PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Étendre la spécification de programmes C concurrents et les vérifier par une transformation de source à source
Auteur: Author:	Guillaume Hétier
Date:	2018
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Hétier, G. (2018). Étendre la spécification de programmes C concurrents et les vérifier par une transformation de source à source [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/2965/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/2965/
Directeurs de recherche: Advisors:	Hanifa Boucheneb
Programme: Program:	Génie informatique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

UNIVERSITÉ DE MONTRÉAL

ÉTENDRE LA SPÉCIFICATION DE PROGRAMMES C CONCURRENTS ET LES VÉRIFIER PAR UNE TRANSFORMATION DE SOURCE À SOURCE

GUILLAUME HÉTIER
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
JANVIER 2018

© Guillaume Hétier, 2018.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ÉTENDRE LA SPÉCIFICATION DE PROGRAMMES C CONCURRENTS ET LES VÉRIFIER PAR UNE TRANSFORMATION DE SOURCE À SOURCE

présenté par : HÉTIER Guillaume
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

Mme NICOLESCU Gabriela, Doctorat, présidente
Mme BOUCHENEB Hanifa, Doctorat, membre et directrice de recherche
M. KHOMH Foutse, Ph. D., membre

DÉDICACE

À Anne, Aude, Adrien, Alexandre et Thomas,
pour votre soutien durant cette maîtrise,
et les bons moments passés ensemble.

REMERCIEMENTS

Je tiens tout d'abord à remercier ma directrice de recherche, Hanifa Boucheneb, sans qui ce travail n'aurait pas été possible. Ses conseils et ses remarques ont constitué une aide précieuse. Un grand merci aussi à Amira Aouina, Laila Boumlik ainsi que tous mes collègues et voisins de laboratoires pour vos suggestions et le partage de votre expérience. Ils m'ont souvent permis d'avancer et de progresser dans mes recherches. Je remercie enfin mes amis, ceux que j'ai rencontrés à Montréal ou ceux qui m'ont soutenu de loin, ainsi que ma famille. Vous m'avez aidé à toujours aller de l'avant.

RÉSUMÉ

L'utilisation croissante de systèmes informatiques critiques et l'augmentation de leur complexité rendent leur bon fonctionnement essentiel. Le model-checking logiciel permet de prouver formellement l'absence d'erreurs dans un programme. Il reste cependant limité par deux facteurs : l'explosion combinatoire et la capacité à spécifier le comportement correct d'un programme. Ces deux problèmes sont amplifiés dans le cas de programmes concurrents, à cause des différents entrelacements possibles entre les fils d'exécutions. Les assertions et les formules logiques LTL sont les deux formalismes de spécification les plus utilisés dans le cadre du model-checking logiciel. Cependant, ils sont limités dans un contexte concurrent : les assertions ne permettent pas d'exprimer des relations temporelles entre différents fils d'exécutions alors que les formules LTL utilisées par les outils actuels ne permettent pas d'exprimer des propriétés sur les variables locales et les positions dans le code du programme. Ces notions sont pourtant importantes dans le cas de programmes concurrents.

Dans ce mémoire, nous établissons un formalisme de spécification visant à corriger ces limitations. Ce formalisme englobe LTL et les assertions, en permettant d'exprimer des relations temporelles sur des propositions faisant intervenir les variables locales et globales d'un programme ainsi que les positions dans le code source. Nous présentons aussi un outil permettant de vérifier une spécification exprimée dans ce formalisme dans le cas d'un programme concurrent codé en C.

