorise même de quitter un macro-état à partir d'un état interne d'un HSM interne.

La notion de machines hiérarchiques a été intégrée dans différentes méthodologies de développement des logiciels orientés objet tels que ROOM [RL03] et le langage de modélisation unifié UML (Unified Modeling Language, en anglais). La hiérarchie apporte deux principaux avantages par rapport aux FSMs ordinaires. Tout d'abord, les macro-états stipulent un mécanisme pratique pour la structuration des systèmes par l'application de raffinements par étapes. Ceci offre une visualisation du système à différents niveaux de granularité. Ensuite, en permettant le partage de tout élément de la machine hiérarchique, un composant peut être spécifié une seule fois et réutilisé plusieurs fois par la suite dans différents contextes. Cette caractéristique assure une représentation concise du modèle étudié et apporte la notion de modularité.

5.2.1 Analyse des HSMs par aplatissement

La façon la plus simple d'analyser une machine à états finis hiérarchique est de l'aplatir afin d'obtenir une FSM ordinaire. Le principe de l'aplatissement consiste a substituer d'une manière récursive les macro-états de niveau supérieur par les machines de plus bas niveau. Il s'agit ensuite d'appliquer les méthodes d'analyse citées précédemment (exploration d'espace d'états) d'une FSM simple. En général, la machine aplatie peut être exponentiellement plus large que la machine hiérarchique d'origine. A savoir, sa taille est égale à n^k, n étant le nombre d'états dans un module de niveau i (sous machine) et k le nombre de niveaux dans la machine globale. Cette approche est raisonnable si la taille de la FSM étendu n'est pas trop grande.

3. http ://testmaster.sourceforge.net/

5.2.2 Analyse des HSMs sans aplatissement

Malgré la simplicité éprouvée de l'approche de l'aplatissement pour la vérification et le test, l'article [SAH+00] montre néanmoins la complexité et les limites de cette technique par rapport à la profondeur et la taille d'une machine hiérarchique. En effet, le nombre des composants concurrents dans un automate augmente clairement suite à l'aplatissement. De même, tous les macro-états englobant ont besoin de conserver toutes les transitions associées dans les machines nouvellement générées. En conséquence, le haut degré de dépendance entre ces composants peut dégrader nettement les performances de la technique d'exploration d'espace d'états, en particulier la technique de composition en arrière "Backward".

Pour remédier à ce problème, J.Staunstrup [SAH+00] a étendu la technique de composition en arrière pour exploiter la structure hiérarchique d'un automate sans avoir besoin de l'aplatir. Le principe est de réutiliser le résultat de parcours d'accessibilité des macro-états pour déterminer l'accessibilité des sous-états correspondants. Partant de ce principe, cette approche devient insensible à la profondeur de la hiérarchie. Sa performance s'améliore même avec l'augmentation de sa profondeur. Particulièrement, l'implémentation de cette technique de composition en arrière a permis le test de plus gros systèmes dans une version récente du logiciel industriel de conception "Visual State". Bien que cette technique de composition en arrière ait fait preuve de réussite à cet égard, un problème a été rencontré avec le modèle d'une machine à états d'une caméra zoom par exemple. La vérification de la totalité de ce modèle était quasiment impossible pourtant le système ne comportait que 36 machines d'états dans l'ensemble. Ceci souligne le fait que compter simplement le nombre de machines dans un modèle ne fournit pas une véritable mesure de la complexité d'une tâche de vérification. Des recherches supplémentaires sont donc nécessaires pour une prédiction plus précise de la difficulté de vérifier une conception donnée avec la technique proposée.

L'article [AY01] montre également que la vérification des automates hiérarchiques peut être accomplie sans avoir besoin de produire au préalable le FSM équivalent aplati. R. Alur [AY01] montre dans cet article qu'une machine ordinaire offre une concision exponentielle à un coût exponentiel, alors qu'une machine hiérarchique peut offrir une concision exponentielle à un coût minimal. Il distingue à cet égard deux types de vérification : une vérification locale et une vérification globale d'un modèle donné. Dans la vérification locale, il s'agit de vérifier si quelques (ou tous les) chemins qui commencent par un état spécifié (par exemple, un état initial) satisfont une propriété en un temps linéaire. Tandis qu'en vérification globale, il s'agit de calculer tous les états accessibles de telle sorte que certains (ou tous les) chemins partant de cet état satisfont une propriété en un temps linéaire.

En ce qui concerne les FSMs ordinaires, les deux approches de vérification locale et globale peuvent être résolues en un temps linéaire proportionnel à la taille du système étudié. De l'autre côté, pour les FSMs hiérarchiques, une variante locale peut être résolue d'une manière plus efficace par l'algorithme proposé évitant ainsi l'analyse répétée d'une sous-structure partagée. La variante globale est par contre plus coûteuse : elle nécessite la division d'une sous-structure du fait que la satisfaction de formules temporelles peut varier d'un contexte à l'autre. Nous ne nous intéressons dans notre contexte qu'à la vérification globale.

6 Conclusion

Nous avons présenté dans ce chapitre une comparaison des différents langages de l'approche synchrone afin de trouver le bon candidat à adopter dans notre approche. Nous avons montré également les techniques classiques d'exploration des machines à états finis qui se basent sur le parcours de tous les états. Ces techniques ainsi que les techniques existantes d'analyse des machines hiérarchiques souffrent clairement du problème d'explosion de l'espace des états face aux gros systèmes. En outre, elles sont généralement limitées à l'analyse des systèmes à entrées et sorties booléennes tandis que les systèmes dans la réalité sont à entrées et sorties numériques.

Ainsi, nous visons dans le chapitre suivant à trouver une façon compacte pour coder les valeurs des fonctions numériques en gardant l'avantage de la représentation des variables booléennes en BDDs [Bry86]. Nous présentons également une comparaison des outils existants de tests automatiques.

Chapitre III

Représentation des données des systèmes réactifs

1 Introduction

Comme cité dans le chapitre précédent, la plupart des algorithmes de tests, de génération de circuits et de vérification de modèles (exploration d'espace d'états, exécution symbolique, etc.) repose sur une représentation efficace des structures de données et de leurs formules propositionnelles. Depuis 1986, les Diagrammes de Décision Binaires BDDs ont émergé comme un choix incontournable pour représenter des fonctions booléennes. L'idée de base des BDDs a été étendue par la suite donnant naissance à de nouvelles structures pour la représentation de différents types de données (booléennes ou numériques) tels que les Diagrammes de Décisions binaires à Terminaux Multiples MTBDD, les Diagrammes de Décisions des Différences DDD, les Diagrammes de Décision Linéaires LDD, etc.

Nous étudions dans ce chapitre les structures les plus pertinentes autour des BDDs afin d'aboutir à une représentation efficace des données numériques. Nous établissons à la fin de ce chapitre une étude comparative des différents outils existants de génération de jeux de tests automatiques y compris ceux qui assurent ou pas la manipulation de données numériques.

2 Travaux autour des BDDs

2.1 BDD

Les diagrammes de décision binaire BDDs ont été introduits à l'origine par Lee [C.Y59] et Akers [Ake78], étudiés et enrichis plus tard par Bryant [Bry86]. Ils permettent de présenter chaque fonction booléenne par un graphe orienté acyclique (DAG : Directed Acyclic Graphs). La figure III.1.b montre une représentation en arbre de décision binaire, aussi appelée arbre de Shannon de la fonction $F1 = (a + b) \times c$ conforme à la table de vérité dans la figure III.1.a

où × et + représentent respectivement les fonctions booléennes AND et OR. Le degré sortant des nœuds est égal au maximum à deux. Les nœuds a, b et c présentent des nœuds de décisions.

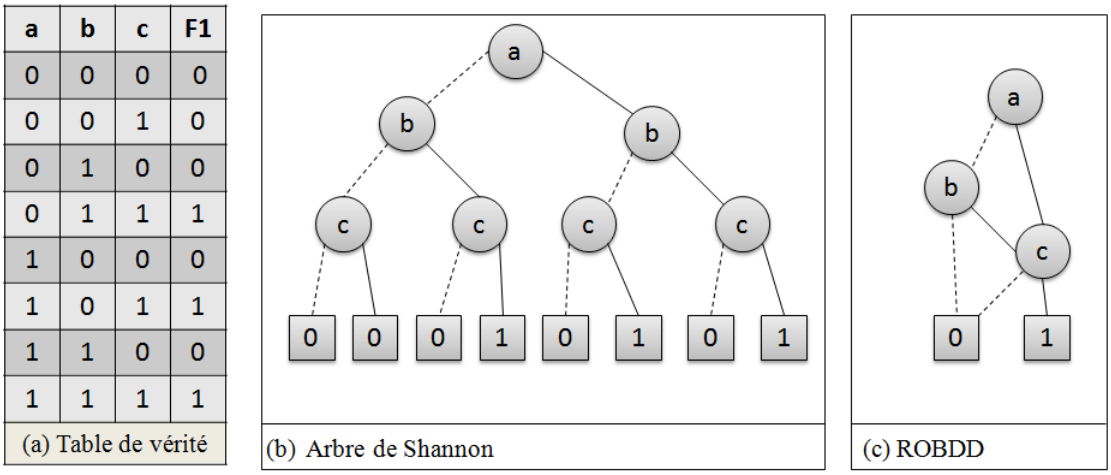


Figure III.1 – Présentation de $F1$ en BDD

Chaque nœud de décision est étiqueté par une variable booléenne et comprend deux nœuds fils : un nœud fils droit correspondant à l'affectation de la variable booléenne à 1 (arête en trait continu) et un nœud fils gauche correspondant à l'affectation de la variable booléenne à 0 (arête en pointillés). Les nœuds terminaux encadrés et marqués par 0 ou 1 correspondent aux valeurs possibles de la fonction $F1$. Pour n'importe quelle attribution des variables a, b et c, la valeur de la fonction $F1$ est déterminée en traçant le chemin de la racine 'a' au nœud terminal suivant la branche appropriée à partir de chaque nœud.

L'idée de base des BDDs dérive à l'origine de la décomposition de Shannon III.1.

$$f(x) = \bar{x}.f_{\bar{x}} + x.f_x \tag{III.1}$$

Une fonction de commutation f peut toujours se diviser en deux sous-fonctions (cofacteurs) f_x et $f_{\bar{x}}$ suivant la structure ITE (If-Then-Else) [1]. En d'autres termes, cette décomposition divise la fonction f en deux cas distincts selon l'évaluation de la variable x à 1 ou à 0. Dans un BDD, un nœud et ses descendants représentent chacun une fonction booléenne f dans laquelle chaque nœud x porte deux arêtes sortantes : une dirigée vers la sous fonction f_x et l'autre vers $f_{\bar{x}}$. Suivre le chemin de la racine à un nœud terminal revient tout simplement à considérer les

1. ITE(a,b,c)=(a∧b)∨($\bar{a}$∧c)

cofacteurs successifs de la fonction f jusqu'à ce qu'elle se réduise à une valeur constante.

L'arbre de décision de la figure III.1.b présente un arbre de Shannon ordonné étant donnée que ses variables restent toujours dans le même ordre a < b < c tout au long de tous les chemins du graphe. Ce graphe peut être transformé ensuite en un diagramme de décision binaire ordonnée et réduit ROBDD en supprimant tout nœud ayant deux fils identiques et en fusionnant tout les sous-graphes isomorphes. Le BDD résultant est représenté par la figure III.1.c. Dans toute la littérature, un BDD se réfère généralement à un ROBDD. Ce type de diagramme fournit plusieurs propriétés intéressantes. Il offre une représentation compacte des expressions booléennes. Il est canonique assurant une représentation unique pour toute fonction booléenne. Cela signifie en particulier, qu'il existe un unique ROBDD pour la fonction constante "vrai" correspondant au nœud terminal 1 et la fonction constante "faux" correspondant au nœud terminal 0. Par conséquence, il est possible de tester en un temps constant si un ROBDD est constamment vrai ou faux.

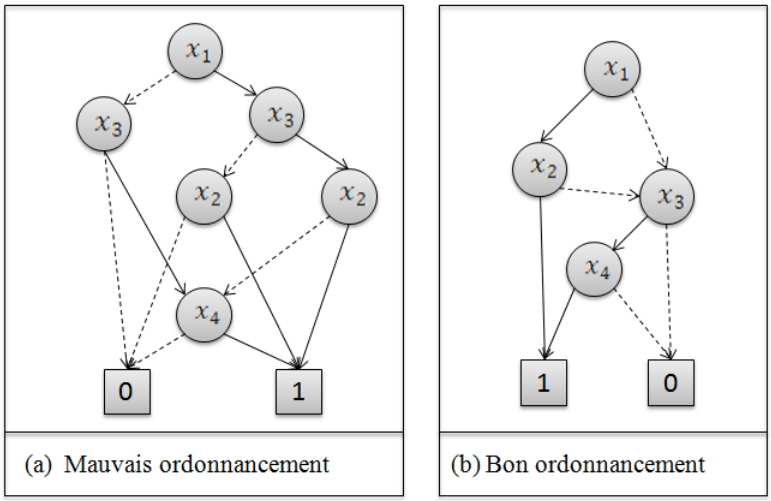


Figure III.2 – Présentation en BDD de $f(x_1, .., x_4) = x_1x_2 + x_3x_4$

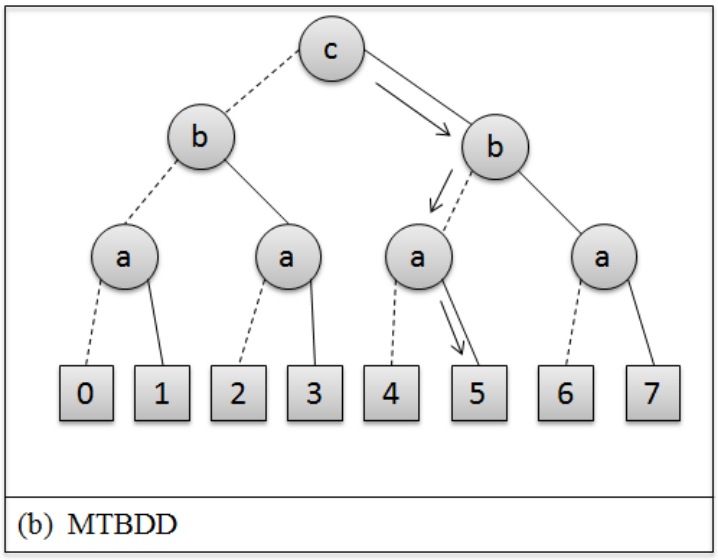
La taille d'un BDD se détermine par la fonction représentée et le type choisi d'ordonnancement de ses variables. Étant donné une fonction booléenne $f(x_1, .., x_n)$, la taille de son graphe est linéaire par rapport à n dans le meilleur des cas et exponentielle dans le pire des cas. Considérons par exemple la fonction $f(x_1, .., x_2n) = x_1x_2+x_3x_4+...+x_{2n-1}x_{2n}$. La figure III.2.a montre qu'en appliquant l'ordre suivant des variables $x_1 < x_3 < .. < x_{2n-1} < x_2 < x_4 < ... < x_{2n}$, le BDD correspondant détermine 2^{n+1} nœuds pour représenter la fonction f. Par ailleurs, en appliquant l'ordre $x_1 < x_2 < x_3 < x_4 < .. < x_{2n-1} < x_{2n}$, le BDD correspondant, d'après la figure III.2.b, détermine simplement 2n+2 nœuds pour la représentation de la fonction f.

2.2 MTBDD

Les diagrammes de décisions binaires à terminaux multiples MTBDD appelés aussi diagrammes de décisions arithmétiques ADD ont été développés par E.Clarke [FMY97] pour la manipulation des fonctions à valeurs numériques. Ils présentent leurs graphes de décision par un BDD, mais qui attribue des valeurs arbitraires sur les nœuds terminaux.

c	b	a	F2
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

(a) Table de vérité



(b) MTBDD

Figure III.3 – Présentation en MTBDD de $F2$

La figure III.3.b montre une représentation en MTBDD de la fonction $F2 = a + 2b + 4c$ conforme à la table de vérité dans la figure III.3.a, à savoir, le résultat de l'interprétation de trois bits comme des nombres binaires non signés.

L'évaluation d'un MTBDD pour une affectation de variables données est quasiment analogue à l'évaluation dans un BDD : il suffit de suivre un chemin unique de la racine au nœud terminal pour retourner à la fin la valeur de la fonction indiquée dans le nœud terminal. En considérant par exemple le chemin indiqué par les flèches dans la figure III.3.b, ceci correspondant à l'affectation $(c = 1; b = 0; a = 1)$, on obtient bien le résultat $F2 = 5$.

Une telle représentation (par les MTBDDs) a fait preuve d'inefficacité vis à vis des fonctions avec une large plage de valeurs. Pour le cas des nombres binaires non signés de longueur n, il existe 2^n valeurs possibles. Une représentation en MTBDD aura par conséquence un nombre exponentiel de nœuds terminaux. Les MTBDDs ne restent donc valables et assez simples que pour certaines applications où le nombre de valeurs possibles est suffisamment restreint.

2.3 EVBDD

Dans le but de supporter un nombre plus large de valeurs, Yung-Te Lai a développé les diagrammes de décision binaires à arcs évalués EVBDDs [LS92] comme alternative aux MTBDDs pour aboutir à une forme plus compacte. Le principe de cette représentation est d'allouer des poids numériques sur les arcs afin de permettre un plus grand partage des sous-graphes. La figure III.4 montre la représentation de la fonction précédente $F2 = a + 2b + 4c$ en un EVBDD. Ce diagramme dessine le poids de chaque arc dans une boite. Par défaut, un arc sans boite a un poids de 0. L'évaluation de cette représentation correspond à tracer chaque chemin en faisant la somme des poids des arcs et de la valeur du nœud terminal. Ceci améne à faire la somme $F2 = 4 + 1 = 5$ relative à l'évaluation (c=1 ;b=0 ;a=1) du chemin fléché de la figure III.4. Cette représentation croît linéairement avec le nombre de bits, ce qui apporte une amélioration majeure par rapport aux MTBDDs. Cette amélioration est manifeste en termes de nombre de nœuds non terminaux mais par contre elle peut présenter un coût assez élevé par nœud.

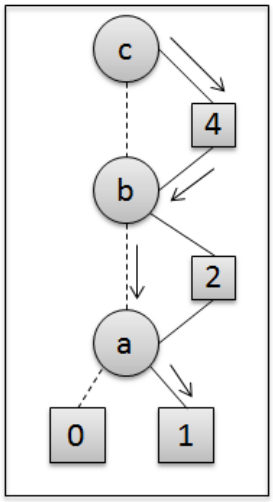


Figure III.4 – Présentation en EVBDD de $F2$

En dépit de cette nette amélioration dans plusieurs cas, les EVBDDs donnent une complexité inacceptable dans certaines classes de fonctions. Un cas classique d'application de fonctions à valeurs numériques est de vérifier formellement des circuits arithmétiques qui calculent diverses fonctions tel que l'addition, la multiplication et la division. Nous désirons dans cette classe d'applications exprimer les valeurs numériques encodées par les opérations en termes de données. Considérons à titre d'exemple deux entiers à n-bits non signés représentés par les vecteurs de bits $\vec{x} = x_{n-1}, .., x_0$ et $\vec{y} = y_{n-1}, .., y_0$. Ces deux derniers peuvent être représentés également par les valeurs numériques encodées $X = \sum_{i=0}^{n-1} 2^i x_i$ et $Y = \sum_{i=0}^{n-1} 2^i y_i$. Comme décrit précé-

demment, EVBDD permet de représenter ces deux codages de fonctions pour X et Y avec une complexité linéaire par rapport à n : Le résultat de l'application d'une addition $X + Y$ ou de soustraction $X - Y$ reste toujours de taille linéaire.

Néanmoins, la représentation en EVBDD de l'opération de multiplication $X \times Y$ ou d'exponentiation 2^X est d'une complexité exponentielle par rapport à n. Dans ce type d'application, nous pouvons conclure que les EVBDD se limitent à des unités de calcul relativement simples tels que les additionneurs, les ALUs (sans multiplication et division intégrées), et les comparateurs.

2.4 BMD

Un Diagramme de Moment Binaire BMD [BC01] est une autre alternative pour manipuler des fonctions numériques. L'idée globale de cette représentation est de changer la fonction de décomposition d'une manière analogue aux FDDs [UKR92] (Functional Decision Diagram). En effet, pour exprimer des fonctions à valeurs numériques, l'expansion de Shannon III.1 devient :

$$f(x) = (1 - x).f_{\bar{x}} + x.f_x \tag{III.2}$$

Cette expansion proposée par MTBDD et EVBDD repose sur l'hypothèse que la variable x est booléenne. Réorganiser les termes de cette équation III.2 amène à l'expression III.3 de décomposition de Shannon dans le cas d'un BMD où $f\partial x = (f_x - f_{\bar{x}})$ représente le moment linéaire de f par rapport à x ; x étant manipulée en tant que variable entière binaire $(0, 1)$; et le complément de x est représenté par $(1 - x)$.

$$f(x) = f_{\bar{x}} + x.(f_x - f_{\bar{x}}) = f_{\bar{x}} + x.f\partial x \tag{III.3}$$

Cette terminologie admet f comme une fonction linéaire par rapport à ses variables, $f\partial x$ est donc la dérivée partielle de f par rapport à x. Le cofacteur négatif est appelé le moment constant désignant la partie de la fonction f qui reste toujours constante par rapport à x. La figure III.5 montre la représentation de la fonction précédente $F2 = x_0 + 2x_1 + 4x_2$ en un BMD. Les deux arrêtes sortantes de chaque nœud désignent les moments constants (pointillés) et les moments linéaires (solides) de la fonction $F2$ par rapport à chaque variable. En d'autres termes, $F2 = const + var \times linmoment = ((0 + x_0 \times 1) + x_1 \times 2) + x_2 \times 4$.

Il est à noter dans cet exemple que le moment linéaire de la fonction $F2$ par rapport à chaque variable x_i est tout simplement égal à 2^i : la fonction se décompose selon les poids des bits. Cette propriété rend les BMDs particuliérement bien appropriés pour la représentation des

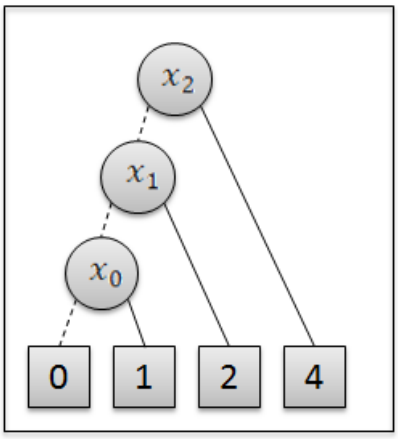


Figure III.5 – Présentation en BMD de F2

fonctions arithmétiques. Cette représentation a été étendue par la suite pour inclure les poids sur les arêtes dans le but de permettre le partage des sous-expressions communes, ce qui a donné naissance au diagramme de moment binaire multiplicatif noté (*BMDs) [Ard]. Ces poids sur les arrêtes se combinent multiplicativement dans un (*BMD), contrairement au principe d'addition dans un EVBDD. A cet effet, l'ensemble de ces fonctions arithmétiques $X + Y$, $X - Y$, $X \times Y$,2^X montrent des représentations de taille linéaire.

Malgré le succés considérable des BMDs et *BMDs canoniques pour la représentation des fonctions arithmétiques, la manipulation des poids sur les arêtes était d'un coût assez élevé. A partir d'une certaine taille (par exemple, 2^{512}états pour la vérification d'un multiplicateur de 256 bits), le code doit utiliser une représentation numérique de précision non bornée. De plus, l'application de ces représentations de moment binaire dans la pratique a rencontré des cas possibles de complexité exponentielle nécessitant de développer certaines techniques pour les contourner. En outre, contrairement au BDDs, les BMDs ne permettent pas de vérifier la propriété de satisfiabilité, et les sorties des fonctions sont des entiers non divisibles en bits séparés. Ceci peut causer un problème dans le cas des applications nécessitant une analyse par bit de sortie.

Suite aux BMDs, les diagrammes de décision hybrides HDDs ont été introduits par Clarke et Zhao [CZ95] représentant une logique plus efficace qui combine les deux représentations de MTBDDs et BMDs. Néanmoins, l'ensemble de ces trois techniques demeurent restreint à la vérification de circuits arithmétiques matériels et non adaptables à la vérification de spécifications de systèmes logiciels. Une autre limite réside dans le fait que la représentation de la fonction x_0x_1 de façon concise est abordable mais, la représentation de l'égalité $x_0x_1 = x_2$ passe à une taille exponentielle de diagrammes.

2.5 TED

Toujours dans le cadre de la vérification de circuits arithmétiques, les diagrammes d'expansion de Taylor TEDs [CKA06] introduisent un nouveau formalisme pour la représentation des fonctions polynomiales à valeurs multiples. Comme son nom l'indique, un TED est basé sur la décomposition en série de Taylor III.4 d'une expression arithmétique donnée ; où $f'(x), f"(x)$, etc, sont les dérivées successives de f par rapport à x.

$$f(x, y, ..) = f(x = 0) + xf'(x = 0) + \frac{1}{2}x^2 f"(x = 0) + .. \tag{III.4}$$

Les termes de cette série seront ensuite décomposés successivement par rapport aux variables restantes (y, etc), admettant une seule variable à la fois. La figure III.6.a montre la décomposition de la fonction $f(x, y, ...)$ à un seul niveau par rapport à la variable x. Les nœuds dans ce graphe représentent les termes de l'expansion et les arrêtes représentent les cœfficients de cette expansion. Le nombre de fils à chaque nœud dépend de l'ordre de l'expression polynomiale par rapport à sa variable de décomposition en considérant ce nœud comme la racine. La figure III.6.b montre la représentation de la fonction $F3 = A^2 + AB + 2AC + 2BC$ en un TED.

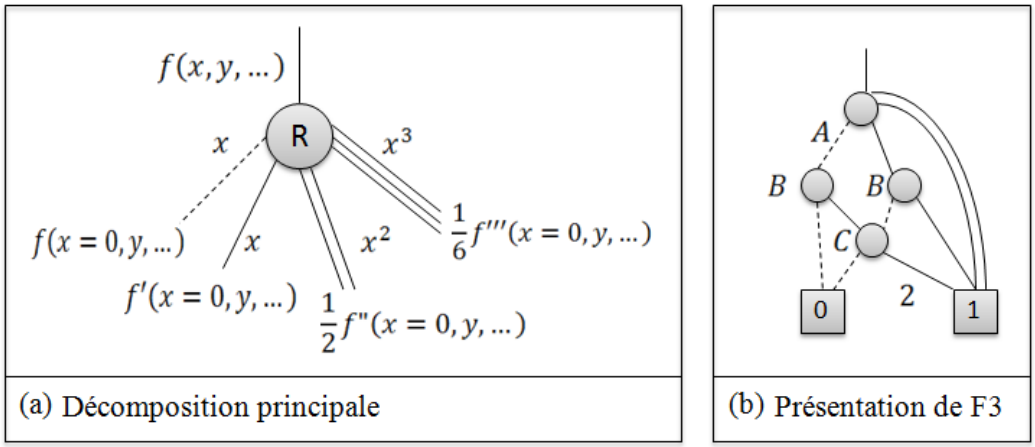


Figure III.6 – Présentation principale et en TED de $F3$

Les arêtes additives correspondants à $F(0)$ sont présentées en pointillés. Les arêtes linéaires de premier ordre, associées à $F'(0)$ sont présentées par des traits simples. Les arêtes de deuxième ordre, associées à $F''(0)$ sont présentées en double traits. L'expression présentée par ce graphe est calculée comme une somme des expressions de tous les chemins de la racine au nœud terminal 1. L'expression de chaque chemin est calculée à son tour par le produit des expressions de tous les arêtes de ce chemin. Une telle représentation favorise une factorisation et une extraction des sous-expressions communes à l'expression arithmétique d'origine.

Contrairement aux BMDs, un TED peut aussi bien être défini au niveau bit qu'au niveau mot (entiers). Il s'agit d'une représentation compacte et normalisée permettant une manipulation à différents niveaux d'abstractions. Pour un ordre fixe de variables, le graphe résultant est réduit, normalisé et canonique. Il est donc très important de trouver un ordonnancement optimal des variables. En effet, le choix d'ordre des variables dans un TED influe nettement la taille de l'espace mémoire nécessaire au stockage et le temps d'exécution du programme impliqué. Ce problème est en général NP complet[Wik12]. Par ailleurs, un TED permet de manipuler une architecture avec des additionneurs, des soustracteurs, des multiplicateurs et des registres de décalage, mais il ne couvre pas tous les autres opérateurs.

De plus, contrairement aux méthodes de présentation citées précédemment, le TED est basé sur un arbre non-binaire. Le nombre de fils d'un nœud TED dépend ainsi du degré de la variable correspondante. Par conséquent, un TED utilise une structure de données plus complexe. Il présente des séries infinies pour certaines fonctions (a^x) et la représentation de la fonction ($x < y$) par exemple est un enjeu important dans les TEDs.

2.6 Autres techniques

Parmi les méthodes permettant la vérification de spécifications de systèmes logiciels, Groote et Tveretina ont proposé dans [Gro] des Diagrammes de Décision (DD) pour la logique de premier ordre complète. En admettant une formule d'entrée dans une forme prénexe (forme précédée d'un quantificateur $\forall$ ou $\exists$), ils représentent la négation de son suffixe de quantification avec un DD, et prouvent une contradiction en utilisant la skolémisation (transformation de formules prénexes) et des stratégies standards telle que l'application des unificateurs.

Les diagrammes de décisions binaires équationnels EQBDD [GvdP00] ont été introduits pour décider des égalités entre des fonctions non interprétées. Les nœuds dans un EQBDD sont étiquetés par des prédicats (égalités) et des règles de réduction assurant l'exécution de la substitution selon un ordre prédéfini "$\leqslant$" entre les variables. Par exemple, si ($x \leqslant y$) alors y est substitué par x dans un sous-DD supérieur d'un nœud étiqueté par ($x = y$). Les BDDs équationnels sont semi-canoniques : Ils se réduisent à 0 (ou 1) s'ils présentent une contradiction (ou tautologie), mais ils sont non-canoniques autrement.

R. Cavada [CCF+07] propose une technique QELIM pour l'arithmétique linéaire (LA) qui combine les BDDs et les solveurs SMT. Ils mettent l'accent sur l'abstraction des prédicats, et considèrent le problème de quantification de l'ensemble des variables numériques à partir des formules de l'arithmétique linéaire sur les prédicats et les variables propositionnelles. La structure booléenne des formules est codée en BDDs. L'algorithme QELIM s'établit en traversant le

BDD d'une manière récursive, traitant ainsi l'ensemble de prédicats linéaires (autrement dit, le contexte) qui se produit tout au long de chaque chemin. A chaque récursion, un solveur SMT est employé ainsi pour vérifier si le contexte abordé est cohérent. Les chemins portant un contexte incohérent seront supprimés par la suite. L'utilisation du contexte exclut l'utilisation de programmation dynamique. Ainsi, l'algorithme est linéaire par rapport au nombre de trajets du BDD étudié. Cette approche est appelée "Boite noire" étant donnée qu'elle utilise une procédure de décision externe.

2.7 DDD

Un diagramme de décision de différence DDD [ML98] est une structure de données pour représenter une logique booléenne de premier ordre à travers des inégalités de la forme $\{x-y < c\}$ et $\{x - y \leqslant c\}$, dans lesquelles les variables peuvent être des entiers ou des réels. L'idée de base est de représenter ces formules logiques par des BDDs ayant des nœuds étiquetés par des prédicats atomiques, et de réduire la redondance en tirant profit des implications entre ces prédicats. Par exemple, un nœud étiqueté par $\{x - y \leqslant 10\}$ n'apparaît jamais comme un fils droit d'un nœud étiqueté par $\{x - y \leqslant 5\}$. Pour un ordre fixe de variables, un DDD d'une formule f n'est pas plus large qu'un BDD représentant une abstraction propositionnelle de cette fonction f.

Les DDDs permettent de déterminer des propriétés fonctionnelles (satisfiabilité et équivalence) et de vérifier d'une manière efficace les systèmes temporisés modélisés par exemple par des réseaux de Petri temporisés [BHR08]. Le succès de cette notation remonte à la représentation symbolique des nœuds et de leurs informations temporisées. D'ailleurs, une caractéristique importante des DDD est l'intégration de l'algorithme QELIM basé sur l'élimination de Fourier-Motzkin [BW94]. Cet algorithme se prête bien à la programmation dynamique, profitant ainsi de la structure DAG des diagrammes de décisions. En outre, les DDDs partagent de nombreuses propriétés avec les BDDs : ils sont ordonnés, ils peuvent être réduits permettant de vérifier les propriétés de tautologie et de satisfiabilité en un temps constant, et la plupart des algorithmes et des techniques pour les BDDs peuvent être généralisés pour être appliqués aux DDDs.

Néanmoins, les DDDs présentent trois principales limites. Tout d'abord, la logique de différence est trop restrictive pour de nombreuses tâches d'analyse de programmes. D'autant plus, les DDDs sont incapables de supporter l'ordonnancement dynamique des variables (DVO). Enfin, il n'existe aucun travaux à notre connaissance caractérisant l'efficacité des DDDs dans la résolution des problèmes d'analyse et de vérification.

3 LDD

Un diagramme de décision linéaire LDD [Cha09] est une structure de données pour représenter et manipuler des formules propositionnelles à travers une arithmétique linéaire. D'une manière formelle, un LDD présente un BDD avec des nœuds étiquetés par des prédicats atomiques et linéaires en satisfaisant une théorie d'ordonnancement et des contraintes de réduction locale. Ces diagrammes impliquent certaines propriétés qui les rendent efficaces pour la manipulation des variables numériques à travers l'arithmétique linéaire. Les structures les plus pertinentes pour décrire les LDDs sont celles qui étiquettent les nœuds avec des prédicats linéaires sur les réels. Ainsi, Les LDDs peuvent être considérés comme une extension des diagrammes DDD visant à couvrir toutes les fonctionnalités de l'arithmétique linéaire. Au-delà de cette extension, les LDDs apportent deux principales contributions : tout d'abord, ils établissent une extension de l'ordonnancement dynamique des variables DVO de BDDs au LDDs. Ensuite, ils développent, implémentent et évaluent plusieurs algorithmes existants de quantification.

Dans la pratique, les LDDs sont implémentés au dessus de CUDD (paquet pour la manipulation des BDDs) et à partir de la théorie UTVPI [LM05]. La théorie UTVPI (Unit Two Variable Per Inequality) est un type particulier de contraintes linéaires ayant au plus deux variables avec des coefficients égaux à 1 ou -1. Un ensemble de UTPVI-atomes est donc présenté sous la forme $\{ax + by \leqslant k\}$ où $x, y \in Var; a, b \in \{-1, 1\}; k \in Z$ et Var est l'ensemble des variables. Deux UTPVI-atomes A et B sont équivalents si et seulement si l'un implique l'autre. Cette théorie est également connue sous le nom d'Octogones [Min06].

Dans le même contexte de la théorie d'Octogones, B. Jeannet a introduit une nouvelle bibliothèque appelée Apron [JM09] dédiée à l'analyse statique des programmes à variables numériques par l'Interprétation Abstraite. Apron caractérise plusieurs domaines numériques abstraits tels que les octogones et les polyèdres convexes. Il définit en particulier une sémantique précise et concrète pour un ensemble d'opérations, y compris l'arithmétique non-linéaire et à virgule flottante favorisant ainsi une approximation plus juste. Une analyse principale d'une présentation par Apron consiste à appliquer un calcul de points fixes sur le domaine abstrait au lieu de l'explorer comme un graphe fini. Ceci amène à des résultats conservatifs : si un état devient inaccessible au cours d'une itération, cette propriété restera vraie pour toutes les itérations suivantes. Cependant, il n'est pas possible de prouver qu'une telle propriété n'est pas vraie sur tout l'ensemble. Cette bibliothéque a besoin d'être améliorée encore afin de couvrir le phénomène de non convexité ainsi que d'autres problématiques. D'où le choix de la théorie UTVPI pour l'expérimentation des LDDs.

3.1 Définition

Partant sur une théorie T, un LDD est défini par un graphe acyclique orienté ayant :
- Deux nœuds terminaux étiquetés respectivement par 0 et 1 ;
- Des nœuds non terminaux. Chaque nœud non terminal "u" est étiqueté par un prédicat ou un terme atomique (T-atome) noté C(u) et posséde deux descendants désignés par H(u) et L(u) ;
- Des arêtes (u, H (u)) et (u, L (u)) pour chaque nœud non terminal "u".

Considérons dans la suite A_T l'ensemble de termes atomiques cohérents, attr(u) le triplet (C(u),H(u),L(u)) et l'ensemble suivant des fonctions définies par la théorie T :
- Implication, IMP : $A_T \times A_T \rightarrow Bool$ tel que $IMP(c_1, c_2)$ ssi $(c_1 \Longrightarrow c_2)$;
- Normalisation, N : $A_T \longmapsto A_T$ tel que si $c \Longleftrightarrow c'$ ou $c \Longleftrightarrow \neg c'$ alors $N(c) = N(c')$;
- Négation, NEG : $A_T \longmapsto A_T$ tel que $NEG(c) \Longleftrightarrow \neg c$.
- Résolution, RSLV : $A_T \times Var \times A_T \cup \{TRUE\}$ tel que $RSLV(c_1, x, c_2) = c_3$ ssi $x \in VARS(c_1) \cap VARS(c_2)$, et $c_3 \Longleftrightarrow \exists x.c_1 \wedge c_2$; où $VARS$ est l'ensemble des variables. RSLV présente ses entités par un ensemble de T-atomes comme suit : $RSLV(S, x, c) = \bigwedge_{s \in S} RSLV(s, x, c)$.

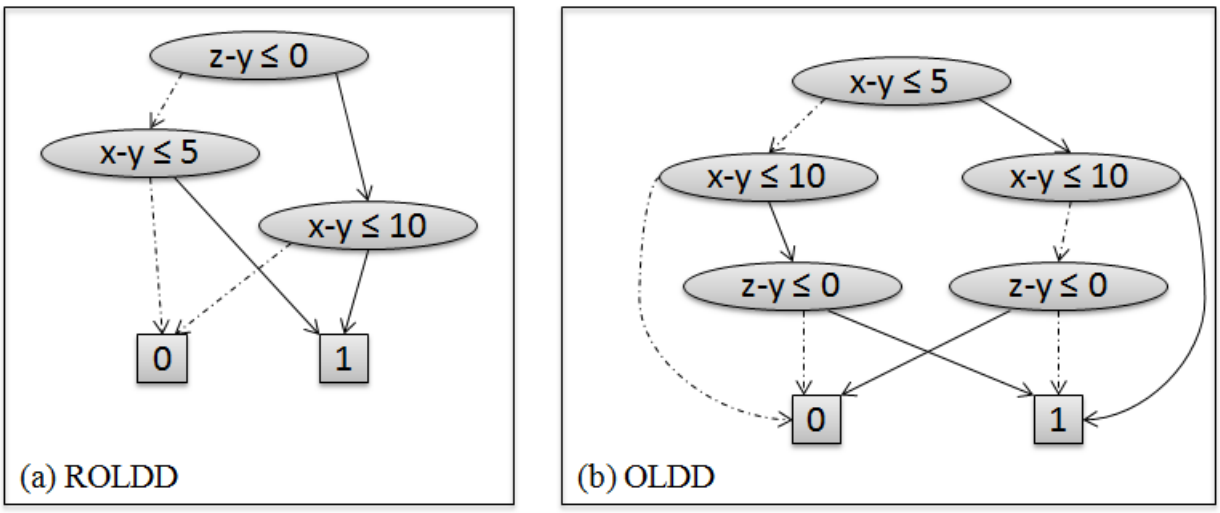


Figure III.7 – (a) ROLDD ordonné par $\{z - y \leqslant 0\}; \{x - y \leqslant 5\}; \{x - y \leqslant 10\}$ et (b) OLDD ordonné par $\{x - y \leqslant 5\}; \{x - y \leqslant 10\}; \{z - y \leqslant 0\}$

La figure III.7.a montre un exemple d'une présentation en LDD de la formule arithmétique $F4 = (z - y \leqslant 0 \wedge x - y \leqslant 10) \vee (z - y > 0 \wedge x - y \leqslant 5)$. Ce graphe est ordonné et réduit ROLDD suivant un choix particulier d'ordre des termes $\{z - y \leqslant 0\} \leq \{x - y \leqslant 5\} \leq \{x - y \leqslant 10\}$.

En optant pour un ordre différent OLDD dans un deuxième cas $\{x - y \leqslant 5\} \leq \{x - y \leqslant 10\} \leq \{z - y \leqslant 0\}$, cette formule peut être représentée d'une manière différente à travers la figure III.7.b. En effet, en exploitant le fait qu'une formule $\{(F \wedge x - y \leqslant 5) \vee (F \wedge x - y > 5)\}$ est toujours vraie, l'équation $\{(z - y \leqslant 0) \wedge (x - y \leqslant 10)\}$ se transforme à l'union suivante $\{(z - y \leqslant 0) \wedge (x - y \leqslant 10) \wedge (x - y \leqslant 5)\} \vee \{(z - y \leqslant 0) \wedge (x - y \leqslant 10) \wedge (x - y > 5)\}$. De même $\{(z - y > 0) \wedge (x - y \leqslant 5)\}$ devient $\{(z - y > 0) \wedge (x - y \leqslant 5) \wedge (x - y \leqslant 10)\} \vee \{(z - y > 0) \wedge (x - y \leqslant 5) \wedge (x - y > 10)\}$. Ce processus de calcul permet à la fin diverses représentations de la formule F4.

3.2 Réduction locale

Un LDD est réduit localement si et seulement si les cinq conditions suivantes sont satisfaites sur chaque nœud interne u et v :

1. Pas de nœuds dupliqués : $attr(u) = attr(v) \Longrightarrow u = v$
2. Pas de nœuds redondants : $L(v) \neq H(v)$.
3. Les étiquettes sont normalisées : $C(v) = N(C(v))$.
4. Implication forte : $\neg IMP(C(v), C(H(v)))$.
5. Implication faible. $IMP(C(v), C(L(v))) \Rightarrow H(v) \neq H(L(v))$.

Algorithm 1 MK : Construction d'un LDD

```
1: fonction MK(A_T c, LDDf, LDDg)
2: if (IMP(c, C(f))) then
3:     f ← H(f) // Ne considérer que le descendant H de f.
4: end if
5: if (f = g) then
6:     renvoie g
7: end if
8: if IMP(c, C(g)) ∧ f = H(g)) then
9:     renvoie g
10: end if
11: renvoie BDD − NODE(c, f, g)
```

Par exemple, le LDD de la figure III.7.a est réduit. Ce qui n'est pas le cas du LDD de la figure III.7.b. La fonction $MK(c, f, g)$ de l'Algorithme 1, construit un ROLDD de la fonction $ITE(c, f, g)$, où C est une contrainte normalisée ; $\{c \leqslant_T C(f)\}$ et $\{c \leqslant_T C(g)\}$; f et g sont deux ROLDDs distincts. Les lignes 2 à 10 assurent que le résultat est réduit. Pour terminer, la

ligne 11 retourne un diagramme unique de nœuds représentant la fonction ITE, étant donné que les LDDs s'appuient sur la structure des BDDs qui est forcément canonique. La preuve d'exactitude de MK est basée sur les deux règles de réduction III.5 et III.6 pour mettre en vigueur les règles d'implication forte et les règles d'implication faible, respectivement.

$$\frac{ITE(x, ITE(y, h, l), z) \quad op \quad ITE(u, v, w) \quad IMP(x, y)}{ITE(x, h, z)} \tag{III.5}$$

$$\frac{ITE(x, y, ITE(z, h, l)) \quad IMP(x, x) \quad y \Longleftrightarrow h}{ITE(z, h, l)} \tag{III.6}$$

Ces deux règles sont exprimées sous forme de Règles de Réécriture $\frac{P}{Q}$, où Q est réalisée si toutes les conditions de P sont satisfaites. La figure III.8 montre un exemple d'une implication forte. Cette implication applique une résolution aux termes $\{t \leqslant 5\}$ et $\{t \leqslant 9\}$ pour obtenir l'espace le moins large $\{t \leqslant 5\}$. De l'autre coté, la figure III.9 montre la résolution de ces deux termes par la règle d'implication faible pour retrouver l'espace le plus large $\{t \leqslant 9\}$.

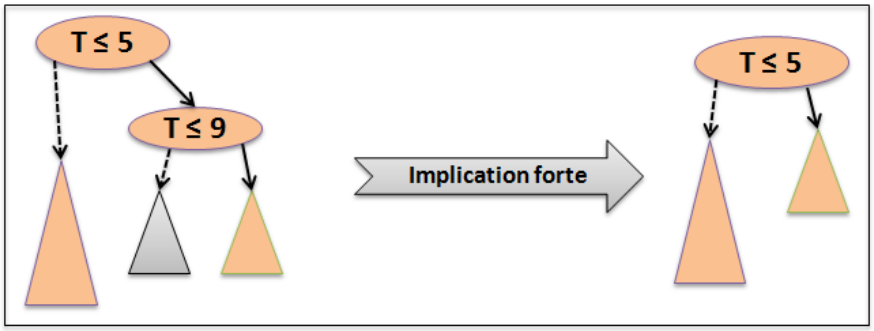


Figure III.8 – Implication forte

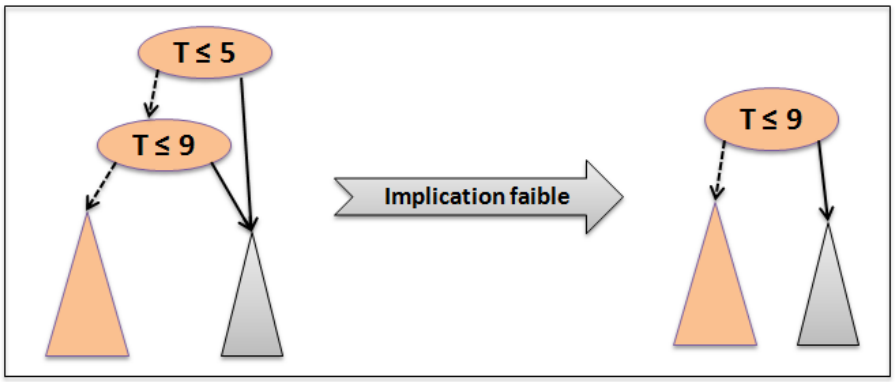


Figure III.9 – Implication faible

3.3 Opérations de base des LDDs

Pour tout opérateur symétrique "op" qui se distribue à travers la fonction ITE, un LDD pour f op g est construit par le biais de la fonction APPLIQUER (Op op, LDD f, g LDD). Cette dernière est basée sur les transformations III.7, III.8, III.9 et leurs versions symétriques obtenues en échangeant les arguments de l'opérateur op (si plusieurs règles s'appliquent, la première est sélectionnée).

$$\frac{ITE(x,y,z)\ op\ ITE(u,v,w) x \Longleftrightarrow u}{ITE(x, y\ op\ v, z\ op\ w)} \tag{III.7}$$

$$\frac{ITE(x,y,z)\ op\ ITE(u,v,w)\ IMP(x,u)}{ITE(x, y\ op\ v, z\ op\ ITE(u,v,w))} \tag{III.8}$$

$$\frac{ITE(x,y,z)\ op\ ITE(u,v,w) x \leqslant_T u}{ITE(x, y\ op\ ITE(u,v,w), z\ op\ ITE(u,v,w))} \tag{III.9}$$

Les opérateurs de conjonction LDD (ET), de disjonction (OU) et du triplet ITE (If-Then-Else) sont implémentés à traves la fonction APPLIQUER. La négation de LDD (NOT) est implémentée par contre d'une façon analogue au BDD. En résumé, l'extension des opérations booléennes de BDD vers des opérations en LDDs est réalisée simplement en décorant le BDD par les informations supplémentaires des LDDs. La difficulté est plutôt dans la projection des variables numériques (quantification QELIM) et l'extension de l'ordonnancement dynamique des variables.

3.4 Ordonnancement des variables

Depuis 1993, l'algorithme d'ordonnancement dynamique DVO introduit par Rudell [Rud93] a été utilisé pour les diagrammes de décisions DDs. Cela est particulièrement vrai pour les LDDs à condition de rapporter des restrictions sur l'ordre des variables. L'exemple suivant montre que le standard DVO ne peut pas être appliqué tel qu'il a été défini à l'origine dans le cas des LDDs. Le changement d'ordre des niveaux des termes suivant DVO du graphe de la figure III.10.a montre une structure différente dans la figure III.10.b, ayant (u, (v, f11, f10), (v, f01, f00)) au lieu de (v, (u, f11, f01), (u, f10, f00)). Cette structure peut être inadaptée dans le cas où ($f00 = f10$) ou ($f01 = f11$) amenant à la réduction du graphe (a) mais non pas du graphe (b). Ceci augmentera nettement la complexité de ce graphe par rapport au nombre de nœuds étiquetés par "u".

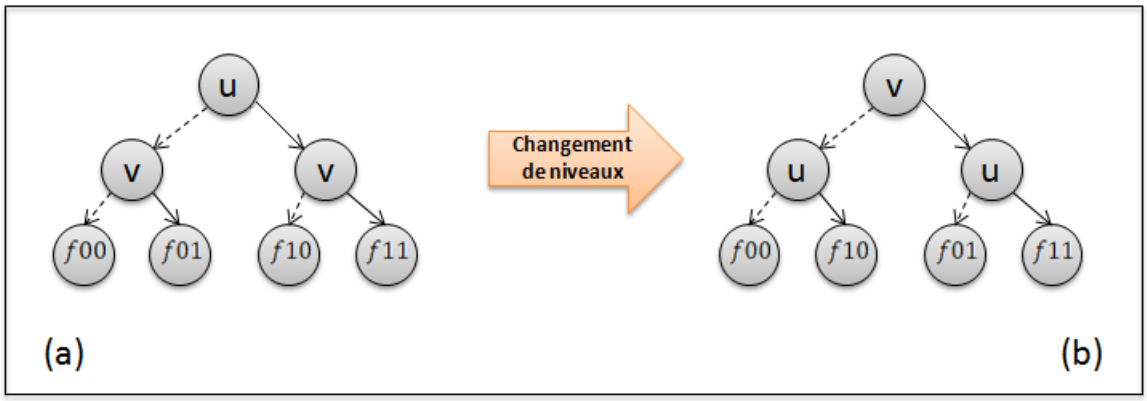


Figure III.10 – Application de l'ordonnancement DVO

En considérant l'exemple du LDD de la figure III.7.a, le fait d'effectuer une permutation libre des différents termes adjacents contredit clairement les contraintes définies au bon ordonnancement des LDDs. En effet, en permutant les termes adjacents $\{z - y \leq 0\}$ et $\{x - y \leq 5\}$, le résultat obtenu n'est pas du tout ordonné car le nœud étiqueté par $\{z - y \leq 0\}$ est situé entre les deux nœuds $\{x - y \leq 5\}$ et $\{x - y \leq 10\}$. L'idée est donc de permuter des groupes de variables à la fois. Il aurait ainsi fallu permuter $\{z - y \leq 0\}$ avec $\{x - y \leq 10\}$. Ainsi, le LDD final est désormais bien ordonné dans la figure III.7.b, mais il n'est pas réduit ! La règle d'implication forte est violée. De même, il est facile de montrer dans d'autres exemples que la règle d'implication faible peut être également violée.

Pour contrecarrer ces phénomènes, S.Chaki a remplacé l'ordonnancement fondamental à niveaux pour les BDDs par un ordonnancement entre les termes atomiques (ordonnancement T-atomique) pour les LDDs. Cette nouvelle technique d'ordonnancement a permis une construction facile des diagrammes réduits pour les LDDs.

Etant donné un ordre partiel noté $\leqslant_{IMP}$ définit par IMP : $c_1 \leqslant_{IMP} c_2 \Longleftrightarrow IMP(c_1, c_2)$, un ordonnancement T-atomique est défini par chaque ordre total $\leqslant_T$ qui étend $\leqslant_{IMP}$ à un ordre total dans A_T. Un LDD "u" est donc ordonné suivant $\leqslant_T$ si et seulement si pour chaque nœud v accessible à partir de u, $\{C(v) \leqslant_T C(H(v))\}$ et $\{C(v) \leqslant_T C(L(v))\}$. Un LDD "u" est dit également bien ordonné s'il est ordonné suivant certains T-termes atomiques. Cette technique d'ordonnancement est implémentée dans un nouveau algorithme appelé GROUPMOVE [Cha09]. GROUPMOVE considère deux ensembles ordonnés de T-termes atomiques $X = \{x_1, .., x_{|x|}\}$ et $Y = \{y_1, .., y_{|Y|}\}$, et les permutent dans l'ordre des variables. L'ordre avant GROUPMOVE était $x_1, .., x_{|x|}, y_1, .., y_{|Y|}$. Après GROUPMOVE, l'ordre devient $y_1, .., y_{|Y|}, x_1, .., x_{|x|}$. Ceci s'établit à partir de l'opération $|X| \times |Y|$ en faisant appel à la fonction LDDSWAPINPLACE décrite dans l'algorithme 2.

Algorithm 2 Permutation de nœuds adjacents dans un LDD

1: **fonction** $LDDSWAPINPLACE(A_T\ x, A_T\ y)$
2: Remplacer chaque LDD $F : (x, H, L)$ par (y, G_1, G_0), où
3: F_{00}, F_{01} sont les $\neg y$ et y cofacteurs de L.
4: F_{10}, F_{11} sont les $\neg y$ et y cofacteurs de H.
5: $F'_{11} \longleftarrow IMP(y, C(F_{11}))\ ?\ H(F_{11}) : F_{11}$.
6: $F'_{01} \longleftarrow IMP(y, C(F_{01}))\ ?\ H(F_{01}) : F_{01}$.
7: **if** $F'_{11} = F'_{01} \bigvee (IMP(x, C(F'_{01})) \bigwedge F_{11} = H(F'_{01}))$ **then**
8: $\quad G_1 = F'_{01}$
9: **else**
10: $\quad G_1 \longleftarrow (x, F'_{11}, F'_{01})$
11: **end if**
12: **if** $F_{00} = F_{10} \bigvee (IMP(x, C(F_{00})) \bigwedge F_{10} = H(F_{00}))$ **then**
13: $\quad G_0 = F_{00}$
14: **else**
15: $\quad G_0 \longleftarrow (x, F_{10}, F_{00})$
16: **end if**

Cette fonction est à l'origine une variante de la fonction SWAPINPLACE de l'algorithme de Rudell, mais elle maintient de plus les réductions des LDDs. Les lignes 7 et 12 permettent de s'assurer que tout LDD nouvellement construit n'a pas de nœuds redondants et satisfait la règle de l'implication forte et faible. Les lignes 5 et 6 permettent d'un autre côté de garantir la conservation de l'implication forte : si un nœud v est accessible à partir d'un nœud u à travers H(u), alors C(u) n'implique pas C(v). Ceci est crucial pour assurer que la règle d'implication forte soit établie à la fin de GROUPMOVE. Ce type d'ordonnancement a montré une réduction significative de la taille des LDDs par rapport à l'ancienne version d'ordonnancement DVO.