Notre formalisme se base sur la logique LTL. Il permet de surmonter deux des principales limitations rencontrées par les variantes de LTL utilisées par les outils de model-checking logiciel : manipuler des positions dans le code et des variables locales dans les propositions atomiques. La principale difficulté est de définir correctement dans l'ensemble du programme une proposition atomique lorsqu'elle dépend d'une variable locale. En effet, cette variable locale n'a de sens que dans une partie limitée du programme. Nous résolvons ce problème en utilisant le concept de *zones de validité*. Une zone de validité est un intervalle de positions dans le code source du programme dans laquelle la valeur d'une proposition atomique est définie à l'aide de sa fonction d'évaluation. Une valeur par défaut est utilisée hors de la zone de validité. Ceci permet de limiter l'utilisation de la fonction d'évaluation aux contextes où tous ses paramètres locaux sont définis.

Nous avons développé un outil, `baProduct`, afin de vérifier un programme concurrent en C spécifié dans le formalisme proposé. Cet outil prend la forme d'une transformation de source à source, et délègue les tâches de vérifications à un model-checker utilisé en arrière-

plan. Ce choix permet de simplifier grandement le développement de l'outil, mais aussi de profiter des performances de model-checkers reconnus et de comparer leurs efficacités sur le code transformé. À partir du code source d'un programme et d'une spécification, `baProduct` produit un programme spécifié par des assertions dont la validité correspond au respect de la spécification initiale. Plusieurs model-checkers peuvent être utilisés en arrière-plan, la principale condition étant qu'ils supportent les spécifications sous forme d'assertions. La transformation de source à source se base sur l'implémentation en C de l'automate de Büchi associé à la spécification et la construction de son produit avec le code source du système.

Nous appliquons ensuite `baProduct` sur un ensemble de tests simples. Nous comparons les performances obtenues à l'aide de deux model-checkers utilisés en arrière-plan, CBMC et ESBMC. Nous obtenons généralement de meilleures performances avec CBMC en raison d'une différence de la gestion des entrelacements par ces outils. Ces tests nous permettent de confirmer qu'il est possible de spécifier des propriétés difficiles à exprimer par des assertions ou une version plus classique de LTL (comme des propriétés d'exclusion mutuelle) de manière efficace et sans qu'une instrumentation manuelle soit nécessaire. Nous vérifions aussi que notre transformation de source à source permet de ramener le problème de la vérification de propriétés LTL à la vérification d'assertions dans le cadre de traces finies. Cependant, nous mettons aussi en évidence un certain nombre de limitations. En particulier, notre transformation est limitée à des traces finies. De plus, notre outil n'implémente pas de méthode de réduction, ce qui nuit à ses performances.

ABSTRACT

Critical programs and IT systems are more and more broadly used. Simultaneously, their complexity increase. More than ever, it is crucial to ensure their correctness. Software model-checking is a verification technique that allows to formally prove the absence of errors in a program. However, it faces two main issues: combinatorial explosion and the ability to specify the correct behavior of a program. These issues are amplified for concurrent programs because of the interleaving between threads. Assertions and LTL formulas are the most used specification formalism for software model-checking. However they are restricted in a concurrent context. On the one hand, it is not possible to express temporal relations between threads or events of the program using only assertions. On the other hand, LTL formulas that are supported by the main software model-checkers does not allow to use local variables and program locations, whereas program locations are often a convenient way to check the synchronization between threads.

In this report, we establish a specification formalism aiming to overcome these issues. This formalism is a superset of both LTL and assertions. It allows to express propositions that use global and local variables and propositions, and to build temporal relations on these propositions. Then, we introduce a tool allowing to check for the correctness of concurrent C programs specified with the formalism we introduce.

Our formalism is based on the LTL logic. It tackles the problems of code location manipulation in specification and the use of local variable in atomic propositions. The main difficulty is to properly define an atomic proposition in the whole program when it depends on a local variable, as the local variable is defined only in part of the program. We solve this issue using the concept of *validity area*. A validity area is an interval of code locations in which the value of an atomic proposition is computed using its evaluation function. A default value is used out of the validity area. Hence, it is possible to limit the use of the evaluation function to the lexical contexts where all local parameters are defined.