3.5 Quantification existentielle QELIM pour les LDDs

Une quantification permet en général de formuler des propositions mathématiques dans le calcul des prédicats du premier ordre. Dans ce cadre, les LDDs définissent deux types de quantifications pour la représentation de ses prédicats atomiques : QELIM en boîte noire et QELIM en boîte blanche. Une quantification QELIM en boîte noire applique un solveur QELIM externe pour chaque chemin dans les LDDs. Ce processus est linéaire par rapport au nombre de chemins ce qui est contraignant, mais il est compatible avec n'importe quel type de solveur ce qui est par contre positif. En revanche, la quantification QELIM en boîte blanche est une variante généralisée et optimisée de la version DDD. Elle applique d'une manière récursive une résolution par paires des différents nœuds DD, à la façon de Fourier-Motzkin.

Cet algorithme fonctionne bien dans la pratique. Il est dans le pire des cas exponentiel par rapport à la taille du diagramme. Grâce à la quantification, les LDDs peuvent être utilisés comme un SMT-solveur pour l'arithmétique linéaire : il suffit de construire le LDD de la formule donnée et quantifier par la suite toutes les variables numériques pour décider si la formule est satisfaisable.

3.5.1 QELIM en boite noire

La quantification en boite noire appelée BBQE (Black Box QELIM) applique une théorie QELIM externe pour chaque chemin du LDD. Cette application nécessite la présence des fonctions auxiliaires suivantes :

- $THUNSAT(P)$, décide si l'ensemble P de T-atomes ($\bigwedge P$) est un-satisfaisable.
- $THQELIM(V, P)$ calcule un ensemble P' de T-atomes pour un ensemble V de variables tel que $\bigwedge P'$ soit équivalente à $\exists V. \wedge P$.

L'exécution de la quantification $BBQE(f, P, V)$ est linéaire par rapport au nombre de chemins de f et exponentielle par rapport à la taille du diagramme correspondant. Cependant, $BBQE$ n'est pas compatible avec la programmation dynamique étant donné qu'il propage le contexte P tout au long de chaque branche d'un LDD.

3.5.2 QELIM en boite blanche

La quantification en boite blanche applique l'élimination de Fourier-Motzkin directement sur les LDDs. Elle étend l'algorithme QELIM des DDDs à n'importe quel fragment adéquat T de l'arithmétique linéaire. Soit φ une conjonction de $T - atomes$, et x une variable numérique, une élimination de Fourier-Motzkin de x à partir de φ est définie comme suit : S est initialement l'ensemble de tous les atomes de φ, et S' est égal à l'ensemble vide. Pour chaque $\{p \in S\}$, p est éliminé de S. Si x n'appartient pas à $VARS(p)$, alors p est ajouté à S'. Sinon, pour chaque $\{t \in S\}$ tel que $\{x \in VARS(p)\}$, $\{RSLV(p, x, t)\}$ est ajouté à S'. Où $RSLV(p, x, t)$ est vraie quand x se produit dans la même phase de p et t. A la fin, $\bigwedge S'$ est équivalent à $\exists x.\varphi$. Considérons par exemple la formule suivante $\{\exists y.(x - y \leqslant 5) \bigwedge (x - z \geqslant 8) \bigwedge (y - z \leqslant 10)\}$. L'application de Fourier-Motzkin permet de résoudre les deux termes $\{(x - y \leqslant 5)\}$ et $\{(y - z \leqslant 10)\}$ pour obtenir à la fin la formule $\{(x - z \geqslant 8) \bigwedge (x - z \leqslant 15)\}$.

L'idée clé de quantification en boite blanche est d'appliquer l'algorithme de Fourier-Motzkin d'une manière simultanée à chaque chemin d'un LDD. Une étape importante de cet algorithme est la résolution simultanée. Elle s'établit essentiellement par la fonction $DR(S, x, f)$ ($DagResolve$) qui considère un ensemble S de $T - atomes$, une variable x, et un LDD f, et

renvoie par la suite le LDD obtenu en ajoutant à chaque chemin $\prod$ de f les résolvantes de S et $\prod$ sur x. Implémentée à travers une programmation dynamique, le nombre d'appels récursifs à la fonction DR pour la résolution simultanée se trouve linéaire par rapport au nombre de nœuds définis dans f. Cependant, cette opération nécessite à la fin la restauration de l'ordre des termes, qui est, comme en DDD, exponentielle par rapport à la taille de f dans le pire des cas.

Algorithm 3 WBQE1 : QELIM en boite blanche

1: **fonction** $WBQE1(x \in Var, LDD\ f)$
2: **if** $f = 1 \vee f = 0$ **then**
3: renvoie f.
4: **end if**
5: **if** $x \notin VARS(C(f))$ **then**
6: renvoie $ITE(f, WBQE1(x, H(f)), WBQE1(x, L(f)))$
7: **end if**
8: $t \longleftarrow WBQE1(x, DR(\{C(f)\}, xH(f)))$
9: $e \longleftarrow WBQE1(x, DR(\neg C(f), xH(f)))$
10: renvoie OR(t,e)

Le principe de base de quantification en boite blanche appelé WBQE1 est représenté par l'algorithme 3. Il présente deux comportements différents à chaque itération : soit il descend aux branches de f (ligne 6), soit il supprime un nœud supérieur de f dont l'étiquette contient x (lignes 8-10). L'algorithme WBQE1 se termine au moment où à chaque itération le nombre de nœuds étiquetés par un atome contenant x diminue.

L'algorithme WBQE2 améliore WBQE1 en réduisant le nombre d'appels à la fonction DR. Ceci amène à la même complexité de QELIM en BDD dans le meilleur des cas.

3.5.3 Elimination de variables multiples

Contrairement à BBQE, la quantification QELIM en boite blanche élimine une seule variable à la fois : c'est une limitation fondamentale de Fourier-Motzkin. Cependant, de nombreuses applications de QELIM dans l'analyse de certains programmes exigent l'élimination de plusieurs variables à la fois. Une solution à ce problème est de proposer une stratégie appelée WBMVQE d'élimination à chaque itération. Cette stratégie se répartit entre la suppression des contraintes ayant des variables qui n'étaient pas résolues durant Fourier-Motzkin, et le choix et l'élimination d'une variable quantifiée qui résulte d'un nombre minimum de résolutions. Cette stratégie a fourni une accélération considérable à QELIM.

L'ordre dans lequel ces variables sont éliminées est crucial. Considérons par exemple la formule suivante $\{x - y \leqslant 1 \bigwedge z - x \leqslant 2 \bigwedge w - z \leqslant 3\}$. L'élimination des deux variables x et y peut se faire de deux façons : commencer par x ou commencer par y. Dans le premier cas, $\{x - y \leqslant 1\}$ est initialement éliminé et résolu avec $\{z - x \leqslant 2\}$ pour obtenir $\{z - y \leqslant 3 \bigwedge z - x \leqslant 2 \bigwedge w - z \leqslant 3\}$. Ensuite, $\{z - x \leqslant 2\}$, et enfin $\{z - y \leqslant 3\}$ sont éliminés pour obtenir $\{w - z \leqslant 3\}$. Dans le second cas, $\{x - y \leqslant 1\}$, puis $\{z - x \leqslant 2\}$ sont supprimées. Aucune résolution n'est nécessaire. Cet exemple met en évidence deux propriétés importantes. Tout d'abord, éliminer les variables qui se produisent en moins de T-atomes amène à moins de résolutions et les résultats intermédiaires sont potentiellement plus petits. Deuxièmement, les variables qui se produisent dans un seul T-atome sont éliminées par l'abandon de leurs atomes sans procéder à la résolution. Comme il n'y a pas de résolution, plusieurs de ces variables peuvent être éliminées à la fois. L'algorithme WBMVQE de l'élimination de variables multiples se répète à travers deux principales phases. Tout d'abord, il élimine toutes les variables qui ne nécessitent pas de résolution. Cette opération se répète jusqu'à l'élimination de toutes les variables à supprimer. Enfin, elle élimine une variable en utilisant WBQE2. Ce processus se répète également jusqu'à l'élimination de toutes les variables.

Pour résumer, l'étude que nous avons muni sur les différents diagrammes de manipulation de données a montré que les LDDs fournissent la meilleure manipulation de données numériques dans notre contexte d'étude. Par conséquent, nous avons choisi de se baser sur la bibliothèque des LDDs dans notre approche, que nous décrivons dans le chapitre suivant, en exploitant l'efficacité de ses algorithmes d'ordonnancement GROUPMOVE et de quantification WBMVQE.

4 Outils inhérents à la génération de jeux de tests

La sécurité et la validation des systèmes critiques ont donné naissance à plusieurs outils de génération de jeux de tests exhaustifs et non exhaustifs. Ces outils manipulent leurs données de manières différentes. En particulier, ils permettent ou excluent la manipulation des données numériques. Nous présentons dans ce paragraphe une liste d'outils que nous souhaitons la plus exhaustive possible dans le cadre de notre mémoire.

Lutess V2 [SP07] est un environnement de test, écrit en Lustre, pour les systèmes réactifs synchrones. Il génère automatiquement des tests qui alimentent d'une façon dynamique le programme sous test à partir de la description formelle de son environnement et de ses propriétés. La version V2 de Lutess permet de gérer des entrées et sorties numériques contrairement à la première version de Lutess [dBZ99]. Lutess V2 est basée sur la programmation logique par

contraintes (CLP) et permet d'introduire des hypothèses sur le programme sous test. Grâce aux capacités des solveurs CLP, il est possible d'associer des probabilités d'occurrence pour chaque expression booléenne. Cependant, cet outil nécessite la conversion des modèles testés en modèle lustre, une telle conversion manuelle peut être la cause de certains aléas au cours des tests.

B. Blanc présente dans [BJM+10] un outil de test structurel appelé GATeL, également basé sur CLP. GATeL permet d'identifier la séquence qui satisfait à la fois l'invariant et l'objectif du test par la résolution du problème de contraintes sur les variables du programme. Contrairement à Lutess, GATeL interprète le code Lustre, commence par l'état final pour revenir vers l'état initial. Cette technique nécessite une intervention humaine importante, ce qui est à éviter strictement dans notre étude.

C. Jard et T. Jeron, présentent dans [JJ05] un outil puissant appelé TGV (génération de tests avec la technologie de vérification) pour la génération de tests à partir de diverses spécifications de systèmes réactifs. Cet outil prend en entrée une spécification et l'objectif du test au format IOLTS (Input Output Labeled Transition System) puis génère les cas de test au même format IOLTS. TGV effectue plusieurs opérations : tout d'abord, il identifie les différentes séquences possibles à partir du produit cartésien de la spécification par l'objectif de test (contraintes liées au système). Ce produit cartésien engendre un grand nombre de transitions infranchissables à cause des contraintes non respectées. De ce fait, TGV sélectionne uniquement les cas de test réels à travers la recherche des états accessibles à partir de l'état initial et des états co-accessibles depuis les états qui l'acceptent. Une limitation importante de cet outil découle du traitement non symbolique (énumératif) des données. Les cas de test qui en résultent peuvent être assez volumineux et par conséquent difficiles à interpréter.

D. Clarke développe une extension de TGV dans [PJJ07], présentant un outil de génération de jeux de tests symboliques appelé STG. Ce dernier ajoute le traitement symbolique des données en exploitant les capacités de l'outil OMEGA [Pug91]. Les cas de tests générés sont par conséquent plus petits et plus lisibles que ceux effectués à travers les approches énumératives dans TGV. STG produit des cas de test à partir d'une spécification en IOSTS (Input Output Symbolic Transition System) et de l'objectif du test. Malgré son efficacité, cet outil n'est plus maintenu, ce qui est fort regrettable.

L'approche [BPZ] décrit STS (Systèmes de Transitions Symboliques), très souvent utilisé dans le test de différents systèmes. STS améliore la lisibilité et l'abstraction des descriptions du comportement du système par rapport aux formalismes avec des types de données limités. STS aborde également le problème de l'explosion d'espace d'états grâce à la mise en œuvre d'observateurs et de paramètres typés liés aux transitions. Pour le moment, la hiérarchie STS n'est pas très adaptée en dehors du domaine des systèmes temporisés/hybrides ou des systèmes

bien structurés. De tels systèmes sortent du cadre de notre étude.

ISTA (Intégration et Automatisation des Tests de Systèmes) [Xu11] présente un outil intéressant pour la génération automatique du code de test de haut niveau pour les réseaux de Pétri. ISTA génère du code test exécutable à partir d'une description de l'implémentation du modèle MID (Model Implementation Description). Les éléments du réseaux de Petri sont ensuite combinés en vue d'une implémentation. ISTA peut être efficace sur des tests de sécurité pour lesquels les réseaux de Petri génèrent des séquences de menaces. Cependant, cet outil est adapté uniquement à la vérification des propriétés de vivacité, tandis que notre intérêt porte plutôt sur la vérification des propriétés de sûreté.

J. Burnim présente dans [BS08], un outil de test des programmes en C appelé CREST. Il insère du code intermédiaire CIL (C Intermediate Language) dans le programme cible. Ceci permet une exécution symbolique en même temps que l'exécution concrète. Les contraintes sur les chemins sont résolues par la suite en utilisant un solveur Yices [DDM06]. CREST raisonne uniquement sur l'arithmétique linéaire d'une manière symbolique. Étroitement lié à CREST, KLOVER [LGR11] est un outil de génération automatique de tests et d'exécution symbolique pour les programmes en C++. Il présente un outil efficace et pratique pour la gestion des applications industrielles. Les deux outils CREST et KLOVER ne sont pas utiles dans notre approche, étant donné qu'ils fournissent des tests sur des systèmes réels, alors que nous ciblons des tests sur des systèmes en phase de conception.

5 Conclusion

Les Diagrammes de Décision pour les théories autres que la logique propositionnelle ont été largement étudiés depuis le début des années 90, en capitalisant sur le succès des BDDs ainsi que de ses nombreuses optimisations. Parmi ceux-ci, nous avons introduit les MTBDDs, EVBDDs, BMDs, TEDs qui sont des diagrammes restreints aux variables de domaines finis. Nous nous sommes intéressés plutôt aux structures qui étiquettent les nœuds par des prédicats linéaires sur les réels comme les DDDs et les LDDs. Dans l'ensemble, nous avons conclu que les LDDs, en combinaison avec les algorithmes DVO et QELIM, conduisent à une structure de données efficace et pertinente pour l'analyse des programmes à fonctions numériques. Nous avons conclu également à la fin de ce chapitre que la plupart des outils de tests existants n'assurent pas la couverture de tous les cas possibles, ou certains parmi eux n'assurent que le traitement booléen des variables. Nous présentons ainsi dans le chapitre suivant notre contribution pour assurer une vérification exhaustive et automatique des systèmes avec le traitement numérique des données.

Chapitre IV

Approche de génération exhaustive de jeux de tests

1 Introduction

Dans le but de compenser certaines insuffisances des outils présentés dans le chapitre précédent, nous introduisons dans ce chapitre notre approche de génération automatique de jeux de tests. Elle permet la génération d'une liste exhaustive des cas de tests possibles d'un système étudié en assurant une manipulation symbolique de ses données numériques. Nous présentons à cette fin les concepts nécessaires et un ensemble d'algorithmes qui interagissent entre eux pour obtenir le résultat escompté : une modélisation efficace du système étudié par l'approche synchrone, le choix du langage synchrone approprié, la mise en œuvre d'une méthode d'aplatissement simple, une méthode de compilation et une méthode significative d'exploration de l'espace d'états, une manipulation efficace des données numériques du système et enfin une méthode de marche en arrière "Backtrack" pour aboutir aux différents cas de test possibles.

2 Architecture globale

Le principe de notre approche se base sur six principales opérations. La figure IV.1 montre l'architecture globale de notre test décrivant un modèle global en entrée, une opération de quasi-aplatissement, une phase de compilation, une génération symbolique de jeux de tests en forte interaction avec l'entité qui manipule numériquement les données, et enfin une opération de génération de cas de tests possibles.

1. Modèle global : il présente l'entrée principale du test. Il doit être conforme à la spécification du système à tester. L'architecture de ce modèle est constituée suivant l'approche synchrone de machines hiérarchiques parallèles qui décrivent le fonctionnement global du système.

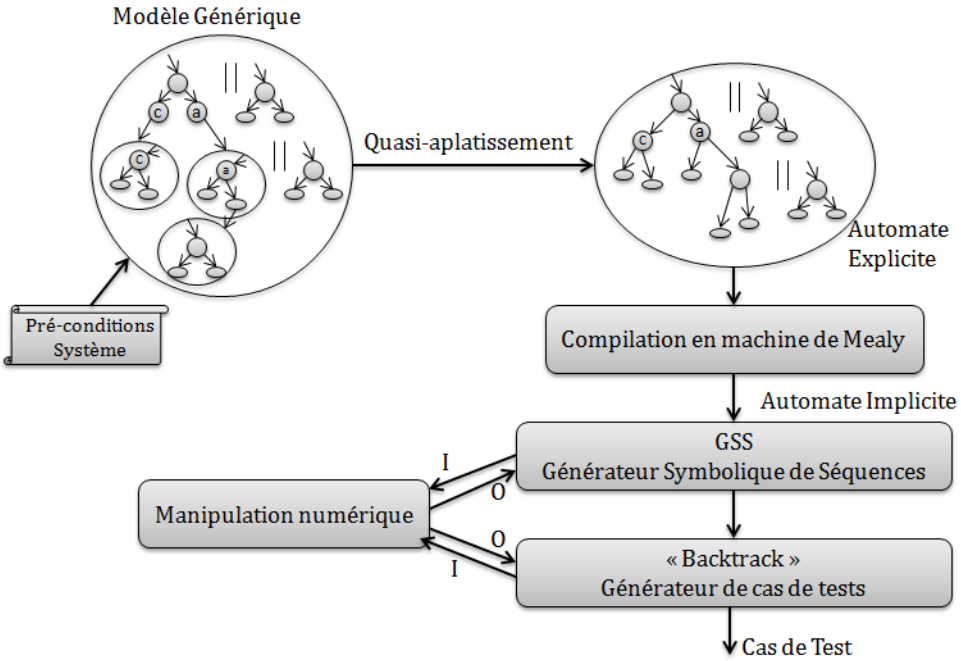


Figure IV.1 – Architecture globale du générateur de tests

2. Processus de quasi-aplatissement : ce processus permet d'aplatir les automates hiérarchiques tout en conservant le parallélisme. Ceci offre un modèle simple à analyser et apporte plus de souplesse pour identifier toutes les évolutions possibles du système.

3. Processus de compilation : il génère un automate implicite représenté par une machine de Mealy à partir d'un automate explicite. Ce processus compile le modèle, vérifie le déterminisme de tous les automates et assure la persistance du comportement du système.

4. SupLDD (Manipulation numérique des données) : elle se base sur la puissance des LDDs décrits dans le chapitre précédent pour exprimer et caractériser les pré-conditions du système par des contraintes numériques. Ces pré-conditions auront un rôle très important dans les opérations suivantes de génération symbolique de séquences et de marche en arrière "Backtrack".

5. GSS (Générateur Symbolique de Séquences) : il identifie les conditions préalables qui conduisent à des états significatifs spécifiques du système à partir de séquences générées. La liste exhaustive des cas de tests possibles sera ainsi présentée d'une manière automatique à travers ces pré-conditions plutôt que par une présentation manuelle et concrète de toutes les combinaisons possibles des commandes du système.

6. Backtrack : ce processus de marche en arrière permet de générer tous les cas de tests possibles. Il s'établit par la recherche d'un chemin qui satisfait les pré-conditions de l'état étudié.

3 Modèle global

Le modèle global est construit à partir de la spécification détaillée du système étudié. En effet, la spécification est interprétée automatiquement au langage synchrone approprié. Le choix du langage peut être effectué par l'utilisateur en choisissant celui qui convient le mieux à la nature du comportement du système. Enfin, le modèle final est présenté sous forme de machines à états finis "FSMs" hiérarchiques mises en parallèles. Le modèle hiérarchique décrit le fonctionnement global du système, tandis que les automates en parallèle capturent des informations de contrôle du système (système verrouillé, session fermée, etc.). L'ensemble de ces automates interagissent en continu entre eux et avec l'environnement par l'échange des entrées/sorties permettant ainsi de mettre en évidence l'aspect réactif synchrone du système étudié. L'environnement peut être représenté également par des automates en parallèles qui émettent des évènements auxquels les états du modèle hiérarchique sont sensibles.

Ce modèle présente une structure générique qui décrit le système d'une manière globale, une spécification particulière (type du système, mode du système, etc.) peut être ainsi établie à travers des entrées booléennes au modèle que nous appelons des "pré-conditions système". Cet aspect générique permet de générer une seule analyse et une seule vérification du système suivant cette description générale sans avoir besoin de tester chaque système spécifique à part, ce qui permet de gagner en temps de vérification et en mémoire. La génération du test spécifique est plutôt établie à la fin à travers l'analyse des pré-conditions système.

4 Aplatissement partiel

Une façon classique et simple d'analyser une machine hiérarchique parallèle est de l'aplatir (substituer d'une manière récursive chaque macro-état d'un FSM hiérarchique par son FSM associé, et réaliser le produit cartésien des sous graphes parallèles), puis d'appliquer, par exemple, un outil de "Model-Checking" ou de "test" sur le FSM résultant.

Considérons le modèle de la figure IV.2, qui montre des automates interagissant et communiquant entre eux. La plupart d'entre eux est de type séquentiel (les automates hiérarchiques tels que les automates 1 et 2) tandis que d'autres sont des automates parallèles tels que les automates 6 et 8. En fait, cette architecture présente en première approximation 13122 ($3 \times 6 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$) états possibles dérivés des exécutions parallèles (produit des graphes) alors qu'il existe beaucoup moins d'états réellement accessibles. Ce modèle est conçu par le langage Light Esterel décrit précédemment. Ce langage s'inspire de SyncCharts [And96] dans sa forme graphique, d'Esterel [BGGL92] dans sa forme textuelle et de Lustre [HCRP91]

dans sa forme équationnelle. Nous avons choisi d'utiliser Light Esterel en particulier dans sa forme graphique pour la représentation des différents automates du système. Le choix de ce formalisme est basé sur la puissance de ce langage à intégrer les opérateurs synchrones, à prendre en compte la notion d'évènements multiples, à garantir le déterminisme, à offrir une interprétation claire dépourvue de comportement caché, à traiter la préemption de manière rationnelle, à intégrer la simultanéité des évènements ainsi que la diffusion instantanée des signaux durant la communication.

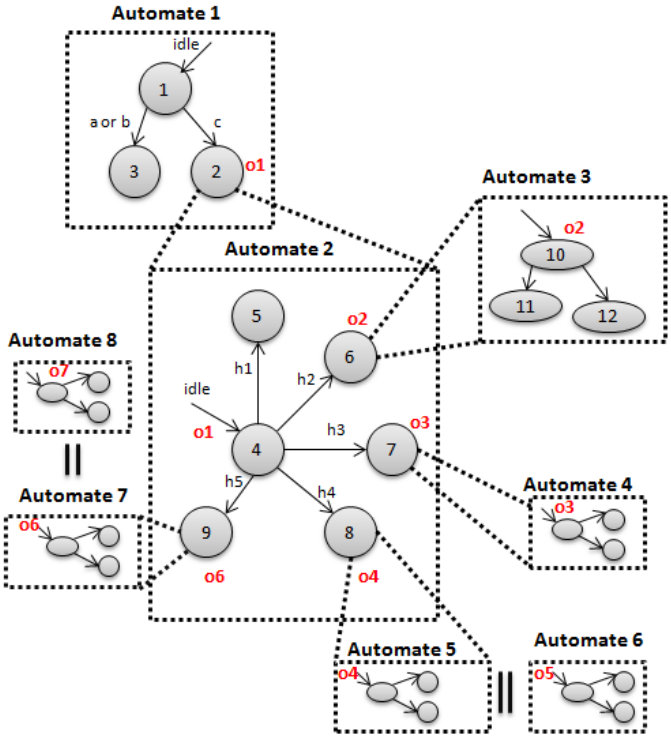


Figure IV.2 – Conception du modèle global

Une analyse classique consisterait à transformer cette structure hiérarchique décrite en Light Esterel en langage synchrone Esterel. Une telle transformation n'est pas optimisée. En effet, le compilateur Esterel n'est pas en mesure de comprendre qu'il n'y a qu'un seul état actif à la fois dans le modèle. Dans la pratique, compiler une telle structure par Esterel génère 83 registres correspondant à environ 9.6 $\times 10^{24}$ états. Ceci justifie l'intérêt de notre processus d'analyse. En effet, nous avons opté dans notre approche pour un aplatissement partiel, nous n'avons donc procédé qu'à l'aplatissement des automates hiérarchiques sans chercher à mettre à plat le parallélisme. Ainsi dans l'exemple de la figure IV.2, l'état 2 de l'automate 1 sera remplacé par l'ensemble des états {4, 5, 6, 7, 8, 9} de l'automate 2 et ainsi de suite. Ensuite, les transitions nécessaires qui en résultent sont réécrites. Les automates en parallèle agissent principalement

en tant qu'observateurs qui gèrent des informations de contrôle du modèle. Comme expliqué précédemment, aplatir les FSMs parallèles explose généralement en nombre d'états. Nous allons détailler dans la suite que l'utilisation d'un codage approprié sous forme d'équations d'états des automates permet de substituer le parallélisme par une concaténation et un tri des équations. Ainsi, il n'est plus nécessaire de les aplatir, comme nous pouvons les compiler séparément, puis les concaténer avec le modèle hiérarchique généré à la fin du processus de compilation.

La figure IV.3 illustre l'opération de liaison des états initiaux internes décrite par les lignes 3 à 9 de l'algorithme 4. Cet dernier commence par marquer le macro-état "St" dans le but de le charger dans une liste pour être supprimé plus tard. Il considère ensuite tous les sous-états associés "sub-St" (les états 3, 4, 5). Pour chaque transition de l'automate en général, si l'état aval de cette transition est le macro-état "St", alors cette transition est reliée à la transition de l'état initial de St (l'état 3). Ceci correspond au raccordement de t0, t1, t2 dans l'exemple étudié.

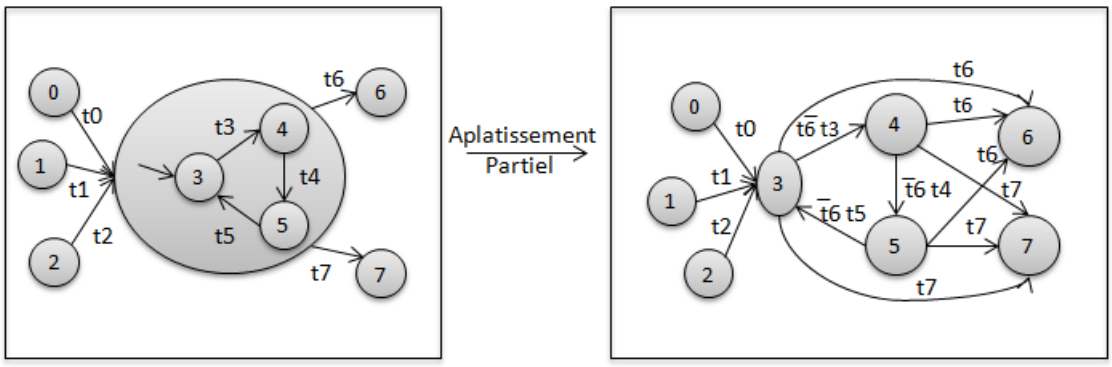


Figure IV.3 – Raccordement des états internes

L'Algorithme 4 détaille notre opération de quasi-aplatissement. On note *amont*, l'état initial de la transition et *aval* l'état final. Cet algorithme implémente trois principales opérations. Il remplace dans l'ensemble chaque macro-état avec son FSM correspondant. Il relie d'abord les états initiaux internes. Ensuite, il remplace d'une manière récursive les terminaisons normales (par référence au SyncCharts [And96], une transition de "terminaison normale" est franchie quand un état du macro-état atteint un de ses états finaux) par les transitions internes. Enfin, il interconnecte tous les états du FSM interne.

Algorithm 4 Opération d'aplatissement

```
1: St ← State ; SL ← State List of FSM ; t ← transition in FSM
2: while (SL ≠ empty) do
3:   Consider each St from SL
4:   if (St is associated to a sub-FSM) then
5:     mark the deletion of St
6:     load all sub-St from sub-FSM (particularly init-sub-St)
7:     for (all t of FSM) do
8:       if (upstream(t) == St) then
9:         upstream(t) ← init-sub-St // illustration in Fig.IV.3 (t0, t1, t2 relinking)
10:      end if
11:    end for
12:    for (all t of FSM) do
13:      if (downstream(t) == St) then
14:        if (t is a normal-term transition) then
15:          // illustration in Fig.IV.4
16:          for (all sub-St of sub-FSM) do
17:            if (sub-St is associated to a sub-sub-FSM) then
18:              create t' (sub-St, upstream(t)) // Keep recursion
19:            end if
20:            if (sub-St is final) then
21:              for (all t'' of sub-FSM) do
22:                if (upstream(t'') == sub-St) then
23:                  upstream(t'') ← upstream(t) ; merge effect(t) to effect(t'')
24:                end if
25:              end for
26:            end if
27:          end for
28:        else
29:          // weak or strong transition : illustration in Fig.IV.3
30:          // For example t3 is less prior than t6 and replaced by t6.t3 and t6
31:          for (all sub-St of sub-FSM) do
32:            if (t is a weak transition) then
33:              create t'(sub-St,upstream(t),trigger(t),weak-effect(t))
34:            else
35:              create t'(sub-St,upstream(t),trigger(t))
36:            end if
37:            for (all sub-t of sub-FSM) do
38:              turn-down the sub-t priority (or turn up t' priority)
39:            end for
40:          end for
41:        end if
42:        delete t
43:      end if
44:    end for
45:  end if
46: end while
```

La figure IV.4 illustre l'opération de raccordement d'une transition de terminaison normale. Dans le cas de la transition de terminaison normale (t5), si son état amont est un macro-état St, alors les sous états associées (les états 1,2,3,4) sont considérés. Si ces sous-états sont eux même des macro-états, une liaison est alors créée entre ces états et leur état aval. Dans le cas contraire, si ces états sont des états finaux (les états 3,4), alors pour toute transition t" de St arrivant à ces états finaux, ces derniers sont fusionnés avec l'état amont de la transition normale t, ce qui correspond à l'état 5 dans l'exemple étudié. Enfin, les sorties des états fusionnés sont reportées vers l'état résultant. St est marqué dans une liste pour être supprimé à la fin de l'algorithme.

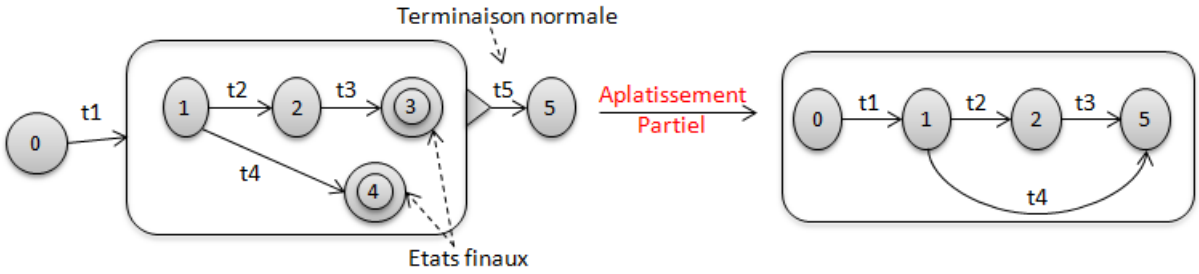


Figure IV.4 – Raccordement d'une transition normale

Par ailleurs, dans le cas d'une transition t qui n'est pas terminale de type *weak* ou *Strong*, des transitions sont créées entre tous les sous états du macro-état St et leurs états avals. Cette étape est illustrée dans la figure IV.3, où tous les états internes du macro-état St sont reliés avec leurs états amont, car t peut être interprétée comme une transition de préemption. Dans ce cas, les états (3,4,5) sont reliés avec les états 6 et 7. La priorité des transitions est ensuite gérée, les transitions de niveau supérieur sont prioritaires par rapport à celles de niveau inférieur. Ainsi, $t3$ est remplacée par $\bar{t6}t3$ pour exprimer le fait que $t6$ est prioritaire par rapport à $t3$ et ainsi de suite. A la fin de cet algorithme, tous les états marqués sont supprimés. De plus, en fonction du modèle de transition *weak*, les sorties sont transférées sur les nouvelles transitions (ce qui n'est pas le cas pour une transition de type *Strong*).

Aplatir le modèle hiérarchique de la Figure IV.2 produit une structure plate représentée dans la Figure IV.5. Comme l'activation de l'état 2 déclenche l'état 4, ces deux états sont fusionnés. De même l'état 6 est fusionné avec l'état 10, et ainsi de suite. Les automates 6 et 8 (observateurs) restent parallèles dans l'automate étendu. Ils sont assez petits et n'augmentent pas la complexité de calcul. Le modèle de la figure IV.5 ne contient maintenant que 144 (16 $\times$ 3 $\times$ 3) combinaisons d'états. Dans la pratique, compiler ce modèle suivant notre approche génère uniquement 8 registres, l'équivalent de 256 états.

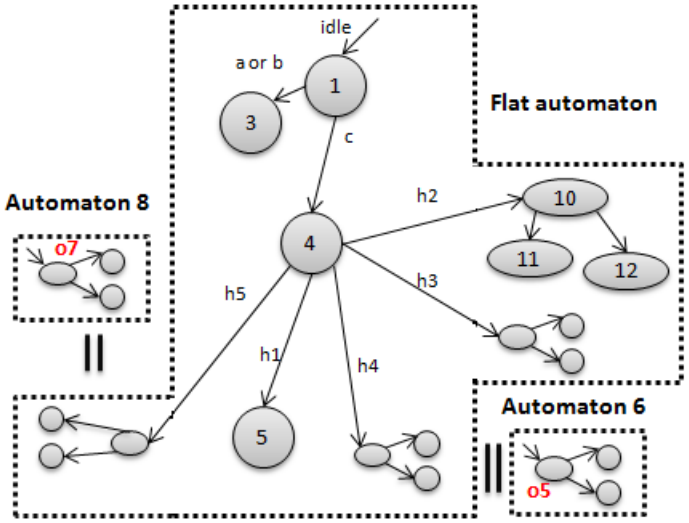


Figure IV.5 – Modèle Plat

Notons que notre technique d'aplatissement diffère considérablement de celles de [PTFV05] et [Was04]. Nous supposons qu'une transition, à la différence des diagrammes d'états en StateCharts, ne peut pas traverser différents niveaux hiérarchiques. Plusieurs opérations sont ainsi exécutées localement, et non sur le système global. Ceci offre un algorithme plus simple et une compilation plus rapide. En résumé, nous avons intégré les hypothèses suivantes dans notre algorithme :

-Terminaison normale : La figure IV.4 montre un exemple d'une terminaison normale exécutée quand un état interne final est atteint. Ceci permet une unique interprétation possible et facilite la génération de code.

-Préemption forte : Contrairement à la préemption faible, les sorties internes de l'état préempté sont perdues au cours de la transition préemptive.

5 Processus de compilation

Nous procédons dans notre approche à une compilation symbolique du modèle en une machine de Mealy, implicitement représentée par un ensemble d'équations booléennes (circuit de portes logiques et de registres représentant l'état du système). Compiler un automate explicite à un automate implicite est un processus bien connu dans la conception matérielle. Beaucoup de travaux considèrent la représentation *one-hot* [CI10] qui utilise un registre par état, alors que notre compilation ne nécessite que log_2(nombre d'états) registres. Avec ce codage des FSMs, nous effectuons la mise en parallèle par la concaténation et le tri des équations obtenues pour

chaque FSM compilé séparément. En conséquence, une réduction importante est notée de la taille du système testé.

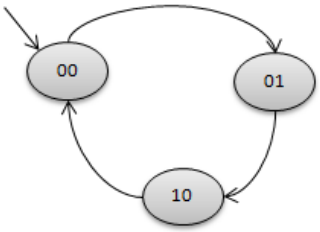
Algorithm 5 Processus de Compilation

1: R ← Vector of states
2: R-I ← Initial vector of states (initial value of registers)
3: Next-State ← Vector of transitions
4: N ← Size of R and Next-State
5: f,f' ← Vectors of Boolean functions
6: N ← $\log_2 (Statesnumber - 1) + 1$
7: Define N registers encapsulated in R.
8: **for** (i=0 to $N - 1$) **do**
9: $BDD_{f(i)}$ ← BDD-0 // BDD initialisation
10: $BDD_{f'(i)}$ ← BDD-0
11: **end for**
12: **for** (j=0 to $Noutputs - 1$) **do**
13: $Output_{O(j)}$ ← BDD-0
14: **end for**
15: R-I ← binary coding of initial state
16: **for** (transition t_{kl}=k to l) **do**
17: V_k ← Binary coding of k
18: V_l ← Binary coding of l
19: BDD_{cond} ← cond (t_{kl}) ;
20: **for** (i=0 to $N - 1$) **do**
21: **if** V_k(i)==1 **then**
22: $BDD_{cond} \leftarrow BDD_{and}$(R(i), BDD_{cond}) // BDD_{cond} : t_{kl} BDD characteristics
23: **else**
24: $BDD_{cond} \leftarrow BDD_{and}(BDD_{not}$ (R(i)), $BDD_{cond})$
25: **end if**
26: **if** (Vl(i)==1) **then**
27: $BDD_{f(i)} \leftarrow BDD_{or}(BDD_{f(i)}$, BDD_{cond})// $BDD_{f(i)}$: set of register
28: **else**
29: $BDD_{f'(i)} \leftarrow BDD_{or}(BDD_{f'(i)}$, $BDD_{cond}))$// $BDD_{f'(i)}$: reset of register
30: **end if**
31: $output_O$(output(t_{kl}))← BDD_{or} ($output_O$(output(t_{kl})), $BDDcond$)
32: **end for**
33: **end for**
34: **for** (i=0 to $N - 1$) **do**
35: BDD-NextState(i)← $BDD_{respect}$ (BDD_f, BDD'_f)
36: // respect : every BDD_h such us $BDD_f \rightarrow BDD_h$ AND $BDD_h \rightarrow not(BDD'_f)$
37: **end for**

L'algorithme 5 décrit en détails ce processus de compilation. Il compte le nombre d'états de l'automate et en déduit la taille du vecteur d'états. De là, il développe pour une variable d'état donnée la fonction de l'état suivant. Le vecteur généré est caractérisé par un ensemble d'expressions booléennes et représenté par un ensemble de BDDs.

Considérons par exemple un automate à 16 états. Le vecteur caractérisant l'état suivant est construit ainsi sous forme de 4 ($log_2(16)$) expressions calculées à partir des entrées données et de l'état courant. Pour chaque transition de l'état "k" à l'état "l", deux types de vecteurs codés sur n (n=4 bits dans cet exemple) bits sont créées : un vecteur V_k caractérisant la fonction caractéristique de la transition BDD_{cond}, et un vecteur V_l caractérisant la fonction de l'état futur $BDD-NextState$. Si $V_k(i)$ est valuée à 1, alors la variable d'état Y_i est considérée positivement. Dans le cas contraire, la variable Y_i est inversée (lignes 20-25). Le BDD caractéristique de la condition de transition est ainsi déduit par la combinaison des "Y_i" et de la condition sur la transition "*cond*". Par exemple, $BDD_{cond} = Y_0 \times \bar{Y_1} \times \bar{Y_2} \times Y_3 \times cond$ pour le cas de $V_k = (1, 0, 0, 1)$.

Les lignes (26-31) montrent la construction de la fonction d'état futur $BDD-NextState = BDD_{f(i)} \times notBDD_{f'(i)}$. $BDD_{f(i)}$ caractérise toutes les transitions qui forcent Y_i^+ à 1 (mettre les registres à 1). A l'inverse, $BDD_{f'(i)}$ caractérise toutes les transitions qui forcent Y_i^+ à 0 (réinitialiser les registres à 0). Toute fonction vérifiant f et not(f') présente donc une solution possible. Le parcours de tous les états du système n'est pas nécessaire dans ce cas. La figure IV.6 montre un exemple où il est possible de trouver une fonction compatible "$BDD1_{respect} = \bar{y_1}.y_0 + not(\bar{y_1}.\bar{y_0} + y_1.\bar{y_0})$" pour Y_1^+ et "$BDD0_{respect} = \bar{y_1}.\bar{y_0} + not(y_1.\bar{y_0} + \bar{y_1}.y_0)$" pour Y_0^+ même si l'état du système dans le cas où "$y_1y_0 = 11$" n'est pas spécifié. Le BDD d'état futur est donc exprimé par les deux $BDD_{respect}$ qui cherchent les expressions les plus simples pour vérifier d'une part f_0 et not(f_0') et d'autre part f_1 et not(f_1').



y_1	y_0	Y_1^+	Y_0^+
0	0	0	1
0	1	1	0
1	0	0	0
1	1	X	X

Figure IV.6 – Fonction d'état futur

6 SupLDD : Manipulation numérique des données

En plus des fonctions booléennes, notre approche permet la manipulation symbolique des données numériques du système. Ceci a pour avantage de donner une représentation plus réelle et expressive du système étudié. Nous avons développé à cette fin une nouvelle couche supérieure que nous appelons "SupLDD" au dessus de la bibliothèque des LDDs décrite dans le chapitre précédent. Nous nous sommes basés sur les fonctions de base des LDDs pour générer des fonctions pratiques, simples et adaptées à nos besoins pour la manipulation des variables numériques. Nous présentons essentiellement des fonctions pour manipuler des inéquations de type $\{\sum a_i x_i \leqslant c\}$; $\{\sum a_i x_i < c\}$; $\{\sum a_i x_i \geqslant c\}$; $\{\sum a_i x_i > c\}$; Où $\{a_i, x_i, c \in Z\}$. Nous avons intégré également différentes opérations sur les inéquations tels que :

- L'intersection de deux inéquations : cette opération correspond parfaitement à l'intersection dans l'espace Z des deux sous-espaces associés à ces inéquations.
- L'union de deux inéquations : de même, cette opération correspond parfaitement à l'union dans l'espace Z des deux sous-espaces associés à ces inéquations.
- Tout l'espace Z est déduit par exemple par l'union des deux inéquations $\{x \leq a \cup x > a\}$.
- L'ensemble vide est de même déduit par l'intersection des inéquations $\{x \leq a \cap x > a\}$.
- L'opérateur d'égalité $\{\sum a_i x_i = c\}$ est établit par l'intersection des deux inégalités $\{\sum a_i x_i \leqslant c\}$ et $\{\sum a_i x_i \geqslant c\}$.
- L'opérateur de résolution permet de simplifier des expressions à l'aide du principe de la quantification QELIM expliquée précédemment. Par exemple, l'expression $\{x - y \leqslant 3 \bigwedge x - t \geqslant 8 \bigwedge y - z \leqslant 6\}$ devient après résolution $\{x - z \leqslant 9 \bigwedge x - t \geqslant 8\}$. Cet opérateur intègre également la règle d'implication faible III.6 où l'expression $\{x \leqslant 3 \bigvee x \leqslant 8\}$ devient $\{x \leqslant 8\}$ après simplification, et la règle d'implication forte III.5 où l'expression $\{x \leqslant 3 \bigwedge x \leqslant 8\}$ devient $\{x \leqslant 3\}$.
- L'opérateur de réduction "Resolve" permet de résoudre une expression A par rapport à une expression B. Autrement dit, si A implique B alors la résolution de A par rapport à B donne la projection de A quand B est vrai. Par exemple, la résolution de l'ensemble A $\{x - y \leqslant 5 \bigwedge z \geqslant 2 \bigwedge z - t \leqslant 2\}$ par rapport à B $\{x - y \leqslant 7\}$ donne l'ensemble $\{z \geqslant 2 \bigwedge z - t \leqslant 2\}$.

L'ensemble de ces fonctions est très utile dans la vérification des modèles du système. Par exemple, elles permettent d'assurer le déterminisme d'un automate : l'intersection de toutes les transitions sortantes d'un état doit être égale à l'élément vide. Elles permettent également de détecter les séquences impossibles : une séquence ne peut pas exister si l'intersection de ses contraintes tout au long du chemin étudié est nulle.

7 GSS : Générateur Symbolique de séquences

Contrairement à l'exécution concrète qui ne suit qu'un seul des chemins possibles, nous procédons à une exécution symbolique [BEL75] dans le but d'explorer tous les chemins possibles du système étudié. Le principe est de manipuler les formules logiques liant les variables entre elles plutôt que de mettre directement à jour les variables en mémoire, dans le cas d'une exécution concrète. Le modèle de la figure IV.7 présente un ensemble des séquences possibles décrivant le comportement d'un système donné.

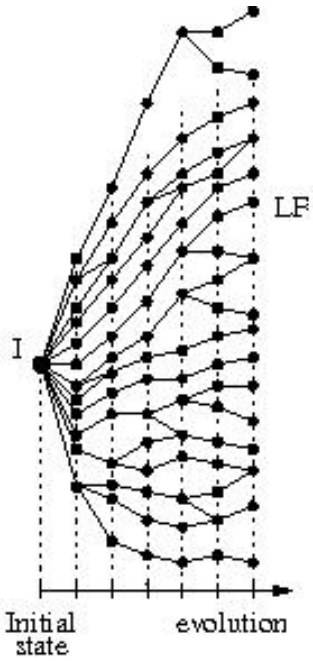


Figure IV.7 – Génération Classique de séquences

Il s'agit d'une représentation classique des évolutions dynamiques du système. Cette représentation montre un très grand arbre voire même un arbre infini. Ainsi, l'exploration de toutes les exécutions possibles du programme n'est pas du tout réaliste. Il s'agirait d'imaginer toutes les combinaisons possibles des commandes du système, ce qui est une tâche quasiment impossible. Nous allons montrer dans le chapitre suivant les limites de cette approche classique dans le cas du test de gros systèmes.

En considérant la représentation du système par une séquence de commandes exécutées de façon itérative, l'arbre précédent des séquences se réduit à une répétition d'un même motif du sous-espace comme représenté par la figure IV.8. L'idée est de restreindre l'espace des états à étudier et ne se limiter qu'au sous espace significatif. Ce dernier représente une commande spécifique du système qui peut être répétée à travers les séquences possibles générées.

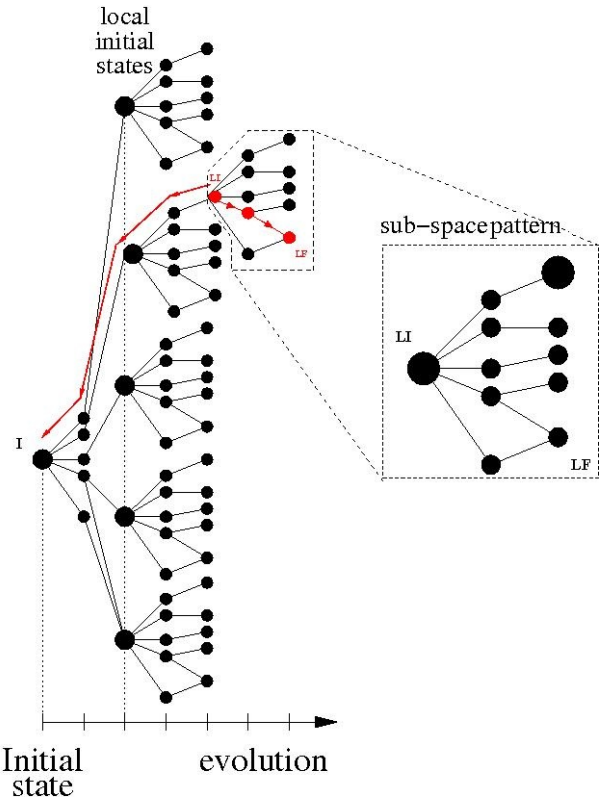


Figure IV.8 – Représentation GSS du modèle

Chaque état dans ce sous-espace est caractérisé par 3 types de variables : les valeurs symboliques des variables du programme, les conditions du chemin et les paramètres de commandes (le paramètre suivant à exécuter). Les conditions de chemin représentent les contraintes qui doivent être satisfaites par les valeurs symboliques pour faire progresser l'exécution du chemin actuel. Elles définissent les conditions préalables pour suivre un chemin avec succès. Nous présentons ces pré-conditions sous deux formes distinctes :

-Des pré-conditions globales : ce sont des variables booléennes de définition du contexte d'exécution d'une commande. Elles apparaissent en tant que sortie d'une autre commande quand celle-ci est correctement exécutée. Elles se présentent également sous forme de contraintes d'entrée pour les commandes suivantes. Elles permettent dans ce cas de caractériser la liste des commandes qui doivent êtres exécutées auparavant ;

-Des pré-conditions locales : ce sont des variables numériques pour représenter les conditions sur les paramètres d'une commande sous forme de contraintes numériques. Ces variables sont manipulées par les fonctions de SupLDD décrites dans la section 6. Elles sont donc représentées sous forme de $\{\sum a_i x_i \leqslant c\}$; $\{\sum a_i x_i < c\}$; $\{\sum a_i x_i \geqslant c\}$; $\{\sum a_i x_i > c\}$; Où x_i représente les différents paramètres des commandes.

En somme, l'opération de GSS s'établit dans le sous espace significatif représentant une commande du système. Toutes les commandes du système sont exécutées mais, une seule commande est testée à la fois. Un premier objectif est d'identifier les séquences impossibles à exécuter. Ceci est réalisé par une analyse des pré-conditions locales en appliquant "SupLDD" lors de la génération de séquences. A cet objectif, nous avons appliqué l'opération de conjonction sur toutes les pré-conditions locales d'un même chemin. Si le résultat de cette conjonction est nulle, cette séquence est alors impossible et doit être corrigée dans le système.

Le deuxième objectif de notre test est de vérifier le contexte d'exécution de chaque commande. Il s'agit en particulier d'identifier et de vérifier que toutes les pré-conditions globales extraites sont satisfaites. Si le contexte est vérifié, la séquence générée est jugée bonne. Cette opération de vérification est assurée par l'opération de marche en arrière "Backtrack " qui est détaillée dans la suite.

L'algorithme 6 montre en détail l'opération de parcours symbolique exécutée dans chaque sous-espace. Ce parcours permet de générer toutes les séquences possibles dans une commande et d'extraire ses pré-conditions globales. Cette opération est assez simple car elle s'appuie sur la souplesse du modèle conçu et compilé à travers l'approche synchrone.

En effet, nous avons appliqué des analyses symboliques (booléennes par les BDDs et entières par les LDDs) à partir de l'état initial local (l'état initial d'une commande) jusqu'à l'état local final du sous-espace spécifié. Pour chaque combinaison de registres, les manipulations en BDD et en LDD permettent de déterminer et de caractériser l'état futur et de remettre à jour les variables d'état. Les pré-conditions nécessaires à cette transition sont ainsi identifiées. Si ces pré-conditions sont globales, elles sont alors insérées dans la liste GPLIST des pré-conditions globales pour être affichées plus tard dans le contexte de la séquence générée. Par ailleurs, si ces pré-conditions sont locales, elles sont alors mises en conjonction avec les précédentes. Si le résultat de cette conjonction est nul, la séquence générée est marquée impossible et doit être corrigée dans le système. Les sorties sont également calculées et stockées dans une pile. Enfin, la séquence est complétée par le nouvel état établi. Une fois toutes les pré-conditions globales identifiées, une prochaine étape est d'établir une marche en arrière "Backtrack" jusqu'à trouver la séquence initiale qui remplit ces pré-conditions.

Algorithm 6 Processus du GSS

```
1: Seq ← sequence
2: BDS ← BDD State
3: BDA ← BDD awaited
4: BDAC ← BDD awaited context
5: OLIST ← Outputs list
6: GPLIST ← Global Precondition list
7: LPLIST ← Local Precondition list
8: BDS ← Initial state
9: BDAC ← 0
10: OLIST ← empty
11: GPLIST ← empty
12: LPLIST ← All the space
13: Push (BDS, OLIST, BDAC)
14: while (stack is not empty) do
15:    Pull (BDS, OLIST, BDAC)
16:    list(BDA)← Compute the BDD awaited expressions list(BDS)
17:    for (i=0 to |list(BDA)|) do
18:       Input ← extract(BDA)
19:       if (Input is a global precondition) then
20:          GPLIST ← Push(GPLIST, Input)
21:       else
22:          if (Input is a local precondition) then
23:             LPLIST ← SupLDD-AND(LPLIST, Input)
24:             if (GPLIST is null) then
25:                Display ( Impossible Sequence !)
26:                Break
27:             end if
28:          end if
29:       end if
30:       NextBDS ← Compute future(BDS, BDA)
31:       OLIST ← Compute output (BDS, BDA)
32:       New-seq ← seq BDA | BDA
33:       if (New New-seq size < maximum diameter) then
34:          Push (NextBDS, OLIST, BDAC)
35:       else
36:          Display (GPLIST)
37:          Display (New-seq)
38:       end if
39:    end for
40: end while
```

8 Opération de marche en arrière "Backtrack"

L'opération de Backtrack est assurée par le processus de compilation qui garde suffisamment d'informations pour retrouver des états antérieurs qui se rapprochent de l'état initial (les rétro-chemins). Ce processus est détaillé en deux principales phases dans l'Algorithme 7. La première phase (lignes 11-25) consiste à étiqueter tous les nœuds non encore étudiés de l'espace d'états. A partir de l'état initial ($e \leftarrow 0$), tous les états successeurs sont étiquetés par ($e \leftarrow e + 1$). Si un état est déjà étiqueté, son indice n'est pas incrémenté. Cette opération est effectuée pour tout les états jusqu'à couvrir tout l'espace d'états. La deuxième phase (lignes 26-31) consiste à identifier les bons états antérieurs. Pour chaque état St, parmi tous ses prédécesseurs, celui qui porte l'étiquette la plus petite fait partie du chemin le plus court qui retourne à l'état initial : c'est un résultat important de la théorie des graphes [Joh73]. En d'autres termes, les états antérieurs convergent toujours vers le même état initial racine, ce qui rend possible l'exécution de l'opération de marche en arrière.