We developed a tool named `baProduct` to verify concurrent C programs specified using our formalism. This tool implements a source to source transformation and calls backend model-checkers to perform verification tasks. This architecture greatly simplifies the development of the tool and supports a variety of model-checkers in backend. It allows us to take advantage of the optimization of state-of-the-art model-checkers and to compare their performances when used in combination with our tool. We base the source to source transformation upon the implementation in C of the Büchi automaton associated with the specification. We build

the product of this automaton with the source code of the system.

Afterward, we apply `baProduct` on a set of tests. We evaluate its performances using two model-checkers as backend, CBMC and ESBMC. We get better performances with CBMC due to difference in the way these tools handle concurrency. The tests confirm it is possible to conveniently specify properties (such as mutual exclusion properties) that are difficult to express using only assertions or a more classical version of LTL, without manual modifications of the source code. We also demonstrate that our transformation successfully reduces the problem of verification of LTL properties to the verification of assertions for finite traces. However, our tool faces some limitations. It is currently restricted to finite traces. Moreover, this version does not implement any reduction method, which reduces its performances.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

BDD	Binary Decision Diagram
BMC	Bounded Model Checking
CEGAR	Counter-Example Guided Abstraction Refinement
CTL	Computation Tree Logic
JSON	JavaScript Object Notation
LTL	Linear Temporal Logic
POSIX	Portable Operating System Interface
SAT	Propositional Satisfiability Problem
SMT	SAT Modulo Theory
SSA	Single Static Assignment

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Les systèmes informatiques ont pris une place sans précédent dans la société actuelle. D'une part, ils sont devenus omniprésents à l'échelle personnelle et dans le monde économique. Les smartphones et les ordinateurs personnels sont entrés dans notre quotidien. L'informatique embarquée est utilisée massivement, que ce soit dans les véhicules ou les usines. À travers internet, nous interagissons en permanence avec des programmes sur des serveurs distants.

Simultanément, le champ d'action des systèmes informatiques s'est étendu. En plus de leur rôle dans les systèmes embarqués et les communications, les logiciels ont maintenant un large accès aux données personnelles de leurs utilisateurs. À travers la domotique, ils peuvent agir sur notre environnement immédiat. Les systèmes de recommandation et d'intelligence artificielle influencent les prises de décisions.

Dans ce contexte, les dysfonctionnements des systèmes informatiques sont un sujet de préoccupation. Une erreur peut avoir des conséquences de grande ampleur, un coût important et porter atteinte à la sécurité des informations et des personnes. Il est cependant difficile de concevoir un système correct. On peut s'en convaincre en consultant la liste des bugs et des failles informatiques apparus ces dernières années. Le site [25] présente certains des dysfonctionnements ayant eu les conséquences les plus importantes. Les exemples les plus connus restent sans doute l'explosion d'Ariane 5, due à une taille de variable sous-dimensionnée, et la faille HearthBleed dans la bibliothèque de cryptographie OpenSSL.

Vérifier les systèmes informatiques a ainsi gagné une importance qui n'est plus discutable. Cependant, l'inspection manuelle des logiciels est une tâche longue, coûteuse et souvent source d'erreurs. Sur les systèmes les plus complexes, elle devient irréalisable. Il est donc nécessaire de se reposer sur des techniques et des outils permettant de faciliter le processus de vérification.

On peut séparer les techniques et les outils utilisés pour vérifier un logiciel selon plusieurs critères. L'un de ces critères est le degré d'automatisation. Les outils peuvent aller d'un système totalement autonome à un simple assistant pour l'ingénieur responsable de la tache de vérification. L'automatisation est un facteur important pour la vérification de systèmes de grande taille. Le risque d'une erreur humaine dans un processus fortement automatisé est aussi réduit. Un second critère est la rigueur du verdict fourni par la technique de vérification. Certaines méthodes (les méthodes formelles) permettent d'obtenir une preuve de validité tandis que d'autres (comme le test ou certaines méthodes d'analyse statiques) permettent seulement d'augmenter son degré de confiance en la validité du programme.