Considérons l'exemple de la figure IV.8. A partir de l'état initial local "IL" (l'état initial d'une commande), le générateur symbolique de séquences génère tous les chemins possibles du sous-espace testé pour atteindre les états locaux finaux par des analyses en BDDs et en SupLDDs. En considérant "LF" (état final local) un état final critique du sous-espace testé, le Backtrack est effectué de l'état "LF" jusqu'à couvrir la rétro-séquence qui satisfait l'ensemble des pré-conditions globales extraites pour arriver au point racine I. En supposant l'état "I" le résultat final du processus de retour en arrière, la séquence de "I" à "LF" donnera un exemple d'un bon jeu de test. Cette démarche est nettement plus efficace que la démarche classique. Rappelons que cette dernière établit un jeu de test de "I" à "LF" qui ne peut être réalisé que par la génération de tous les chemins de l'arbre. Un tel test devient impossible si le nombre de pas pour atteindre "LF" est considérablement élevé.

L'opération de Backtrack qui utilise les rétro-chemins s'établit en deux principales étapes. Il s'agit d'abord de vérifier le contexte et d'identifier la liste des commandes qui doivent être exécutées avant la commande testée. Ce processus appelé "Backtrack global" se base sur les pré-conditions booléennes de contrôle pour établir la liste des commandes précédentes. Cette liste étant établie, une deuxième étape consiste à établir le chemin final liant toutes les commandes à exécuter jusqu'à l'état final spécifié. Ce "Backtrack" appelé local est appliqué à l'aide des pré-conditions numériques locales de chaque commande de la liste.

Algorithm 7 Recherche du prédécesseur

```
St ← State
LSt ← List of States
LS ← List of Successors
LabS ← State Label
IS ← Initial State
S ← State
LabS(IS) ← 1
LSt ← IS
LP ← List of Predecessors
SMin ← Minimum Lab State
// Expansion
while (LSt != 0) do
   for (all St of LSt) do
      LS ← Get-Successors(St)
      if (LS != 0) then
         for (all S of LS) do
            if (!LabS(S)) then
               LabS(S) ← LabS(St)+1
            end if
         end for
         NLSt ← Push(LS)
      end if
   end for
   LSt ← NLSt
end while
// Recherche des Prédécesseurs
for (all St) do
   LP ← Get-Predecessor(St)
   SMin ← LabS(first(LP))
   StMin ← first(LP)
   for (all St' in LP) do
      if (LabS(St') < SMin) then
         SMin ← LabS(St')
         StMin ← St'
      end if
      Memorise-Backtrack(St,StMin)
   end for
end for
```

L'algorithme 8 décrit en détail le processus de Backtrack global. A partir d'un état final spécifique du sous espace étudié, il récupère les pré-conditions globales booléennes (BBP) cor-

respondantes. Pour chaque pré-condition, il cherche dans la table des actions globales l'action qui l'émet. Ensuite, il récupère l'état S déclencheur de cette action. Cet état est ainsi inséré dans la liste des états précédents BKL. Cette opération est répétée pour tous les états précédents trouvés jusqu'à atteindre l'état initial ayant zéro pré-conditions globales.

Algorithm 8 Processus du Backtrack Global

```
1: BKS ← Backtrack States
2: BIS ← Backtrack Initial States
3: BFS ← Backtrack Final States
4: BBP ← Backtrack Boolean Preconditions
5: BKL ← Backtrack List
6: P ← Priority
7: PL ← Priority List
8: BKS ← BIS
9: BFS ← BKS
10: while (BKS not null) do
11:    for (all BKS) do
12:       for (all BBP) do
13:          for (all actions) do
14:             if (action == BBP) then
15:                S ← Find-State(action)
16:                index ← exists (S,BKS)
17:                if (index != 0) then
18:                   BKL ← Push-Back(S)
19:                   P ← PL-size
20:                   PL ← Push-Back(P)
21:                else
22:                   PL ← PL[index]+1
23:                end if
24:             end if
25:          end for
26:       end for
27:    end for
28:    BKS ← BKL
29:    BFS ← Push-Back(BKS)
30: end while
31: for (all BFS) do
32:    PS ← Find-Prior-State(PL,BFS)
33:    Local Backtrack(PS)
34: end for
```

Etant donné que les différentes commandes peuvent partager les mêmes pré-conditions, les états trouvés peuvent être répétés dans la liste BKS. A cet égard, nous avons alloué dans une liste PL une priorité P à chaque état trouvé. Un état d'une priorité n doit précéder un état d'une priorité n+1. Les lignes (17-23) montrent le principe d'allocation de ces priorités. Si un état existe déjà dans la liste BKS, alors il aura la priorité suivante sinon sa priorité est augmentée d'une unité. Enfin, après avoir établi la liste BFS de tous les états finaux des commandes qui doivent être exécutées au préalable de l'état spécifié, cet algorithme considère l'état de plus faible priorité et établit un Backtrack local de cet état.

Algorithm 9 Processus du Backtrack Local

```
1: FS ← Final State
2: BL ← Backtrack SupLDD
3: TL ← Total SupLDD
4: LL ← Local SupLDD
5: PT ← Pre Transitions(FS)
6: BL ← Get-Global-SupLDD(FS)
7: while (BL != 0) do
8:    TB ← PT.Begin
9:    if (amont(TB) != aval(TB)) then
10:      TL ← Get-Global-SupLDD(TB)
11:      if (TL ⩾ BL) then
12:         LL ← Get-Local-SupLDD(TB)
13:         if (LL != 0) then
14:            BL ← Resolve (BL, LL)
15:         end if
16:         amont ← Get-amont-state(TB)
17:         PT ← Pre Transitions(amont)
18:      else
19:         This is a boolean transition !
20:         TB ← TB+1
21:      end if
22:   else
23:      This is a loop !
24:      TB ← TB+1
25:   end if
26: end while
```

L'opération de Backtrack local est décrite dans l'algorithme 9. Elle repose principalement sur les pré-conditions numériques locales dans chaque commande. Partant de l'état final spécifique FS, cet algorithme récupère la pré-condition locale LL de sa liste et la pré-condition totale TL

de cet état. La pré-condition totale TL est la conjonction de toutes les pré-conditions locales LL traversées dans le chemin étudié. Une pré-condition du "Backtrack" BL est définie par la suite et initialisée par TL. Il s'agit donc de chercher un chemin qui satisfait BL. En partant de la première transition entrante PT à FS, si $\{TL \geqslant BL\}$ alors, cette transition est testée. Si son LL est non nul, alors PT est une transition numérique. Dans ce cas, le BL est résolu par rapport au LL à travers la fonction de réduction "Resolve" décrite dans la section 6. Le LL étant déjà satisfait, il est réduit du BL. Il s'agit de passer ensuite à l'état amont de la transition testée et de tester ses transitions entrantes (nouvelle transition PT). Dans le cas où le TL d'une transition PT ne satisfait pas le BL $\{TL > BL\}$, il s'agit de passer à une autre transition entrante. Cette opération est répétée jusqu'à satisfaire toutes les contraintes du Backtrack, autrement dit quand le BL est égal à zéro. L'objectif est donc de trouver un chemin possible de l'état initial à un état final de chaque commande exécutée. Enfin, tous les chemins trouvés sont affichés dans l'ordre de priorité de chaque commande jusqu'à obtenir un cas de test possible pour aboutir à un état critique donné du système.

9 Conclusion

Nous avons proposé dans ce chapitre notre approche pour tester automatiquement des systèmes réactifs. Notre travail est adéquat pour le test des systèmes exécutant des commandes itératives. Il permet de gérer de grands modèles de taille industrielle, où le risque d'explosion combinatoire de l'espace d'états est important. Ceci a été assuré essentiellement en se focalisant sur le sous-espace d'état significatif représentant une commande du système au lieu de considérer tout l'espace. Tout le calcul est donc établi en particulier dans ce sous-espace. Le test du modèle global est plutôt établi par la manipulation des pré-conditions locales et globales produites de l'analyse du sous-espace. En outre, la manipulation des pré-conditions "système" permet de plus le test de différents types ou spécifications du système global. Le chapitre suivant présente les détails de l'implémentation de notre approche et les résultats expérimentaux de son illustration à travers un cas de test industriel.

Chapitre V

Mise en Œuvre et expérimentation

1 Introduction

Dans ce chapitre, nous présentons une implémentation des concepts que nous avons présentés dans le chapitre précédent. A cette fin, nous avons développé un nouvel outil de test. Nous avons évalué la performance et l'efficacité de notre outil pour la vérification d'une carte à puce sans contact, de taille industrielle, destinée pour le service de transport. Nous avons comparé notre démarche de test par rapport à celle adoptée dans l'entreprise qui a fabriqué la carte. Nous avons également comparé les résultats obtenus par notre outil GAJE par rapport à ceux obtenus avec une approche classique.

2 Description de l'outil GAJE

Nous avons concrétisé notre approche dans un outil appelé GAJE, Générateur Automatique de Jeux de Tests. Cet outil GAJE peut être utilisé d'une manière efficace pour le test de différents types de systèmes réactifs. Nous avons mené dans cet ouvrage deux implémentations différentes de GAJE. La première version permet l'analyse des automates implicites, tandis que la deuxième version permet plutôt l'analyse directe des automates explicites. Les figures V.1 et V.2 montrent l'ensemble des outils implémentés et utilisés dans les deux versions de GAJE qui matérialisent les notions présentées dans cet ouvrage.

La figure V.1 montre une première vision de GAJE. Elle présente tous les outils implémentés pour décrire notre approche. Cette première version est opérationnelle pour le test de tous les types de systèmes, notamment les gros systèmes. La figure V.2 montre une version allégée de GAJE adaptée au cas d'applications décrit dans la section 3. Cette deuxième version est plus rapide que la première, mais potentiellement plus coûteuse en mémoire. Elle analyse directement l'automate explicite sans le compiler et manipule directement les variables numériques sans

avoir besoin de faire la correspondance implémentée dans la première version de GAJE entre les variables booléennes et numériques. En effet, les outils utilisés et déjà existants dans la première version ne manipulent que des variables booléennes.

2.1 Outils implémentés dans la première version de GAJE

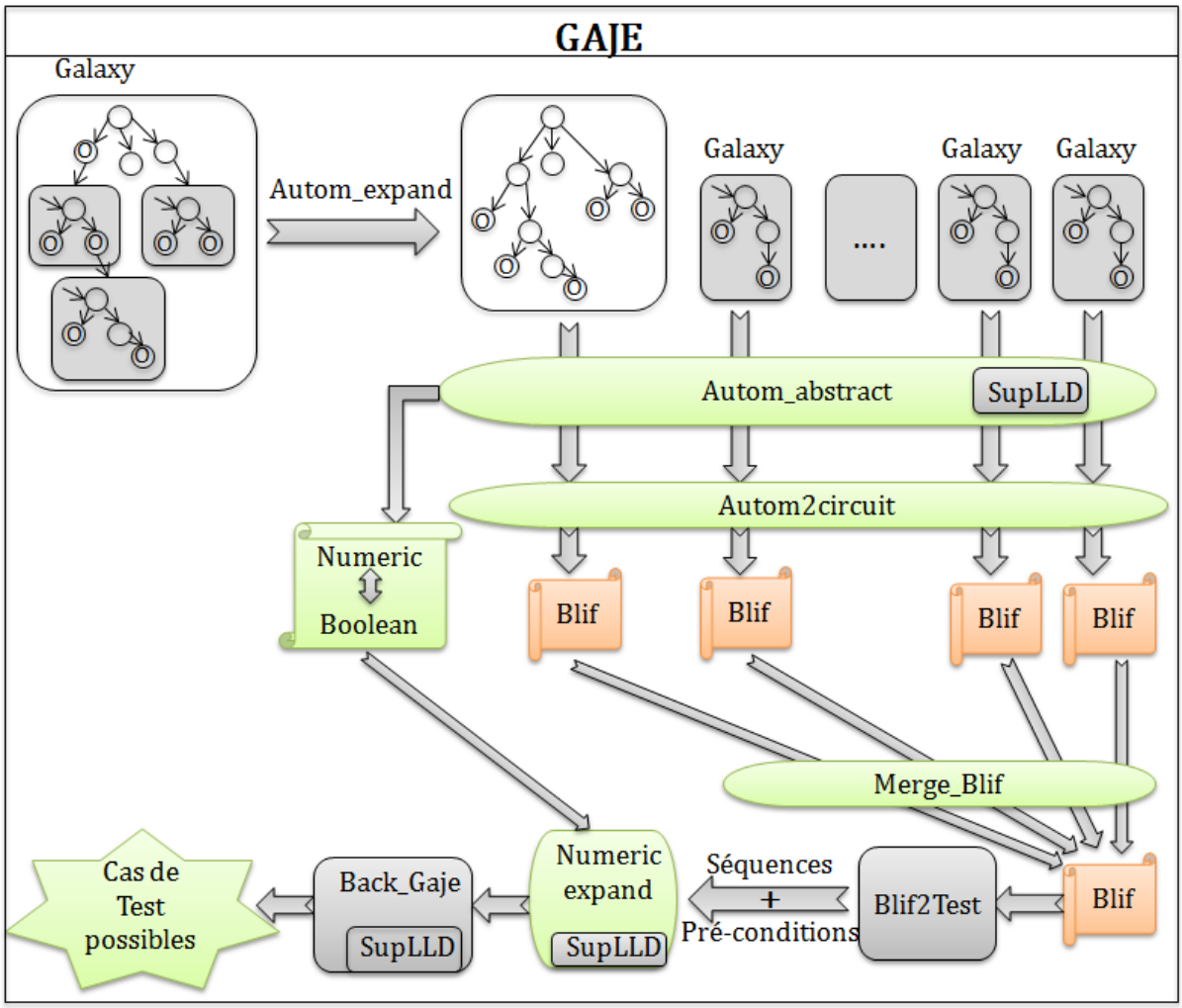


Figure V.1 – Chaîne des outils de la première version de GAJE

2.1.1 Galaxy

Galaxy est un éditeur d'automates d'états finis qui utilise la librairie graphique fltk [1]. Cet éditeur a été développé par M.Gaffe [Gaf] dans le cadre d'un projet de recherche autour des

1. http ://www.fltk.org/index.php

machines à états finis et des langages synchrones. Galaxy regroupe quatre modes d'éditions distincts : C'est en premier lieu un éditeur d'automates simples dans le mode "basic". Il permet aussi l'édition des automates parallèles en mode "parallel automaton". Il permet également l'édition des automates hiérarchiques dans le mode "Light Esterel". Enfin, il permet de gérer des SyncCharts en mode "SyncCharts" d'où son nom [2]. Nous avons utilisé cet éditeur en mode "Light Esterel" qui est le plus adapté pour représenter le comportement du système étudié.

2.1.2 Autom-expand

Nous avons contribué au développement de l'outil Autom-expand qui implémente le principe de quasi-aplatissement décrit par l'Algorithme 4. Cet outil charge un fichier ".gal" qui représente le modèle hiérarchique généré par Galaxy. L'exécution de Autom-expand génère un modèle aplati sous forme d'un fichier ".gal".

2.1.3 SupLDD

SupLDD est la nouvelle bibliothèque décrite dans la section IV. 6 que nous avons développé au dessus de la bibliothèque LDD afin d'exprimer et de manipuler les pré-conditions numériques du système sous forme de contraintes numériques. A cette fin, cette bibliothèque implémente un ensemble de fonctions tels que : *SupLDD-Create-inequation*, *SupLDD-And*, *SupLDD-Or*, *SupLDD-All*, *SupLDD-Null*, *SupLDD-Reduce*, *SupLDD-Resolve*, etc.

2.1.4 Autom-abstract & Numeric-expand

Nous avons développé ces outils dans le but de fournir un filtre indispensable aux outils qui ne manipulent que des variables booléennes comme Autom2circuit. En effet, Autom-abstract transforme chaque contrainte sur les entiers en une variable booléenne pour toutes les transitions de l'automate étudié. Il gère également tous les aspects implication et exclusion de contraintes. Nous avons appliqué cet outil sur le fichier ".gal" en sortie de Autom-expand afin de générer un autre fichier ".gal" compréhensible pour Autom2circuit. Un deuxième fichier est également généré à la sortie de Autom-abstract qui retient la correspondance entre les variables numériques données et les variables booléennes générées. Ce fichier permet à Numeric-expand de récupérer les données numériques abstraites par Autom-Abstract.

2. En SyncCharts, un ensemble d'états s'appelle une constellation

2.1.5 Autom2circuit

Autom2circuit est l'outil qui implémente le principe de compilation décrit par l'algorithme 5. Il représente une version allégée et simplifiée du compilateur CLEM décrit dans la section II. 4.2. Il a été développé [Gaf] à l'origine dans le but de synthétiser des automates explicites en une machine de Mealy ou Moore implicite. Il prend en entrée un fichier galaxy ".gal" et génère de multiples formats en sortie suivant le besoin (lustre, esterel, blif, vhdl, C, latex, etc). Nous utilisons cet outil dans le but de compiler la structure aplatie en une machine de Mealy implicite. Les automates de contrôle en parallèle peuvent être ainsi compilés séparément par Autom2circuit. Le résultat de cette compilation génère un fichier Blif Berkeley Logic Interchange Format [Uni98] en sortie de chaque compilation. Un fichier Blif est un format compact et standard pour exprimer une Netlist. Il est bien adapté pour représenter des systèmes d'équations booléennes.

2.1.6 Merge-Blif

Merge-Blif [Gaf] est un outil de concaténation de fichiers au format "Blif" et de réorganisation de l'ordre d'évaluation des équations booléennes. Nous utilisons cet outil afin de concaténer le résultat de compilation de l'automate aplati et des automates en parallèles. Le fichier "Blif" résultant représente alors le produit synchrone de tous les automates en parallèles.

2.1.7 Blif2Test

Blif2Test est l'outil que nous avons développé pour mettre en œuvre l'opération de GSS détaillée dans l'algorithme 6 du chapitre IV sur les variables booléennes. Cet outil charge le fichier Blif. Il génère à partir des calculs en BDDs les différentes séquences possibles dans le sous espace étudié et extrait les pré-conditions booléennes identifiées.

2.1.8 BackGaje

BackGaje est l'outil que nous avons développé pour mettre en œuvre l'opération de Backtrack. Cet outil utilise dans cette version le résultat de Autom-abstract qui permet d'extraire l'ensemble des pré-conditions globales booléennes et des pré-conditions locales numériques à partir du fichier de correspondance généré auparavant. BackGaje permet d'établir le Backtrack global détaillé dans l'algorithme 8 à partir des pré-conditions globales. Il permet ensuite d'établir l'opération de Backtrack locale détaillée dans l'algorithme 9 en exploitant la bibliothèque SupLDD et l'algorithme 7 de recherche des prédécesseurs.

2.2 Outils implémentés dans la deuxième version de GAJE

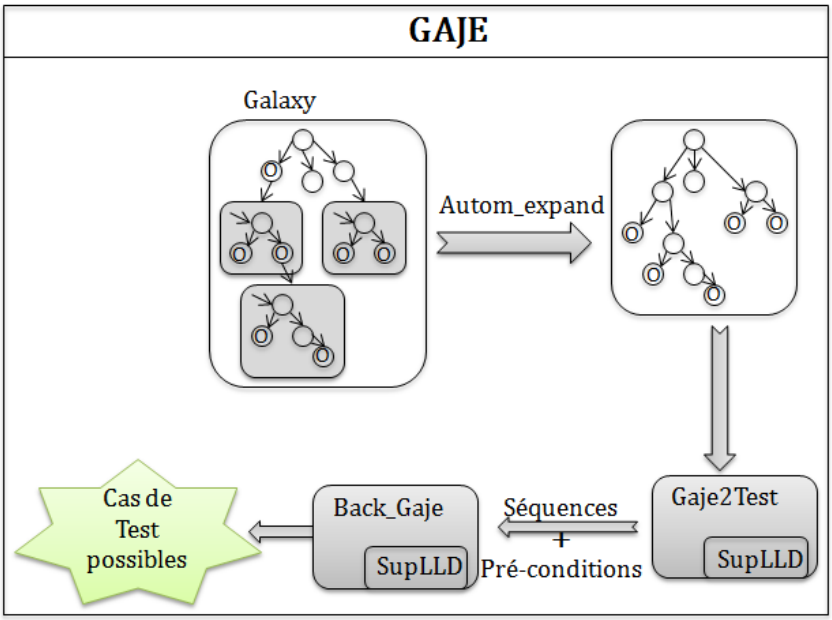


Figure V.2 – Chaîne des outils de la deuxième version de GAJE

2.2.1 Galaxy & Autom-expand

Nous avons utilisé également dans cette version les outils galaxy et Autom-expand décrits dans la section 2.1. Cependant, nous générons qu'un seul automate hiérarchique du système étudié. Les automates en parallèles, générés dans la première version, sont remplacés par des pré-conditions manipulées par la bibliothèque SupLDD.

2.2.2 Gaje2Test & BackGaje

A la différence de Blif2test qui opère sur les automates implicites, Gaje2Test permet l'analyse directe des automates explicites. Cet outil que nous avons également développé met en œuvre l'opération de GSS sur les données booléennes et numériques. Cet outil charge le fichier galaxy et utilise la bibliothèque SupLDD afin de générer les différentes séquences possibles dans le sous espace étudié et extrait les pré-conditions booléennes et numériques identifiées. Ces pré-conditions sont utilisées ensuite par BackGaje pour établir l'opération de Backtrack et générer les différents cas de tests possibles.

Dans la suite, nous présentons l'application de la deuxième version de l'outil GAJE à un cas industriel. Ceci est dans le but de montrer sa performance et sa capacité vis à vis du test exhaustif de gros systèmes.

3 Application : Étude d'une carte à puce sans contact

3.1 Contexte de l'étude

Pour illustrer notre approche, nous avons appliqué GAJE à la vérification du système d'exploitation d'une carte à puce sans contact multi-applicative. Nous avons testé en particulier une carte sans contact pour le service de transport public fabriquée par la société ASK, leader sur le marché de cartes à puce sans contact [3] . Notre objectif principal est de vérifier les fonctionnalités et les caractéristiques de sécurité de cette carte. En effet, les cartes à puce sont omniprésentes de nos jours, plus de 200 millions de cartes sont utilisées à travers le monde pour le service de transport, de téléphonie, d'assurance maladie, des services bancaires, d'identité, etc. Les fraudes sont assez critiques pour ce type de cartes. Ainsi, il faut s'assurer de l'absence d'éventuelles vulnérabilités du système de codage, que des pirates pourraient exploiter.

Par ailleurs, la complexité de la carte rend difficile pour un humain d'identifier toutes les situations possibles, en particulier celles non souhaitées ou de les valider par les méthodes classiques. De plus, la nature "sans contact" de ce système favorise considérablement les possibilités d'attaques malveillantes. Nous avons estimé avoir besoin d'environ 500000 années pour tester les 8 premiers octets si nous considérons un processeur classique qui est capable légitimement de générer 1000 cas de test par seconde. En outre, l'explosion combinatoire des modes de fonctionnement possibles de la carte rend quasiment impossible toute tentative de simulation exhaustive. Le problème est encore accentué quand le système intègre un traitement de données, où les résultats ont eux mêmes des effets significatifs sur le comportement du système. Avec toutes ces considérations, la validation du comportement de cette carte est une tâche sujette à de nombreuses difficultés pour l'entreprise ASK : d'où l'intérêt de GAJE.

3.2 Fonctionnement de la carte à puce

Nous avons commencé cette étude par la modélisation du fonctionnement de la carte à puce étudiée à partir d'une norme de transport appelée Calypso décrite par ASK. Cette norme définit 33 principales commandes. La succession de ces commandes présente les différents scénarios possibles de l'opération de la carte. La figure V.3 montre un exemple de scénario entre la carte et un terminal. Ce scénario est établi par l'exécution de différentes commandes de la norme. Il commence par l'activation d'une application Calypso à travers la commande "*Select*" émise par le terminal. La carte répond ainsi et fournit le numéro de série Calypso correspondant.

3. http ://www.ask-rfid.com/

Ensuite, une session peut être ouverte par la commande "*Open Session*". Durant cette session, la commande "*SV Get*" permet de lire le solde de la valeur stockée et de préparer l'achat. La commande suivante "*SV Debit*" permet donc d'achever l'opération de débit. La commande "*Append Record*" permet d'ajouter cet évènement dans le fichier journal de la carte. Enfin, la session est fermée à travers la commande "*Close Session*".

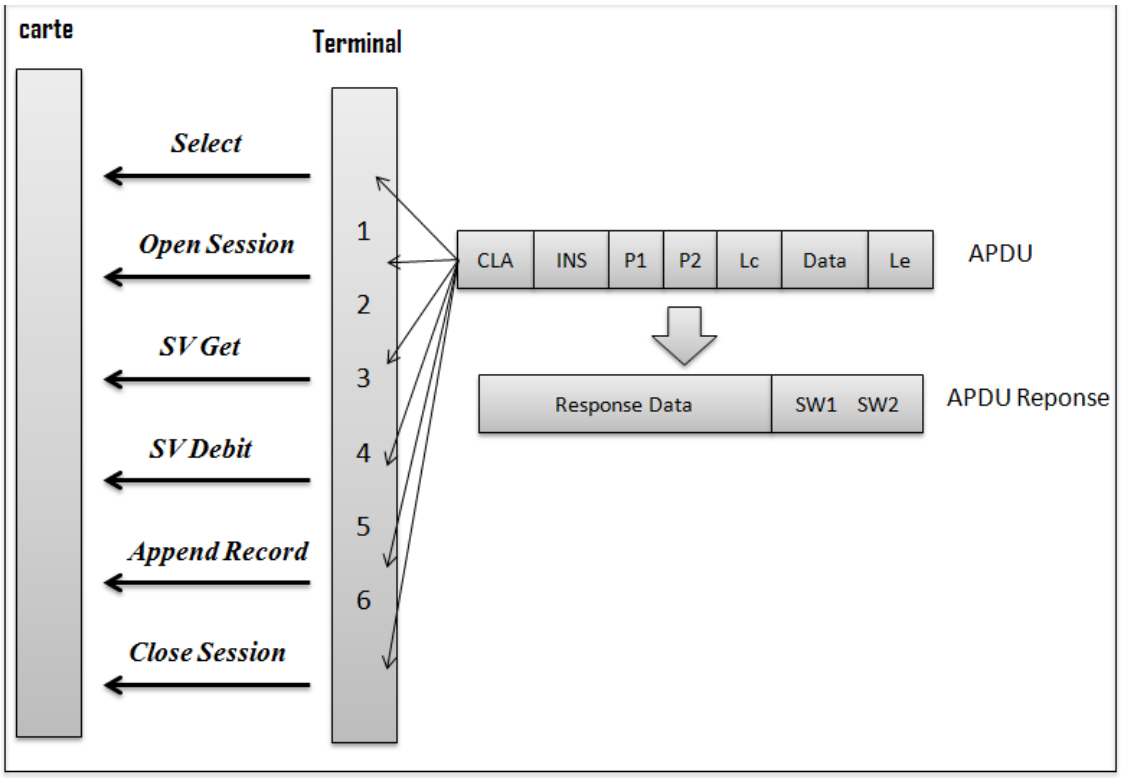


Figure V.3 – Exemple d'un scénario de la carte

Chaque commande envoyée par le terminal est exprimée par une trame de données appelée "*APDU*" (Application Protocol Data Unit). Cette trame représente les paramètres d'une commande, plus précisément le prochain octet à exécuter (*CLA, INS, P1, P2*, etc). Nous avons modélisé ces paramètres sous forme de pré-conditions. A cette fin, nous avons utilisé la bibliothèque SupLDD pour présenter les conditions sur ces paramètres par des contraintes numériques. Par exemple, l'exécution de l'octet *CLA* qui doit être compris entre 1 et 3 est modélisée par la pré-condition $GAJEINT(1 \leqslant CLA \leqslant 3)$. Une réponse à la trame *APDU* est ensuite retournée par la carte sous forme de trame de données appelée "*APDU Response*". Cette réponse inclut des données (numéro de série, solde actuel, la valeur stockée, etc) et un code d'exécution. Ce dernier renvoie le résultat de la commande. Par exemple, si les paramètres *P1* et *P2* ne sont pas supportés alors le code d'exécution *SW1SW2=6B00* est renvoyé pour indiquer une commande erronée.

Par ailleurs, si tous les paramètres sont corrects alors le code d'exécution *SW1SW2=9000* résultant est renvoyé pour indiquer une exécution correcte de la commande. Nous avons modélisé l'ensemble de ces codes d'exécution par des pré-conditions globales booléennes émises dans les sorties des commandes étudiées. *BackGaje-Append-Record* est un exemple de pré-condition booléenne émise quand la commande *Append Record* est correctement exécutée.

3.3 Configuration

La carte étudiée peut fonctionner dans différents modes suivant la technologie adaptée, avec ou sans contact, avec ou sans valeur stockée, etc. En général, ces caractéristiques doivent être initialement configurées pour spécifier chaque test. A priori, changer les paramètres de configuration de la carte nécessite de recompiler et tester chaque nouvelle spécification séparément et de régénérer tous les jeux de tests possibles. Cette approche n'est pas du tout réaliste, les tests de cette carte en industrie peuvent prendre plusieurs heures, voire des jours à compiler. En outre, ceci permettrait de générer autant de modèles que de types de cartes, ce qui peut limiter fortement la lisibilité et augmente le risque des bugs de spécification.

Contrairement à cette démarche complexe, nous avons généré un modèle générique unique pour tous les types de cartes et d'applications. Le principe de notre démarche est de modéliser les paramètres globaux de configuration de la carte par les pré-conditions système définies dans la section IV. 3. La spécification est plutôt établie à la fin à travers l'analyse des pré-conditions système. Par exemple, le test d'une carte à puce sans contact consiste à évaluer la pré-condition booléenne *GAJE-Contactless-mode* à 1 et de vérifier ensuite le contexte d'exécution correspondant. Par conséquent, les séquences avec la pré-condition *not GAJE-Contactless-mode* sont fausses et à corriger dans le système.

3.4 Modèle global

Nous avons modélisé l'ensemble des commandes définies dans la norme Calypso avec l'outil Galaxy 2.1.1. Nous avons choisi le langage synchrone Light Esterel afin d'interpréter la spécification de la carte et de générer des automates hiérarchiques. Ce choix a été justifié auparavant dans les sections II. 3.8 et IV. 4. Le modèle global généré présente 52 automates interconnectés contenant 765 états. Il montre une structure hiérarchique composée de 43 automates. Les autres automates sont parallèles. Ils agissent comme des observateurs pour gérer le contexte global de l'automate hiérarchique (*Closed Session*, *Verified PIN*, etc).

La figure V.4 montre une petite partie du modèle global introduisant le début des scénarios possibles de la carte étudiée.

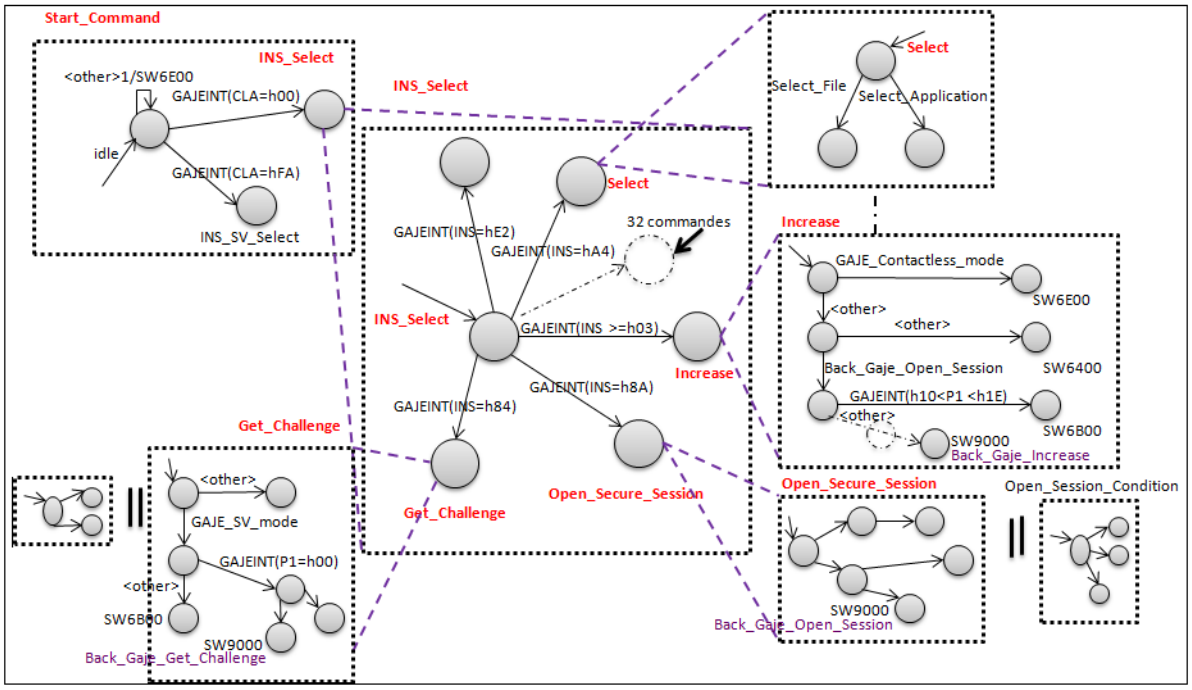


Figure V.4 – Modèle global de la carte à puce

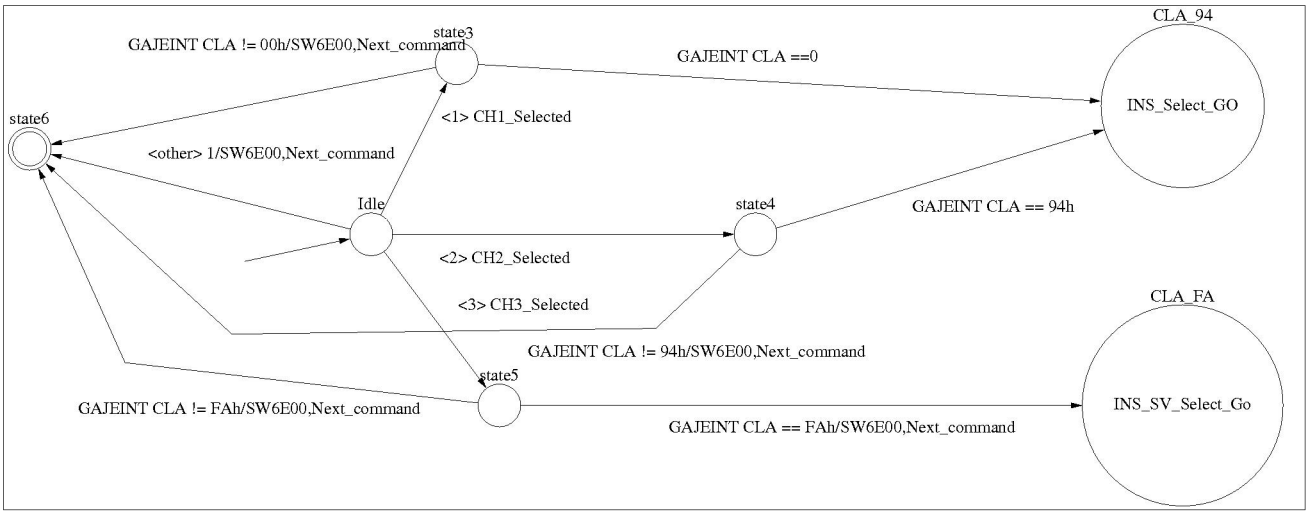


Figure V.5 – Start Command en galaxy

Ce modèle montre certaines pré-conditions locales telles que $GAJEINT(CLA = hFA)$, $GAJEINT(INS \geqslant h03)$ caractérisant les contraintes sur les octets d'exécution "*CLA*" et "*INS*". Les pré-conditions globales apparaissent comme des sorties pour certaines commandes et des entrées pour d'autres. Par exemple, *Back-Gaje-Open-Session* est une sortie qui indique que la commande *Open-Secure-Session* a été correctement exécutée. Cette pré-condition est

également présente dans les entrées de la commande *Increase* pour indiquer que cette dernière doit être exécutée après l'ouverture de la session. Les pré-conditions système tels que *GAJE-Contactless-mode* et *GAJE-SV-mode* montrent par exemple que la commande *Increase* doit être exécutée en mode sans contact et que la commande *Get-Challenge* doit être exécutée en mode *StoredValue*. Les figures V.5 et V.6 montrent respectivement les automates galaxy *Start-Command* et *INS-command*.

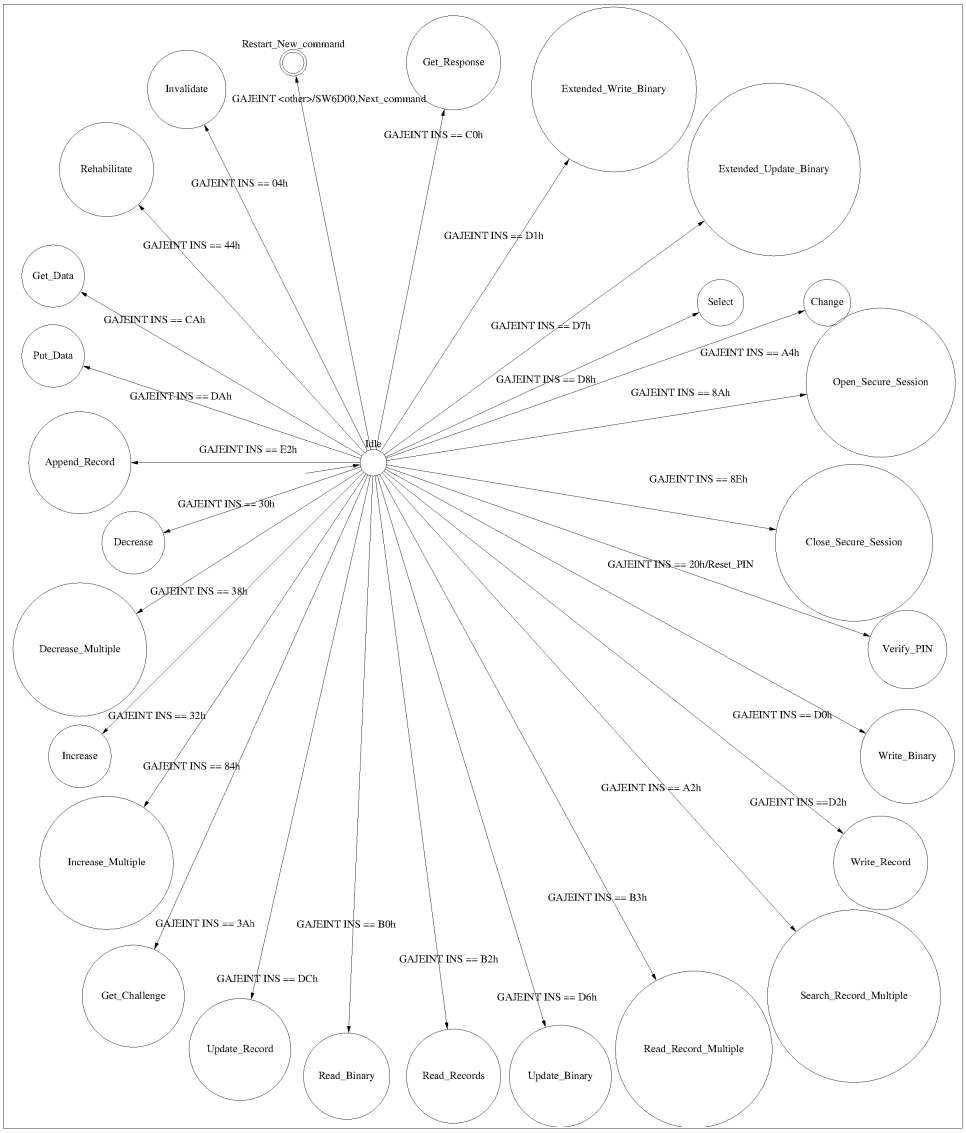


Figure V.6 – INS Command en galaxy

3.5 Aplatissement et compilation

Aplatir la structure hiérarchique (*Start-Command*, *INS-Select*, *Change*, etc) avec "Autom-expand" montre dans la figure V.7 une structure aplatie en parallèle avec les automates de contrôle (*Get-Challenge-Condition*, *Open-Session-Condition*, etc). Grâce à l'outil d'aplatissement "Autom-expand", nous sommes passé d'un modèle de 2^{150} états à un modèle à 765 états et 945 transitions seulement. Ce modèle aplati a donné également des résultats importants lors du processus de compilation par "Autom-abstract" suivi de "Autom2circuit", nous sommes passé de 477 registres à un modèle à 22 registres seulement.

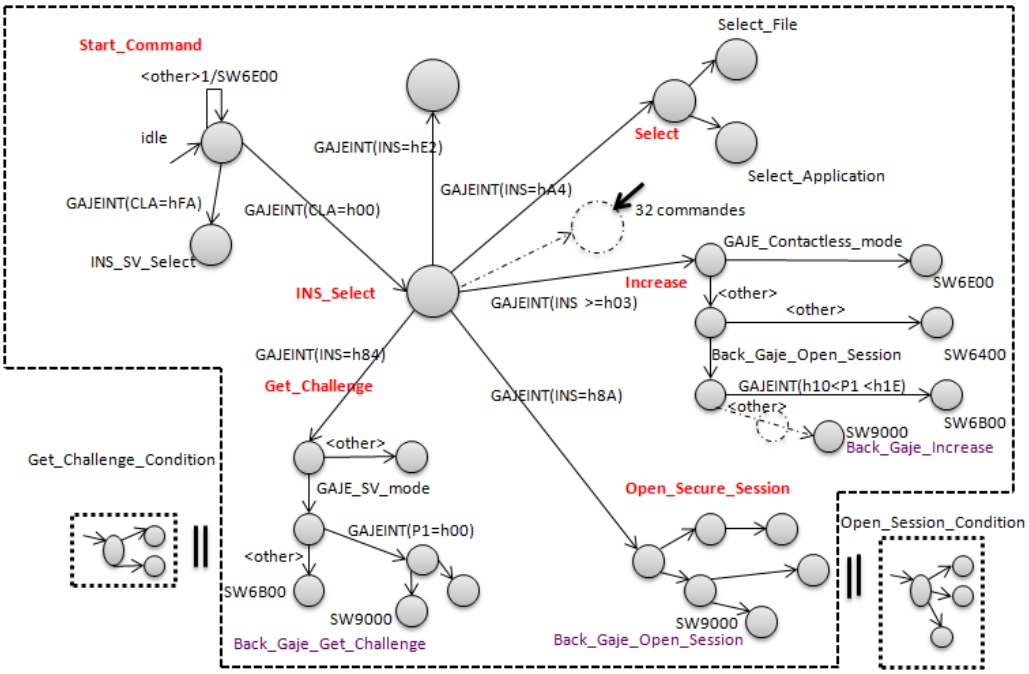


Figure V.7 – Modèle aplati de la carte à puce

3.6 Génération symbolique de séquences

Nous présentons dans cette section l'application du générateur symbolique de séquences sur deux représentations différentes du modèle de la carte à puce étudiée. Nous avons commencé par un test classique de cette carte qui consiste à parcourir et à vérifier toutes les séquences possibles des 33 commandes de la norme calypso. Il s'agit dans ce cas de parcourir tous les chemins de l'arbre de la figure V.8 représentant un extrait des successions possibles des commandes. Ceci correspond en pratique à l'analyse répétitive de l'automate mis à plat (figure V.9). Cet automate obtenu par Autom-expand est parfaitement illisible pour un être humain.

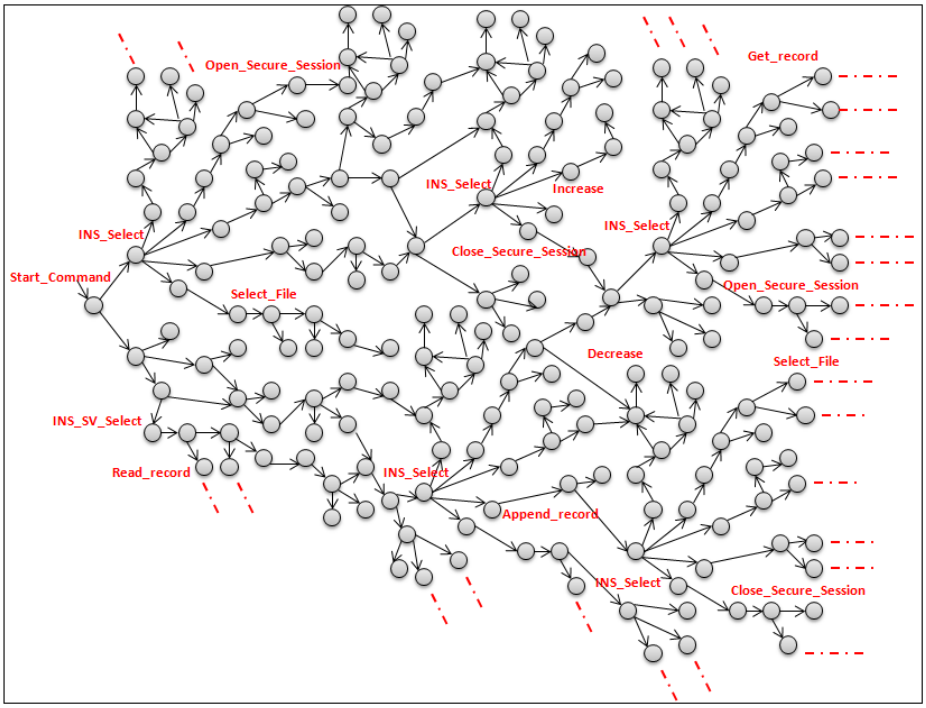


Figure V.8 – Test Classique de la carte Calypso

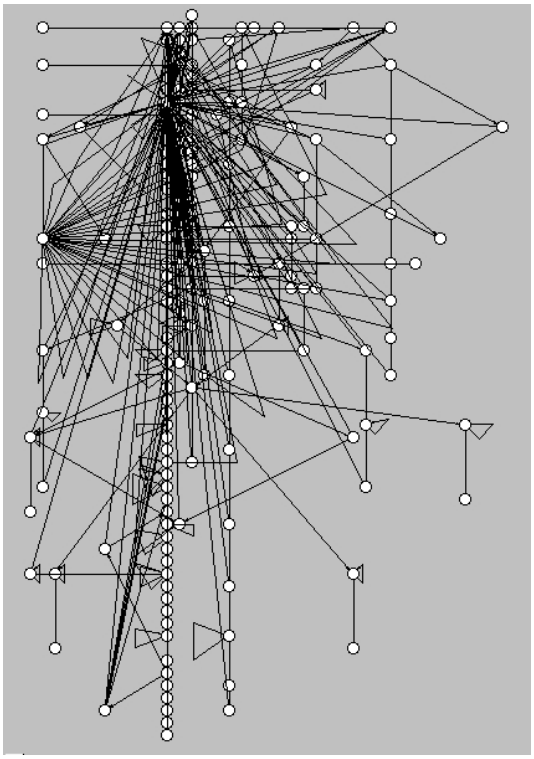


Figure V.9 – Automate Aplati par Autom expand

Nous avons effectué ensuite un deuxième test sur l'automate réduit à l'exécution d'une seule commande à la fois pour illustrer notre approche. A cette fin, nous avons appliqué GSS sur le modèle aplati de la carte représenté dans la figure V.8. Ce test consiste donc à tester séparément chaque sous espace significatif. En d'autres termes, nous allons parcourir tous les chemins possibles de chaque commande.

Nous avons mené nos expériences sur un PC avec un processeur Intel Dual Core, 2 GHz et 8 Go de RAM. Les résultats de ces tests montrent dans la figure V.10 l'évolution du nombre de séquences générées par rapport au nombre d'octets testés. Le test de l'approche classique montre une évolution exponentielle. Le nombre de séquences généré explose au bout de 13 octets testés générant 3993854132 séquences possibles. Sachant que chaque commande dans la norme Calypso est codée en 8 octets minimum, cette approche classique n'est même pas en mesure de tester plus que 2 commandes de la carte Calyso.

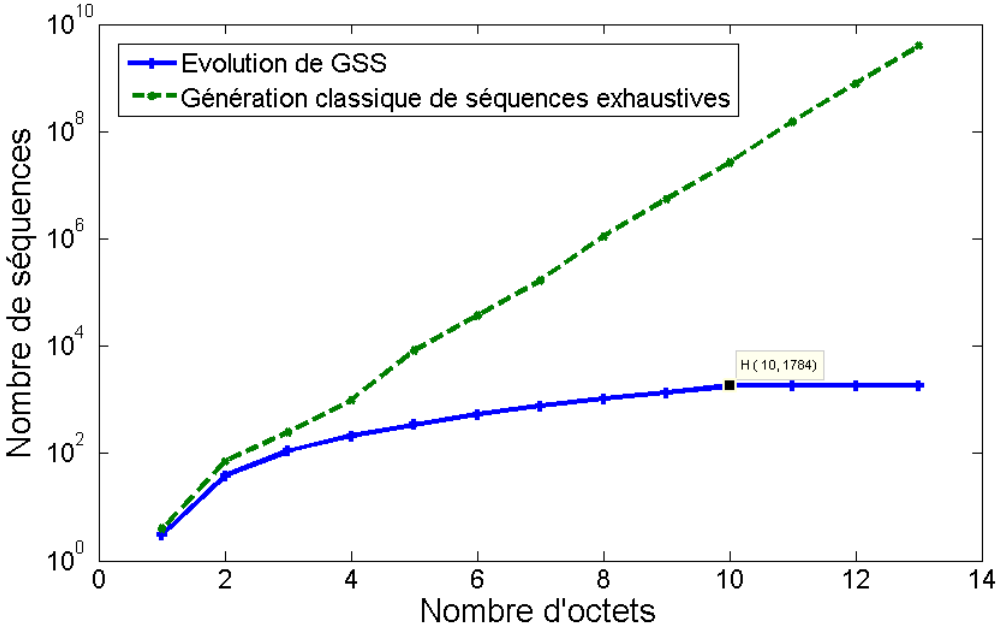


Figure V.10 – Evolution des tests

Cependant, la deuxième courbe représentant le nombre de séquences générées par notre outil GAJE est nettement inférieure. Elle stabilise en un temps assez court au point H(10, 1784). Ce résultat montre que notre outil permet de couvrir l'ensemble des 33 commandes Calypso en 10 pas (10 octets) seulement, générant un nombre total de 1784 chemins avec leurs propres pré-conditions. Couvrir l'ensemble d'états en 10 pas démontre le fait que nous testons séparément une seule commande (8 octets) à la fois. Les octets supplémentaires (2 octets dans notre cas) correspondent au test des pré-conditions système.

Nous montrons dans la figure V.11 un extrait des 1784 séquences générées (*EXPLORATION END FINDING 1784 PATHS...*). Cet extrait présente les deux derniers chemins exécutés. Il montre également le contexte requis qui doit être satisfait pour l'exécution de chaque séquence.

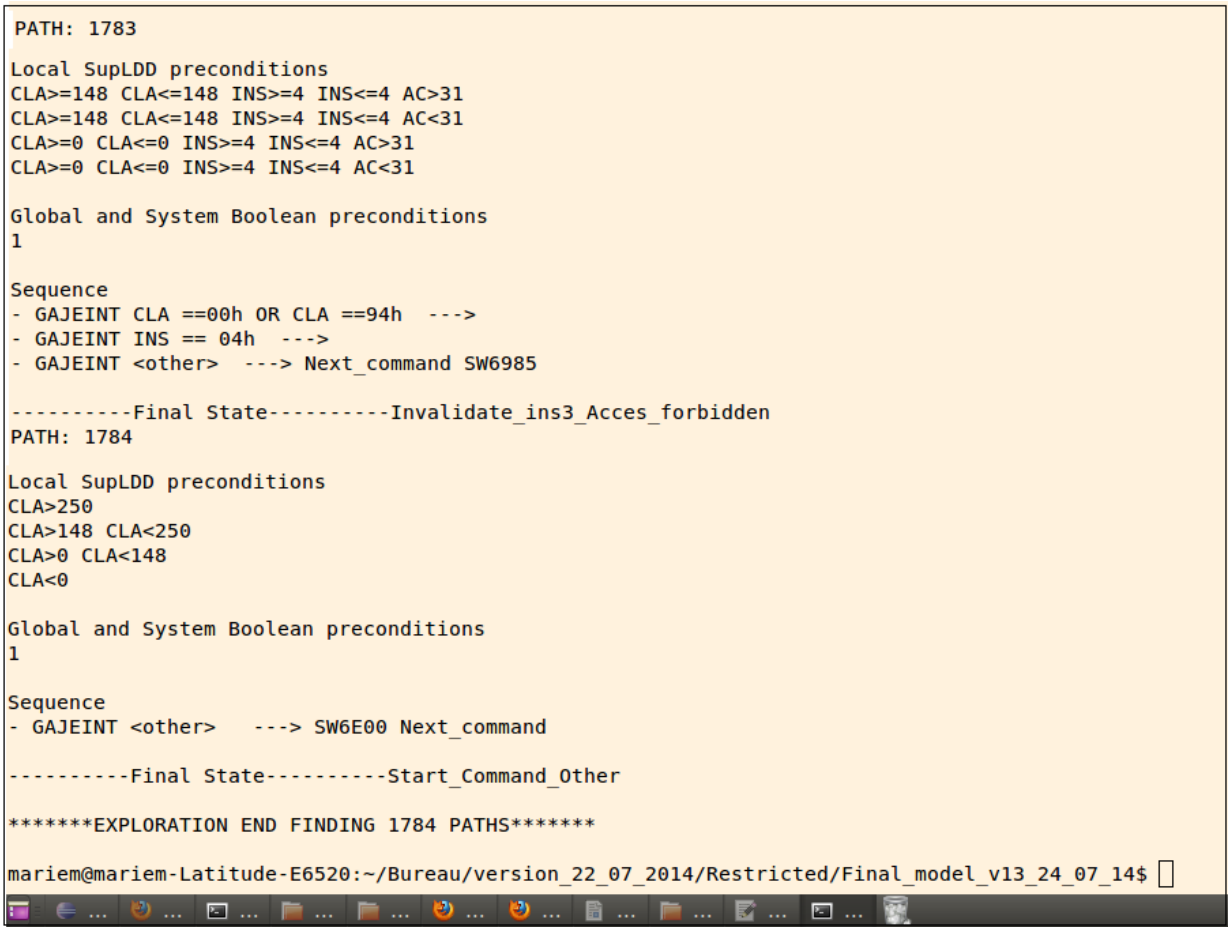


Figure V.11 – Exploration de toutes les séquences

La figure V.12 montre l'exécution du chemin 1428 et de son contexte qui aboutit à l'état final "*SV-UnDebit-ins32-Correct-execution*". Ce contexte regroupe l'ensemble des trois pré-conditions locales $\{CLA = 250\,;\, INS = 188\,;\, DF = 0\,;\, LC = 20\,;\, Amount < 0\,;\, Transaction - Counter > 0\,;\, TNum > 255\}$ OU $\{CLA = 250\,;\, INS = 188\,;\, DF = 0\,;\, LC = 20\,;\, Amount < 0\,;\, Transaction - Counter > 0\,;\, TNum = 255\,;\, etc\ \}$, globales $\{$ *!Back-Gaje-Al-Open-Session ; Back-Gaje-SV-Get*$\}$ et système $\{$ *!contact mode ; Present-SV ; !Session-memory-full ; Correct-Signature*$\}$.