Dans ce mémoire, nous allons nous intéresser aux techniques de vérification automatisées et fournissant une garantie de qualité rigoureuse du programme à vérifier. Nous allons en particulier nous concentrer sur les techniques de model-checking. Nous allons tout d'abord rapidement présenter quelques-unes des méthodes de vérifications existantes afin de dresser le contexte dans lequel le model-checking s'inscrit.

1.1 Définitions et concepts de base

1.1.1 Le test

La méthode de vérification la plus employée est de loin le test. Un programme ou un système informatique a généralement pour objectif de réaliser une tâche précise, en respectant des critères de qualité et de performances. Un test permet, étant donné un environnement et des paramètres d'entrée pour le programme, de vérifier s'il produit le résultat attendu (indiqué par un oracle).

La grande majorité des projets de développement maintiennent ainsi des suites de tests afin de s'assurer que le programme se comporte de la manière attendue. Différents paramètres et scénarios d'exécution sont testés afin de détecter autant d'erreurs que possible.

Ces suites de tests sont gérées de manière automatisée et sont étendues tout au long du développement d'un logiciel. Ils peuvent prendre des dimensions considérables. Par exemple, dans le cas de SQLite, les suites de tests représentent près de cent millions de lignes, soit plus de 700 fois le nombre de lignes de la bibliothèque elle-même[3].

Cependant, les tests souffrent d'un certain nombre de défauts. Tout d'abord, il est nécessaire d'avoir une bonne connaissance du système et de faire preuve d'une certaine imagination afin de concevoir les scénarios susceptibles de mener à une erreur. Lorsque c'est le cas, il est souvent complexe de remonter jusqu'à la source de l'erreur afin de la corriger. Les tests rencontrent aussi un problème de couverture : il est extrêmement difficile de produire un test pour chaque comportement du système. Ce problème est accru si l'on passe à l'échelle : le nombre de comportements d'un système augmente exponentiellement avec sa taille, le nombre de tests nécessaire pour couvrir l'ensemble des comportements devrait donc faire de même. Créer, maintenir et corriger les tests et les oracles représente alors un investissement conséquent.

Les tests assurent donc généralement une couverture partielle du programme à vérifier. Une suite de tests ne permet donc pas de garantir l'absence d'erreurs, mais seulement l'absence d'erreurs dans les chemins d'exécutions testés.

Différents critères de couvertures existent afin de mesurer la qualité d'une suite de tests. On peut demander que toutes les fonctions du programme soient exécutées au moins une fois lors des tests (on parle alors de *function coverage*), ou que l'ensemble des instructions du programme soit exécuté (*statement coverage*). On peut aussi demander à une suite de tests de contenir des exécutions sollicitant successivement différentes instructions du programme.

Les tests sont généralement moins performants dans le cas de programmes concurrents. L'entrelacement entre les instructions peut avoir un impact sur l'issue du test, et cet ordre change de manières non déterministes entre les exécutions. Une erreur peut alors être détectée lors d'une exécution des tests, mais être très difficile à reproduire. Pour tenter de détecter plus efficacement les erreurs liées à la concurrence, des techniques comme le stress-testing ont été mises en place. Elles consistent à ajouter du bruit à l'exécution des tests ou à lancer plusieurs instances du test en parallèle afin d'augmenter la probabilité d'occurrence des entrelacements d'instructions les moins fréquents.

Malgré ces limitations, le test reste généralement la méthode de vérification la plus simple à mettre en place. Les autres techniques de vérification sont parfois utilisées de manière complémentaire à une suite de tests, afin d'apporter une preuve de correction plus rigoureuse sur les fragments critiques du programme.

1.1.2 Les méthodes formelles

Les méthodes de test permettent de trouver des erreurs dans un programme et d'augmenter la confiance des développeurs dans le fait que le programme est correct, mais elles ne permettent pas de prouver l'absence d'erreurs. Les méthodes de vérification formelle sont une réponse à cette limitation : elles permettent de prouver l'absence d'erreurs dans un programme, quels que soient les entrées et les chemins d'exécution.