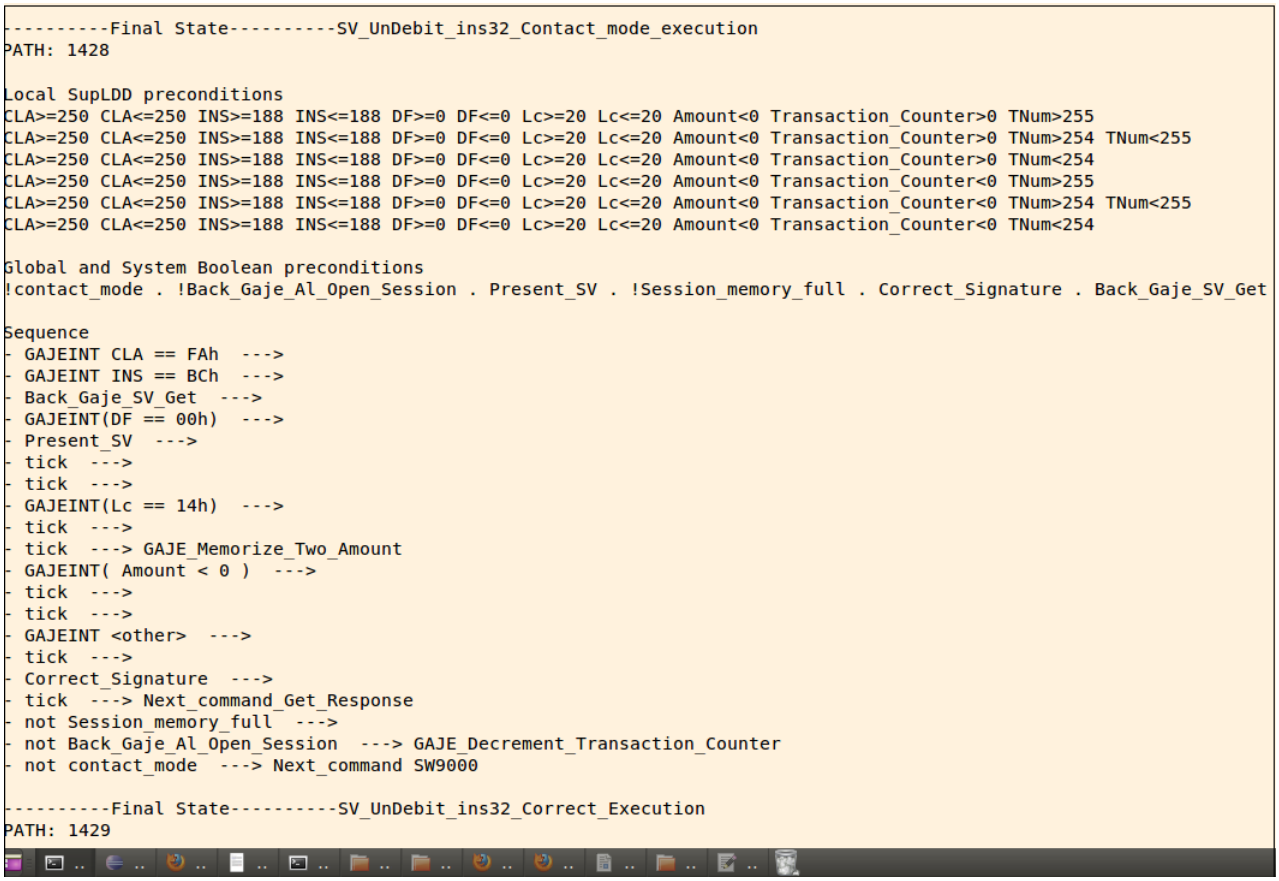


Figure V.12 – Exemple de séquence et contexte générés

Nous avons simulé dans la figure V.13 un cas d'anomalie dans le comportement de la carte étudiée : l'exécution de la séquence 1578 est impossible (Death Sequence), elle doit être corrigée dans la norme Calypso. Ce résultat a été émis suite à une conjonction nulle des préconditions locales $\{CLA = 00h\ OR\ CLA = 94h\}$; $\{INS = 8Ah\}$; $\{TransactionCounter! = 00h\}$; $\{RecordNumber \geqslant 01h\ AND\ RecordNumber \leqslant 31h\}$; $\{KeyNumber \geqslant 01h\}\ AND\ KeyNumber \leqslant 03h\}$; $\{RecordNumber > FFh\}$. Cette conjonction a été exécutée par la fonction *SupLDD-And* qui a indiqué que "*Record-Number*" ne peut pas être inférieur à *31h* et supérieur à *FFh* au même temps. **Cette anomalie correspond à une sur-spécification de la norme Calypso**.

Nous avons simulé dans la figure V.14 un deuxième cas d'anomalie dans le comportement de la carte étudiée : le message "Incomplete Behavior" indique dans cette séquence qu'un comportement inaccompli se trouve au niveau de la $11^{ème}$ transition. En effet, uniquement deux types de comportements sont définis : le cas où (Tag=54h) et le cas où (Tag=03h). Il manque donc tous les états où Tag est différent de 54h et de 03h. **Ce phénomène correspond à une sous-spécification de la norme Calypso**. Cette vérification est assurée automatiquement par la

```
CLA>=0 CLA<=0 INS>=138 INS<=138 Transaction_Counter>=0 Transaction_Counter<=0

Global and System Boolean preconditions
1

Sequence
- GAJEINT CLA ==00h OR CLA ==94h  --->
- GAJEINT INS == 8Ah  --->
- GAJEINT( Transaction_Counter == 00h)  ---> SW6900 Next_command

----------Final State----------Open_Secure_Session_ins20_TransactionCounternull

PATH: 1578

Empty LDD -----> Death sequence   GAJEINT (Record_Number > FFh)

- GAJEINT CLA ==00h OR CLA ==94h  --->
- GAJEINT INS == 8Ah  --->
- GAJEINT (Transaction_Counter !=00h)  --->
- GAJEINT (Record_Number >= 01h AND Record_Number <= 31h AND Key_Number >=01h AND Key_Number <= 03h)
 ---> GAJE_Memorize_bool_Record_Number
- GAJEINT (Record Number > FFh)  ---> SW6A83 Next_command
---> Death Sequence !!
mariem@mariem-Latitude-E6520:~/Bureau/version_22_07_2014/Restricted/Final_model_v13_24_07_14$
```

Figure V.13 – Exemple de sur-spécification de la norme Calypso

```
Post transitions:
Parse expression GAJEINT Tag == 54h

Parse expression GAJEINT Tag == 03h

-----> Incomplete Behavior

Local SupLDD preconditions
CLA>=148 CLA<=148 INS>=215 INS<=215 DF>1 Lc>=7 Lc<=255 P2>=0 P2<=30 P1>=0 P1<=0 SFI>=0 SFI<=30 AC>0 EFtype>=1 EFtype<=1 Tag>=84 Tag<=84
CLA>=148 CLA<=148 INS>=215 INS<=215 DF>1 Lc>=7 Lc<=255 P2>=0 P2<=30 P1>=0 P1<=0 SFI>=0 SFI<=30 AC<0 EFtype>=1 EFtype<=1 Tag>=84 Tag<=84
CLA>=148 CLA<=148 INS>=215 INS<=215 DF<1 Lc>=7 Lc<=255 P2>=0 P2<=30 P1>=0 P1<=0 SFI>=0 SFI<=30 AC>0 EFtype>=1 EFtype<=1 Tag>=84 Tag<=84
CLA>=148 CLA<=148 INS>=215 INS<=215 DF<1 Lc>=7 Lc<=255 P2>=0 P2<=30 P1>=0 P1<=0 SFI>=0 SFI<=30 AC<0 EFtype>=1 EFtype<=1 Tag>=84 Tag<=84
CLA>=0 CLA<=0 INS>=215 INS<=215 DF>1 Lc>=7 Lc<=255 P2>=0 P2<=30 P1>=0 P1<=0 SFI>=0 SFI<=30 AC>0 EFtype>=1 EFtype<=1 Tag>=84 Tag<=84
CLA>=0 CLA<=0 INS>=215 INS<=215 DF>1 Lc>=7 Lc<=255 P2>=0 P2<=30 P1>=0 P1<=0 SFI>=0 SFI<=30 AC<0 EFtype>=1 EFtype<=1 Tag>=84 Tag<=84
CLA>=0 CLA<=0 INS>=215 INS<=215 DF<1 Lc>=7 Lc<=255 P2>=0 P2<=30 P1>=0 P1<=0 SFI>=0 SFI<=30 AC>0 EFtype>=1 EFtype<=1 Tag>=84 Tag<=84
CLA>=0 CLA<=0 INS>=215 INS<=215 DF<1 Lc>=7 Lc<=255 P2>=0 P2<=30 P1>=0 P1<=0 SFI>=0 SFI<=30 AC<0 EFtype>=1 EFtype<=1 Tag>=84 Tag<=84

Global and System Boolean preconditions
!Wrong_Key . GAJE_Verify_SFI . Back_Gaje_Al_Open_Session

Sequence
- GAJEINT CLA ==00h OR CLA ==94h  --->
- GAJEINT INS == D7h  --->
- Back_Gaje_Al_Open_Session and not Wrong_Key  --->
- GAJEINT P1 == 00h  --->
- GAJEINT (P2 >= 00h AND P2 <= 1Eh)  ---> GAJE_Memorize_bool_SFI
- GAJE_Verify_SFI  --->
- GAJEINT (SFI >= 00h and SFI <= 1Eh)  --->
- GAJEINT <other>  --->
- GAJEINT (EFtype == 01h)  --->
- GAJEINT (Lc >= 07) and ( Lc<= FFh)  ---> GAJE_Memorize_int_EUB_Lc
- GAJEINT Tag == 54h  --->

----------Final State----------Extended_Update_Binary_ins3_Valid_Tag
```

Figure V.14 – Exemple de sous-spécification de la norme Calypso

fonction *SupLDD-Or* qui permet de vérifier si l'union de toutes les transitions sortantes d'un état donné est égale à tout l'espace d'états. Cette opération permet de vérifier le déterminisme

du système étudié comme a été expliqué dans la section IV. 6.

Nous présentons dans la section suivante les résultats de l'opération de Backtrack établie sur les pré-conditions locales et globales extraites des séquences générées afin d'établir tous les cas de tests possibles de l'opération de la carte Calypso.

3.7 Génération des cas de tests

3.7.1 Génération manuelle des cas de tests

Nous décrivons dans cette partie la méthode optée par les industriels pour générer les différents cas de tests possibles à appliquer pour la carte Calypso. Cette phase de test est extrêmement importante pour déterminer si la carte répond correctement aux exigences et aux spécifications mentionnées dans la norme Calypso. La figure V.15 montre un extrait des cas de tests générés manuellement pour la commande *SV-Undebit*.

588	Stored Value Debit	REQ_177_02	SV Undebit increases the stored value by an amount less than or equal to the previous SV Debit, within a session
589	Stored Value Debit	REQ_178_01	SV Undebit in a session returns SW '6200' instead of '9000'
590	Stored Value Debit	REQ_178_02	SV Undebit in a session is canceled if the session does not close correctly: abort session
591	Stored Value Debit	REQ_178_03	SV Undebit in a session is canceled if the session does not close correctly: RF field interruption
592	Stored Value Debit	REQ_178_04	SV Undebit in a session : further comands reading an SV file return an error until the end of the session
593	Stored Value Debit	REQ_179_01	SV Undebit outside of a session : the transaction counter is decremented if the command is correct
594	Stored Value Debit	REQ_179_02	SV Undebit outside of a session : the transaction counter is decremented if the certificate is incorrect.
595	Stored Value Debit	REQ_180_01	SV Undebit outside of a session: the correct information is stored in the SV Purchase Log after the
596	Stored Value Debit	REQ_180_02	SV Undebit within a session: the correct information is stored in the SV Purchase Log after the command
597	Stored Value Debit	REQ_181_01	SV Undebit does not change the current file and DF outside of a session
598	Stored Value Debit	REQ_181_02	SV Undebit does not change the current file and DF within a session
599	Stored Value Debit	REQ_182_01	SV Undebit returns SW '9000' for correct execution, with the correct format and length
600	Stored Value Debit	REQ_182_02	SV Undebit returns SW '6200' for correct execution within a session
601	Stored Value Debit	REQ_182_03	SV Undebit returns SW '6400' if the session memory is full
602	Stored Value Debit	REQ_182_04	SV Undebit returns SW '6700' for unsupported Lc values
603	Stored Value Debit	REQ_182_05	SV Undebit returns SW '6900' if the Transaction Counter is 0
604	Stored Value Debit	REQ_182_06	SV Undebit returns SW '6900' if the SV TNum is FFFEh or FFFFh
605	Stored Value Debit	REQ_182_07	SV Undebit returns SW '6985' if no SV Get has been previously done
606	Stored Value Debit	REQ_182_08	SV Undebit returns SW '6985' if the DF is invalidated
607	Stored Value Debit	REQ_182_09	SV Undebit returns SW '6985' if the balance underflows
608	Stored Value Debit	REQ_182_10	SV Undebit returns SW '6985' if the amount is positive
609	Stored Value Debit	REQ_182_11	SV Undebit returns SW '6988' if the signature is incorrect
610	Stored Value Debit	REQ_182_12	SV Undebit returns SW '6A80' if the undebit amount is greater than the previous debit amount
611	Stored Value Debit	REQ_182_13	SV Undebit returns SW '6D00' if the stored value application is not present

Test Cases List

Prêt

Figure V.15 – Les cas de tests de la commande SV Undebit

Le principe de ce test est d'imaginer tous les scénarios possibles de l'exécution de la commande *SV-Undebit*. Par exemple, l'exécution de *SV-Undebit* doit générer le code *SW6985* si la commande *SV-Get* n'est pas exécutée auparavant ou si *DF* est invalidé (lignes 605-606). Dans

le cas d'une exécution correcte au cours une session ouverte, *SV-Undebit* doit émettre le code *SW6200* (ligne 600). Tester la carte Calypso consiste donc à imaginer et générer tous les cas de tests possibles pour les 33 commandes décrites dans la norme étudiée. Il s'agit d'appliquer ensuite ces cas de tests et de vérifier si la carte répond avec le code prévu. Cette phase de validation de la carte Calypso est assez coûteuse en temps et en ressources pour l'industrie. Par ailleurs, imaginer tous les cas de tests possibles est une tâche délicate qui s'expose à de grandes difficultés.

3.7.2 Génération automatique des cas de tests "Backtrack"

Contrairement aux tests générés manuellement en industrie, nous montrons dans cette partie des cas de tests exhaustifs générés automatiquement pour la validation de la carte Calypso. Nos résultats sont obtenus par l'application de notre outil BackGaje sur les pré-conditions locales et globales extraites des séquences générées à partir de Gaje2Test.

```
Backtrack   259 of state  Invalidate_ins3_Too_many_modifications_in_session
Backtrack command from State Invalidate_ins3_Too_many_modifications_in_session

x0>=148 x0<=148 x1>=4 x1<=4 x3>=0 x3<=0 x13>=31 x13<=31 x19>255
x0>=0 x0<=0 x1>=4 x1<=4 x3>=0 x3<=0 x13>=31 x13<=31 x19>255
Backtrack Sequence
- GAJEINT <other>  ---> Next_command SW6400
- GAJEINT Lc == 00h  --->
- tick  --->
- tick  --->
- Back_Gaje_V_Select_File  --->
- GAJEINT(AC == 1Fh )  --->
- GAJEINT INS == 04h  --->
- GAJEINT CLA ==00h OR CLA ==94h  --->
Prior commands to execute 0

Backtrack   260 of state  Start_Command_Other
Backtrack command from State Start_Command_Other

x0>250
x0>148 x0<250
x0>0 x0<148
x0<0
Backtrack Sequence
- GAJEINT <other>   ---> SW6E00 Next_command
Prior commands to execute 0
********* END Backtrack *******
mariem@mariem-Latitude-E6520:~/Bureau/version_22_07_2014/Restricted/Final_model_v13_24_07_14$
```

Figure V.16 – Génération de tous les cas de tests de la carte Calypso

Nous avons appliqué l'opération de Backtrack à partir des états décrits critiques dans la norme Calypso. Il s'agit de tous les états finaux du modèle Calypso aplati que nous avons présenté dans la figure V.7. Le résultat obtenu montre dans la figure V.16 la fin de génération de tous les jeux de tests possibles fournissant 260 cas de tests pour l'ensemble des 33 commandes

de la norme Calypso. La figure V.17 présente un exemple détaillé d'un cas de test généré. Ce dernier montre la marche en arrière à partir d'un état final "*SV-UnDebit-ins32-Postponed-response-data* de la commande *SV-Undebit* qui émet le code *SW6200*.

```
Backtrack   7 of state  SV_UnDebit_ins32_Postponed_reponse_data
Backtrack command from State SV_UnDebit_ins32_Postponed_reponse_data

x0>=250 x0<=250 x1>=188 x1<=188 x2>=0 x2<=0 x3>=20 x3<=20 x4<0 x5>0 x6>255
x0>=250 x0<=250 x1>=188 x1<=188 x2>=0 x2<=0 x3>=20 x3<=20 x4<0 x5>0 x6>254 x6<255
x0>=250 x0<=250 x1>=188 x1<=188 x2>=0 x2<=0 x3>=20 x3<=20 x4<0 x5>0 x6<254
x0>=250 x0<=250 x1>=188 x1<=188 x2>=0 x2<=0 x3>=20 x3<=20 x4<0 x5<0 x6>255
x0>=250 x0<=250 x1>=188 x1<=188 x2>=0 x2<=0 x3>=20 x3<=20 x4<0 x5<0 x6>254 x6<255
x0>=250 x0<=250 x1>=188 x1<=188 x2>=0 x2<=0 x3>=20 x3<=20 x4<0 x5<0 x6<254
Backtrack Sequence
- Back_Gaje_Al_Open_Session  ---> SW6200 Back_Gaje_Warning_status
- not Session_memory_full  --->
- tick  ---> Next_command_Get_Response
- Correct_Signature  --->
- tick  --->
- GAJEINT <other>  --->
- tick  --->
- tick  --->
- GAJEINT( Amount < 0 )  --->
- tick  ---> GAJE_Memorize_Two_Amount
- tick  --->
- GAJEINT(Lc == 14h)  --->
- tick  --->
- tick  --->
- Present_SV  --->
- GAJEINT(DF == 00h)  --->
- Back_Gaje_SV_Get  --->
- GAJEINT INS == BCh  --->
- GAJEINT CLA == FAh  --->
Backtrack variable is Back_Gaje_SV_Get
Backtrack variable is Back_Gaje_Al_Open_Session
Backtrack variable is Back_Gaje_Verify_PIN
Prior commands to execute 3
Backtrack command from State SV_Get_ins29_Correct_execution

x0>=250 x0<=250 x1>=124 x1<=124 x8>=9 x8<=9 x9>=38 x9<=38
x0>=250 x0<=250 x1>=124 x1<=124 x8>=9 x8<=9 x9>=30 x9<=30
x0>=250 x0<=250 x1>=124 x1<=124 x8>=9 x8<=9 x9>=0 x9<=0
x0>=250 x0<=250 x1>=124 x1<=124 x8>=7 x8<=7 x9>=41 x9<=41
x0>=250 x0<=250 x1>=124 x1<=124 x8>=7 x8<=7 x9>=33 x9<=33
x0>=250 x0<=250 x1>=124 x1<=124 x8>=7 x8<=7 x9>=0 x9<=0
Backtrack Sequence
```

Figure V.17 – Exemple de cas de test

A partir des pré-conditions globales extraites *Back-Gaje-Al-Open-Session* et *Back-Gaje-SV-Get*, nous avons identifié les commandes qui doivent être exécutées (*Open-Secure-Session* et *SV-Get*) auparavant pour arriver au point de départ. Nous avons cherché ensuite d'une manière récursive les pré-conditions globales de chaque commande identifiée afin de tracer tout

le chemin vers l'état d'origine. Les résultats du Backtrack global indiquent dans la figure V.17 que la commande *Open-Secure-Session* doit être précédée de la commande *Verify-PIN*. Ainsi, le chemin final consiste à rebrousser les quatre commandes *SV-Undebit*, *SV-Get*, *Open-Secure-Session* et *Verify-PIN* dans l'ordre. Le parcours de chaque commande s'établit à partir de son état final, par exemple "*SV-Get-ins29-Correct-execution* qui émet la pré-condition globale "*Back-Gaje-SV-Get*. Enfin, le chemin constitué de l'état final de la commande *SV-Undebit* jusqu'à l'état initial de la commande *Verify-PIN* constitue un cas de test possible de la carte Calypso pour atteindre l'état *SV-UnDebit-ins32-Postponed-response-data*.

4 Conclusion

Nous avons évalué dans ce chapitre notre outil de test GAJE issu de toutes les études présentées dans les chapitres précédents en l'appliquant au test d'un système réel industriel, une carte à puce sans contact pour le service de transport. Les résultats obtenus, par rapport à une approche classique de génération de jeux de tests, ont montré la capacité de notre outil à générer automatiquement des jeux de tests exhaustifs qui couvrent le comportement global du système étudié. A cette fin, nous avons élaboré un modèle global de la carte à puce à partir d'une norme appelée Calypso. Un soin important a été porté sur la modélisation de cette norme car la précision de cette modélisation conditionne directement la qualité du jeu de test généré automatiquement. Une interprétation différente pourrait amener à laisser passer des bugs de conception. D'un autre coté, un modèle précis et cohérent permet de faire ressortir très rapidement toute implémentation non-conforme avec la norme qu'elle soit volontaire ou non.

Chapitre VI

Conclusion générale et perspectives

La vérification et la validation des systèmes réactifs est une phase très importante et délicate, notamment pour les systèmes qui remplissent des fonctions de plus en plus critiques et complexes. Ainsi, ces systèmes, soumis à des contraintes normales ou exceptionnelles, doivent être validés en profondeur afin de s'assurer qu'ils répondent correctement aux comportements attendus. Vérifier le comportement de ces systèmes face aux failles et aux évènements imprévus révèle également un véritable défi de sécurité et de robustesse.

Dans ce contexte, la vérification formelle a pris de l'ampleur dans la conception des systèmes critiques. Cependant, le passage d'un cas d'étude à un cas industriel en pratique révèle de problèmes de passage à l'échelle pour les systèmes qui possèdent un nombre élevé d'états et d'entrées. Des solutions alternatives telles que le test sont alors nécessaires. Mais, la génération des données de test et de son exécution est généralement une tâche très couteuse et quasiment infinie notamment quand elle est exécutée manuellement : un cas de test est une longue séquence arbitraire de vecteurs d'entrées. Automatiser la phase de test permet de détecter plus de défauts du système et d'économiser considérablement le temps et l'effort humain. Néanmoins, les techniques de tests existantes reposent le plus souvent sur des critères limités de couverture et n'assurent pas la couverture totale du comportement global du système.

1 Bilan de la thèse

Notre idée est d'exploiter les avantages des deux approches de vérification formelle et de test qui sont considérées aujourd'hui comme complémentaires. Nous avons proposé dans ce travail de thèse un générateur automatique de jeux de tests qui intègre les principes des outils formels pour tester des systèmes réactifs relativement complexes. Contrairement à la plupart des outils de tests existants décrits dans la section III. 4 qui se limitent à un cadre strictement booléen, notre générateur est essentiellement adapté aux applications réelles gérant des entrées et des

sorties numériques. Il fournit des tests plus expressifs et évolués des systèmes complexes.

Une part importante de l'étude a consisté dans un premier temps à donner les principes de l'interprétation formelle d'une spécification du système en machines à états finis hiérarchiques et parallèles et la caractérisation de celle-ci par un ensemble de contraintes booléennes et numériques. Les techniques proposées de conception permettent de tester plusieurs types d'applications à la fois. Ceci a été établi par la conception d'un modèle générique du système étudié et la spécification du test se fait à la fin aux travers des pré-conditions systèmes prédéfinies. Nous avons montré à travers une étude de l'état de l'art que Light Esterel [RGR08] est un langage approprié pour interpréter les systèmes réactifs en un modèle bien structuré et facile à analyser et à tester dans la suite.

Dans le cadre de cette modélisation, nous avons proposé ensuite un algorithme efficace de quasi-aplatissement pour aplatir uniquement l'automate hiérarchique et l'adapter à la compilation ultérieurement. Contrairement aux méthodes classiques d'aplatissement total, nous avons conservé dans un premier temps le parallélisme global de l'application et procédé qu'à l'aplatissement des automates hiérarchiques séparés. En effet, l'aplatissement des machines parallèles explose généralement en nombre d'états. Ceci a été assuré grâce à l'utilisation d'un codage approprié sous forme d'équations d'états des automates : ce qui a permis de substituer le parallélisme par une concaténation et un tri des équations obtenues pour chaque automate. D'ailleurs, cet algorithme de quasi-aplatissement montre qu'il n'existe qu'un seul état actif à la fois dans chaque automate parallèle, ce qui a fourni une réduction considérable de l'espace d'état dans l'automate aplati.

Afin d'optimiser le modèle généré, nous avons proposé une compilation symbolique séparée des automates explicites générés en une machine de Mealy implicite. Notre processus de compilation vérifie le déterminisme du modèle compilé et ne nécessite que log_2(nombre d'états) registres au lieu d'un registre par état dans la plupart des techniques de compilation telle que la compilation *one-hot* [CI10]. Suite à ce codage des FSMs, nous avons effectué la mise en parallèle par la concaténation et le tri des équations obtenues pour chaque FSM compilé séparément. En conséquence, une réduction importante a été notée de la taille du modèle compilé.

Nous avons abordé dans une seconde part de notre travail le problème de manipulation des données numériques qui représente un enjeu majeur pour le test d'un grand nombre de logiciels de la vie réelle. En effet, nous nous sommes basés au début sur la représentation ingénieuse des variables du système et de ses fonctions booléennes par les graphes de décisions binaires BDD [Bry86]. Nous avons étendu ensuite notre travail pour représenter davantage des variables et des fonctions numériques réelles sans provoquer un dépassement de mémoire. Nous avons développé à cette fin une nouvelle bibliothèque de manipulation de données que nous avons

appelé "SupLDD" sur la base de la bibliothèque LDD [Cha09] étudiée en détails dans l'état de l'art. Cette bibliothèque adaptée à nos besoins a permis d'une part le calcul symbolique des données du système et d'autre part la vérification du comportement du système (déterminisme, séquences impossibles, identification des cas de tests possibles, etc.) .

Basés sur cette bibliothèque, nous avons mis en place les deux phases de génération des séquences et des cas de tests possibles. Notre approche est adaptée essentiellement pour le test des systèmes exécutant des commandes itératives. Par conséquent, nous nous sommes focalisés sur chaque sous-espace significatif du modèle global représentant une commande du système au lieu de considérer tout l'espace d'états. Ceci a permis de pallier considérablement au phénomène d'explosion combinatoire de l'espace d'états. Cette restriction a été assurée grâce à la caractérisation de l'ensemble des pré-conditions définissant le contexte d'exécution du sous-espace associé. Ainsi, tout le calcul complexe a été établi en particulier dans ce sous-espace significatif. Sur la base de cette représentation efficace du modèle étudié, nous avons généré la liste exhaustive de toutes les séquences possibles de chaque sous-espace exploré et de l'ensemble des pré-conditions définissant son contexte d'exécution. Enfin, la vérification du modèle dans sa globalité a consisté à manipuler et à vérifier les pré-conditions de chaque sous-espace étudié. Des calculs robustes en "SupLDD" ont permis de vérifier le déterminisme des automates analysés : l'intersection de toutes les transitions sortantes d'un état doit être égale à l'élément vide. Nous avons pu également détecter les séquences impossibles : une séquence ne peut exister que si l'intersection de ses contraintes numériques tout au long du chemin étudié est non nulle.

A partir des pré-conditions extraites de chaque sous-espace analysé, nous avons défini l'opération finale de marche en arrière "Backtrack" pour générer tous les cas de tests possibles du comportement du système testé. A travers un algorithme de recherche des états précédents et des calculs en "SupLDD", nous avons pu tracer tous les chemins possibles partant des états critiques finaux jusqu'à l'état initial racine ayant zéro pré-conditions.

L'implémentation des techniques de test que nous avons proposées dans ce manuscrit ont donné naissance à un nouveau outil appelé GAJE. Afin d'évaluer la performance de notre outil, nous avons muni des expérimentations sur un système réel de taille industrielle. Les résultats obtenus prouvent que notre méthodologie, ainsi que l'outil développé, peuvent être utilisés pour des applications industrielles de la vie réelle où le risque d'explosion combinatoire de l'espace d'états est important. La génération automatique des jeux de tests se fait d'une manière assez rapide et exhaustive. Elle permet également de détecter les sur-spécifications (séquences impossibles) et les sous-spécifications (indéterminisme) directement liées à la norme analysée du système ou à son interprétation. Ce retour d'information est essentiel pour toute la chaîne de conception.

2 Evolution et Perspectives

Le travail réalisé dans cette thèse ouvre de nombreuses perspectives pour des travaux futurs. Certaines constituent des perspectives immédiates à court terme alors que d'autres nécessitent une exploration plus approfondie à long terme.

2.1 Perspectives à court terme

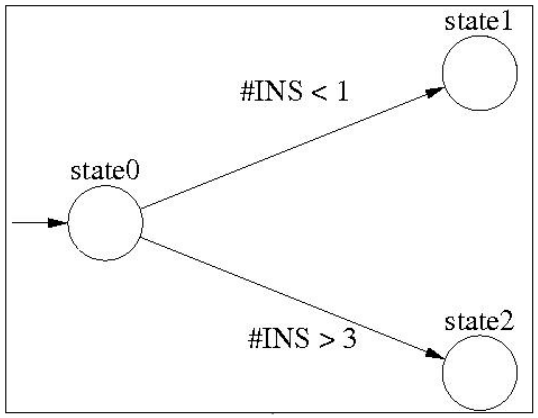
2.1.1 Optimisation de SupLDD

Nous avons implémenté dans ce travail la bibliothèque SupLDD au dessus du paquet CUDD [1]. Ce dernier représente un paquet qui implémente les bases et les principales fonctions des diagrammes de décisions binaires BDD. CUDD s'appuie sur une représentation très simple des BDDs alors que d'autres outils tels que Autom2circuit et Merge-Blif utilisent des BDDs à arcs complémentés [BRB90] [MB88] beaucoup plus efficaces et rapides en terme de manipulation et qui génèrent des BDDs beaucoup plus réduits. Il serait donc intéressant de refaire la bibliothèque SupLDDD sur la base des BDDs à arcs complémentés afin d'obtenir une meilleure optimisation de la bibliothèque de manipulation des données numériques SupLDD.

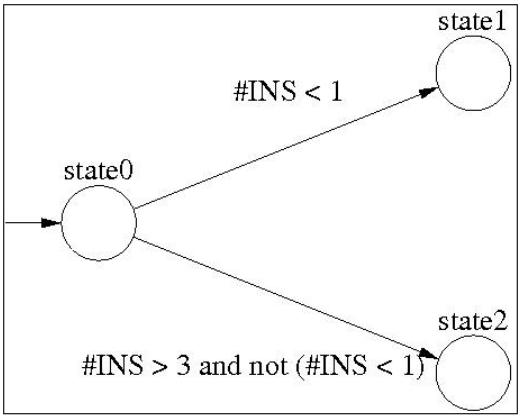
2.1.2 Intégration de SupLDD dans CLEM

L'outil de compilation CLEM présenté dans la section II. 4.2 ainsi que sa version allégée autom2circuit ne sont opérationnels que pour les variables booléennes. Une contribution intéressante serait d'intégrer la bibliothèque SupLDD à l'abstraction des données dans CLEM. En effet, pour l'instant, chaque test sur les valeurs du système est remplacé par une variable booléenne indépendante et les dépendances entre ses variables booléennes sont malheureusement perdues. Par exemple, ayant $\{x \geqslant 3\}$ et $\{x \geqslant 5\}$, CLEM ne sait pas conclure que x est supérieur à 5. Ceci génère de multiples incidences : n'ayant pas la capacité de SupLDD pour détecter les séquences impossibles, CLEM génère potentiellement de fausses séquences liées par exemple à $\{x \leqslant 3\}$ et $\{x > 5\}$. En outre, ce compilateur détecte de fausses causalités : Considérons la figure VI.1a, CLEM ne se rend pas compte que les transitions $\{INS > 3\}$ et $\{INS < 1\}$ sont exclusives et considère que l'automate est indéterministe. Il est impérativement nécessaire dans ce cas de remplacer la condition $\{INS > 3\}$ par $\{(INS > 3)\ and\ not\ (INS < 1)\}$ comme l'illustre la figure VI.1b. Intégrer les SupLDD permettra de gérer toutes ces anomalies dans CLEM.

1. http ://sourceforge.net/projects/lindd



(a) Exemple de fausse Causalité dans CLEM



(b) Version acceptée par CLEM

2.2 Perspectives à long terme

D'après le cas d'application étudié dans la section V. 3, nous avons conclu qu'un soin important doit été apporté sur la modélisation de la norme support car la précision de celle-ci conditionnera directement la qualité du jeu de test généré automatiquement. En effet, une interprétation différente pourrait amener à laisser passer des bugs de conception. D'un autre coté, cette tâche est très délicate et peut prendre plusieurs mois ce qui fut le cas de l'interprétation de la norme Calypso. Enfin, un modèle précis et cohérent permet de faire ressortir très rapidement toute implémentation non-conforme avec la norme qu'elle soit volontaire ou non.

C'est pourquoi, nous aimerions définir dans des travaux futurs un nouveau langage standard qui permette d'interpréter automatiquement voire de compiler toute spécification d'un système donné en un modèle exploitable directement par des outils de vérification et de génération de tests comme les nôtres. Ceci permettrait de standardiser les spécifications données, garantir la précision et la cohérence du modèle généré, réduire l'effort humain, et bien évidemment gagner un temps de conception conséquent.

Références bibliographiques

[AB00] C. André and H. Boufaied. Execution machine for synchronous languages. In *IDPT'2000 (Integrated Design and Process Technology)*, pages 144–149, Dallas (TX), June 2000. SDPS, (TX). 16

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6) :509–516, June 1978. 37

[And96] C. André. Representation and analysis of reactive behaviors : A synchronous approach. In *Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille (F), July 1996. IEEE-SMC. 16, 20, 61, 63

[Ard] Laurent Arditi. A bit-vector algebra for binary moment diagrams. 43

[AY01] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst.*, 23(3) :273–303, May 2001. 35

[BC01] Randal E. Bryant and Yirng-An Chen. Verification of arithmetic circuits using binary moment diagrams, 2001. 7, 42

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jerome Feret, Laurent Mauborgne, Antoine Mine, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. pages 196–207. ACM Press, 2003. 5

[BEL75] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. Select&mdash ;a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.*, 10(6) :234–245, April 1975. 32, 33, 70

[Ber99] Gérard Berry. The esterel v5 language primer. Technical report, 1999. 23

[Ber07] Gérard Berry. Scade : Synchronous design and validation of embedded control software. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33, India, January 2007. Springer Netherlands. 17

[BFG+93] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications, 1993. 7

[BGGL92] Gerard Berry, Georges Gonthier, Ard Berry Georges Gonthier, and Place Sophie Laltte. The esterel synchronous programming language : Design, semantics, implementation, 1992. 8, 16, 18, 61

[BHR08] Patricia Bouyer, Serge Haddad, and Pierre-Alain Reynier. Timed petri nets and timed automata : On the discriminating power of zeno sequences. *Inf. Comput.*, 206(1) :73–107, January 2008. 46

[BJM+10] Benjamin Blanc, Christophe Junke, Bruno Marre, Pascale Le Gall, and Olivier Andrieu. Handling state-machines specifications with gatel. *Electron. Notes Theor. Comput. Sci.*, 264(3) :3–17, December 2010. 57

[Bou97] Amar Bouali. Xeve : an esterel verification environment (version v1.3), 1997. 4

[BPZ] Lina Bentakouk, Pascal Poizat, and Fatiha Zaïdi. F. : A formal framework for service orchestration testing based on symbolic transition systems. In *In : Proc. of TESTCOM. (2009)*. 57

[BRB90] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a bdd package. In *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pages 40–45, June 1990. 102

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35 :677–691, August 1986. 7, 36, 37, 100

[BS08] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society. 58

[BT00] E. Bui and F. Thibaut. Test fonctionnel. `http://users.polytech.unice.fr/~hugues/GL~\/rational/robot/robot.htm`, 2000. 6

[BW94] Aart J. C. Bik and Harry A. G. Wijshoff. Implementation of fourier-motzkin elimination. Technical report, 1994. 46

[CCF+07] Roberto Cavada, Alessandro Cimatti, Anders Franzén, Krishnamani Kalyanasundaram, Marco Roveri, and R. K. Shyamasundar. Computing predicate abstractions by integrating bdds and smt solvers. In *Proceedings of the Formal Methods in Computer Aided Design*, FMCAD '07, pages 69–76, Washington, DC, USA, 2007. IEEE Computer Society. 45

[CCGR00] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv : a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2 :2000, 2000. 30

[CE05] Cristian Cadar and Dawson Engler. Execution generated test cases : How to make systems code crash itself. In *Proceedings of the 12th International Conference on Model Checking Software*, SPIN'05, pages 2–23, Berlin, Heidelberg, 2005. Springer-Verlag. 33

[CGH+95] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the futurebus cache coherence protocol, 1995. 6

[Cha09] A. ; Strichman O Chaki, S. ; Gurfinkel. Decision diagrams for linear arithmetic. *Formal Methods in Computer-Aided Design*, pages 53–60,, 2009. 47, 52, 101

[CI10] Garaur A. Chiuchisan I., Potorac A.D. Finite state machine design and vhdl coding techniques. In *10th International Conference on development and application systems*, pages 273–278, Suceava, Romania, 2010. Faculty of Electrical Engineering and Computer Science. 66, 100

[CKA06] Maciej Ciesielski, Priyank Kalla, and Serkan Askar. Taylor expansion diagrams : A canonical representation for verification of data flow designs. *IEEE Transactions on Computers*, 55(9) :1188–1201, 2006. 44

[Cla76] Lori A. Clarke. A program testing system. In *Proceedings of the 1976 Annual Conference*, ACM '76, pages 488–491, New York, NY, USA, 1976. ACM. 33

[Cou78] P. Cousot. *Méthodes itératives de construction et d'approximation de point fixes d'opérateurs monotone sur un treillis analyse sémantique des programmes*. PhD thesis, Thèse d'état, Grenoble, 1978. 4

[C.Y59] C.Y.Lee. Representation of switching circuits by binary decision programs. *Bell Systems Technical Journal*, 38 :985–999, 1959. 37

[CYF94] Ben Chen, Michihiro Yamazaki, and Masahiro Fujita. Bug identification of a real chip design by symbolic model checking. In Robert Werner, editor, *EDAC-ETC-EUROASIC*, pages 132–136. IEEE Computer Society, 1994. 6

[CZ95] Edmund Clarke and Xudong Zhao. Word level symbolic model checking : A new approach for verifying arithmetic circuits. Technical report, Pittsburgh, PA, USA, 1995. 43

[Dar96] Ian F. Darwin. *Checking C programs with lint - C programming utility.* O'Reilly, 1996. 5

[dBZ99] L. du Bousquet and N. Zuanon. An overview of lutess - a specification-based tool for testing synchronous software. In *In Proc. 14th IEEE Intl. Conf. on Automated SW Engineering*, pages 208–215, 1999. 56

[DDM06] B Dutertre and L De Moura. The Yices SMT solver. http://yices.csl.sri.com/tool-paper.pdf, 2006. 58

[Dow97] Mark Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2) :84–, March 1997. 2

[FHL+01] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise wcet determination for a real-life processor. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software, First International Workshop, EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485, Tahoe City, CA, USA, october 8-10 2001. Springer. 5

[FMY97] M. Fujita, P. C. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams : An efficient datastructure for matrix representation. *Form. Methods Syst. Des.*, 10(2-3) :149–169, April 1997. 40

[Fri95] E.G. Friedman. *Clock distribution networks in VLSI circuits and systems*. Institute of Electrical and Electronics Engineers, 1995. 18

[Gaf] Daniel Gaffé. Research web site. http://sites.unice.fr/dgaffe/recherche/research.html. 80, 82

[GD13] Ressouche A. Gaffé D. Algebraic framework for synchronous language semantics. In *Theoretical Aspects of Software Engineering (TASE), 2013 International Symposium on*, pages 51–58, Birmingham, UK, July 1-3 2013. IEEE. 22

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart : Directed automated random testing. *SIGPLAN Not.*, 40(6) :213–223, June 2005. 33

[GLGB87] Thierry Gautier, Paul Le Guernic, and Löic Besnard. Signal : A declarative language for synchronous programming of real-time systems. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 257–277, London, UK, UK, 1987. Springer-Verlag. 16, 17

[Gro] Jan Friso Groote. Binary decision diagrams for first order predicate logic. 45

[G.T10] G.TOUSSAINT. Hubble, le miroir de l'univers. http://www.lalibre.~be/actu/planete/hubble-le-miroir-de-l-univers-51b8bb27e4b0de6db9bb2c76, 24 avril 2010. 3

[GvdP00] Jan Friso Groote and Jaco van de Pol. Equational binary decision diagrams. In *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*, LPAR'00, pages 161–178, Berlin, Heidelberg, 2000. Springer-Verlag. 45

[Har87] David Harel. Statecharts : A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3) :231–274, June 1987. 20

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991. 16, 17, 61

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE Trans. Softw. Eng.*, 18(9) :785–793, September 1992. 4

[Hoe12] Christine Hoekenga. Tragedies in science : The crash of the mars climate orbiter. http ://www.visionlearning.com/blog/2012/09/21/tragedies-in-science-the-crash-of-the-mars-climate-orbiter/, September 21, 2012. 2

[Jea03] B. Jeannet. Dynamic partitioning in linear relation analysis : Application to the verification of reactive systems. *Form. Methods Syst. Des.*, 23(1) :5–37, July 2003. 4

[JJ05] Claude Jard and Thierry Jéron. Tgv : Theory, principles and algorithms : A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4) :297–315, August 2005. 57

[JM09] Bertrand Jeannet and Antoine Miné. Apron : A library of numerical abstract domains for static analysis, 2009. 47

[Joh73] Donald B. Johnson. A note on dijkstra's shortest path algorithm. *J. ACM*, 20(3) :385–388, July 1973. 74

[KC66] T. I. Kirkpatrick and N. R. Clark. Pert as an aid to logic design. *IBM J. Res. Dev.*, 10(2) :135–141, March 1966. 28

[KEH+09] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4 : Formal verification of an os kernel. In *ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES*, pages 207–220. ACM, 2009. 4

[KPV03] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *In Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003. 33

[LGR11] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. Klover : A symbolic execution and automatic test generation tool for c++ programs. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 609–615, Berlin, Heidelberg, 2011. Springer-Verlag. 58

[LM05] Shuvendu K. Lahiri and Madanlal Musuvathi. An efficient decision procedure for utvpi constraints. In *Proceedings of the 5th International Conference on Frontiers of Combining Systems*, FroCoS'05, pages 168–183, Berlin, Heidelberg, 2005. Springer-Verlag. 47

[LPV06] Y. T. Lai, M. Pedram, and S. B.K. Vrudhula. Evbdd-based algorithms for integer linear programming, spectral transformation, and function decomposition. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 13(8) :959–975, November 2006. 7

[LS92] Y.-T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, DAC'92, pages 608–613, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. 41

[MB88] Jean-Christophe Madre and Jean-Paul Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, DAC '88, pages 205–210, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press. 102

[MBLG00] Herve Marchand, Patricia Bournai, Michel LeBorgne, and Paul Le Guernic. Synthesis of discrete-event controllers based on the signal environment. In *IN DISCRETE EVENT DYNAMIC SYSTEM : THEORY AND APPLICATIONS*, pages 325–346, 2000. 4, 18

[McM92] Kenneth Lauchlin McMillan. *Symbolic Model Checking : An Approach to the State Explosion Problem.* PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209. 4

[Min06] Antoine Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19(1) :31–100, March 2006. 47

[ML98] Jesper Møller and Jakob Lichtenberg. Difference decision diagrams. Master's thesis, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, August 1998. 46

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM. 3

[MPFH11] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 95–111, Berlin, Heidelberg, 2011. Springer-Verlag. 33

[MR01] F. Maraninchi and Y. Rémond. Argos : an automaton-based synchronous language. *Computer Languages*, (27) :61–92, 2001. 20

[MS04] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. 6

[MSS96] J. P. Marques-Silva and K. A. Sakallah. Grasp - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996. 3

[NÓ1] J.C. NÉNOT. Les accidents d'irradiation, 1950-2000 leçons du passé. *Extrait de Radioprotection*, 36(4) :431–450, 2001. 2

[NWP11] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin Heidelberg, 2011. 5

[Pap08] V. Papailiopoulou. Automatic test generation for lustre/scade programs. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 517–520, Washington, DC, USA, 2008. IEEE Computer Society. 5

[PJJ07] F. Ployette, B. Jeannet, and T. Jéron. Stg : a symbolic test generation tool for reactive systems. TESTCOM/FATES07 (Tool Paper), June 2007. 57

[PTFV05] Ana C. R. Paiva, Nikolai Tillmann, João C. P. Faria, and Raul F. A. M. Vidal. Modeling and testing hierarchical guis. In *Proc.ASM05. Universite de Paris 12*, pages 8–11, 2005. 66

[Pug91] William Pugh. The omega test : A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM. 57

[QA] The mccabe iq difference. http ://mccabe.com/ ?file=./prod/qa.html. 6

[Ray08] Pascal Raymond. Synchronous program verification with lustre/lesar. In *Modeling and Verification of Real-Time Systems*, chapter 6. ISTE/Wiley, 2008. 17

[RG11] Annie Ressouche and Daniel Gaffé. Compilation modulaire d'un langage synchrone. 4(30) :441–471, June 2011. 28

[RGR08] Annie Ressouche, Daniel Gaffé, and Valerie Roy. Modular compilation of a synchronous language. In Roger Lee, editor, *Soft. Eng. Research, Management and Applications, best 17 paper selection of the SERA'08 conference*, volume 150, pages 157–171, Prague, August 2008. Springer-Verlag. 21, 100

[RL03] Frédéric Raimbault and Dominique Lavenier. ROOM. des machines reconfigurables orientées objet pour les applications spécifiques. *Technique et Science Informatiques*, 22(6) :759–782, 2003. 34

[Rud93] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '93, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press. 51

[SA06] Koushik Sen and Gul Agha. Cute and jcute : Concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 419–423, Berlin, Heidelberg, 2006. Springer-Verlag. 33

[SAH+00] Jørgen Staunstrup, Henrik Reif Andersen, Henrik Hulgaard, Jørn Lind-Nielsen, Kim G. Larsen, Gerd Behrmann, Kåre Kristoffersen, Arne Skou, Henrik Leerberg, and Niels Bo Theilgaard. Practical verification of embedded software. *Computer*, 33(5) :68–75, May 2000. 35

[Sch09] Klaus Schneider. The synchronous programming language quartz. Technical report, Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009. 21

[Sla98] Gregory Slabodkin. Software glitches leave navy smart ship dead in the water. *Government Computer News*, 1998. 2

[SLQP07] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor : A tiny hypervisor to provide lifetime kernel code integrity for commodity oses, 2007. 4

[SP07] Besnik Seljimi and Ioannis Parissis. Automatic generation of test data generators for synchronous programs : Lutess v2. In *Workshop on Domain Specific Approaches to Software Test Automation : In Conjunction with the 6th ESEC/FSE Joint Meeting*, DOSTA '07, pages 8–12, New York, NY, USA, 2007. ACM. 56

[SP10] Matt Staats and Corina Păsăreanu. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 183–194, New York, NY, USA, 2010. ACM. 33

[TWV08] H. Tews, T. Weber, and M. Volp. A formal model of memory peculiarities for the verification of low-level operating-system code. In *Proc. 3rd Int. WS on Systems Software Verification (SSV'08), volume 217 of ENTCS*, pages 79–96. Elsevier, 2008. 4

[UKR92] E. Schubert U. Kebschull and W. Rosenstiel. Multilevel logic synthesis based on functional decision diagrams. *European Design Automation Conference*, 38 :16–19, 1992. 42

[Uni98] Berkeley University. Berkeley logic interchange format (blif). 1998. 22, 29, 82

[Was04] Andrzej Wasowski. Flattening statecharts without explosions. *SIGPLAN Not.*, 39(7) :257–266, Jun 2004. 66

[Wik12] S. Wikipedia. *Problème Np-Complet : Problème Du Sac À Dos, Problème de la Couverture Exacte, Problème SAT, Tetris, Liste de Problèmes NP-Complets, Coloration de Grap*. General Books LLC, 2012. 45

[Xu11] Dianxiang Xu. A tool for automated test code generation from high-level petri nets. In *Proceedings of the 32Nd International Conference on Applications and Theory of Petri Nets*, PETRI NETS'11, pages 308–317, Berlin, Heidelberg, 2011. Springer-Verlag. 58

[YB04] P.Castéran Y. Bertot. Coq'art : The calculus of inductive constructions, 2004. 5

[Zah80] J. Zahnd. *Machines Séquentielles*, volume XI of *Traité d'Electricité*. Editions Georgi, Suisse, 1980. 7, 30

[Zha97] Hantao Zhang. Sato : An efficient prepositional prover. In William McCune, editor, *International Conference on Automated Deduction(CADE)*, volume 1249 of *Lecture Notes in Computer Science*, pages 272–275. Springer Berlin Heidelberg, 1997. 3

Un des plus grands défis dans la conception matérielle et logicielle est de s'assurer que le système soit exempt d'erreurs. La moindre erreur dans les systèmes embarqués réactifs peut avoir des conséquences désastreuses et coûteuses pour certains projets critiques, nécessitant parfois de gros investissements pour les corriger, ou même conduire à un échec spectaculaire et inattendu du système. Prévenir de tels phénomènes en identifiant tous les comportements critiques du système est une tâche assez délicate. Les tests en industrie sont globalement non exhaustifs, tandis que la vérification formelle souffre souvent du problème d'explosion combinatoire. Nous présentons dans ce contexte une nouvelle approche de génération exhaustive de jeux de test qui combine les principes du test industriel et de la vérification formelle académique. Notre approche construit un modèle générique du système étudié à partir de l'approche synchrone. Le principe est de se limiter à l'analyse locale des sous-espaces significatifs du modèle. L'objectif de notre approche est d'identifier et extraire les conditions préalables à l'exécution de chaque chemin du sous-espace étudie. Il s'agit ensuite de générer tout les cas de tests possibles à partir de ces pré-conditions. Notre approche présente un algorithme de quasi-aplatissement plus simple et efficace que les techniques existantes ainsi qu'une compilation avantageuse favorisant une réduction considérable du problème de l'explosion de l'espace d'états. Elle présente également une manipulation symbolique des données numériques permettant un test plus expressif et concret du système étudié. Nous avons implémenté notre approche dans un outil appelé GAJE. Afin d'illustrer notre travail, cet outil a été appliqué pour vérifier un projet industriel portant sur la sécurité et la vérification d'une carte à puce sans contact.

Mots clés : Jeux de test, Approche synchrone, Modèle synchrone, Pré-conditions, Couverture de l'espace d'états, Manipulation numérique des données, Backtrack, GAJE.

One of the biggest challenges in hardware and software design is to ensure that a system is error-free. Small errors in reactive embedded systems can have disastrous and costly consequences for a project. Preventing such errors by identifying the most probable cases of erratic system behavior is quite challenging. Indeed, tests in industry are overall non-exhaustive, while formal verification in scientific research often suffers from combinatorial explosion problem.We present in this context a new approach for generating exhaustive test sets that combines the underlying principles of the industrial test technique and the academic-based formal verification approach. Our approach builds a generic model of the system under test according to the synchronous approach. The goal is to identify the optimal preconditions for restricting the state space of the model such that test generation can take place on significant subspaces only. So, possible all the test sets are generated from the extracted subspace preconditions. Our approach exhibits a simpler and efficient quasi-flattening algorithm compared with existing techniques and a useful compiled internal description to check security properties and reduce the state space combinatorial explosion problem. It also provides a symbolic processing technique of numeric data that provides a more expressive and concrete test of the system.We have implemented our approach on a tool called GAJE. To illustrate our work, this tool was applied to verify an industrial project on contactless smart cards security.

Keywords : Test Sets, Synchronous Approach, Synchronous Model, Pre-conditions, States Space Covering, Numeric Data Processing, Backtrack, GAJE, Contactless Smart Card.