Les erreurs que l'on va rechercher dans un programme peuvent être séparées en trois catégories. Les erreurs de syntaxes correspondent à l'écriture d'un programme invalide et sont détectées lors de la compilation, nous n'allons donc pas nous y intéresser. Les erreurs d'exécution sont des erreurs qui ont lieu lorsque le programme s'exécute et qui sont liées à son environnement. Elles peuvent provoquer l'arrêt du programme ou un résultat faux. Elles dépendent souvent des entrées du programme ou des comportements indéterminés d'un langage de programmation (division par zéro, déréférencement d'un pointeur invalide...). Les erreurs de logique ont lieu lorsqu'un programme ne fournit pas le résultat attendu sans qu'une erreur d'exécution soit présente : il s'agit donc d'un programme correct, mais ce n'est pas celui que l'on voulait concevoir.

En pratique, vérifier qu'un système est complètement correct (c.-à-d il ne contient pas d'erreurs d'exécution ou de logique) est généralement trop complexe. Il est cependant possible de garantir l'absence de certains types d'erreurs et de vérifier des propriétés critiques pour le système.

De nombreuses méthodes existent afin de prouver la correction d'un programme vis-à-vis d'une propriété. Nous allons ici nous intéresser aux méthodes automatiques, qui ne requièrent pas (ou peu) d'interaction humaine pour fournir un résultat. En particulier, nous excluons les méthodes basées sur des assistants de preuve (comme Coq ou HOL), qui demandent un effort significatif et des connaissances avancées à l'utilisateur afin de construire une preuve.

L'interprétation abstraite

L'interprétation abstraite fait partie du domaine de l'analyse statique. L'analyse statique regroupe les techniques permettant d'obtenir des informations sur le comportement d'un programme à partir de son code source, sans l'exécuter.

Le théorème de Rice indique que la plupart des propriétés d'un programme ne sont pas décidables. L'essence de l'interprétation abstraite est de contourner ce problème en construisant une approximation du résultat de manière efficace et correcte. Le principal enjeu de l'interprétation abstraite est d'être suffisamment précise afin de ne pas indiquer un trop grand nombre de fausses (*spurious*) erreurs.

Plutôt que d'essayer de suivre l'ensemble des valeurs possibles pour chaque variable, à chaque point du programme, l'interprétation abstraite utilise un domaine abstrait qui permet d'approximer l'ensemble des valeurs qu'une variable peut prendre. Par exemple, on peut approximer l'ensemble des valeurs d'une variable par le plus petit intervalle contenant l'ensemble de ces valeurs. On va alors interpréter le programme en transposant chaque opération concrète dans le domaine abstrait. L'utilisation des domaines abstraits permet de réduire largement le nombre de valeurs à manipuler et facilite les opérations.

On peut alors prouver la correction du programme vis-à-vis d'une propriété en l'évaluant dans le domaine abstrait. Pour poursuivre l'exemple précédent, supposons que l'on veut s'assurer qu'une variable n'est jamais nulle à un certain point du programme (pour éviter une erreur arithmétique par exemple). Si l'analyse permet d'établir qu'elle se situe dans l'intervalle $[1, 5]$, on peut conclure que la propriété est vérifiée. Cependant, si l'on obtient l'intervalle $[-1, 1]$, il n'est pas possible de conclure : une erreur est possible, mais elle n'est pas certaine.

En pratique, l'interprétation abstraite est souvent utilisée pour vérifier des propriétés génériques, intégrées dans les outils (*built-in*). C'est une technique qui passe bien à l'échelle et qui

permet analyser de larges bases de code. Des propriétés typiquement vérifiées sont l'absence d'erreurs arithmétiques, l'absence de variables utilisées avant d'être initialisées, ou le respect des bornes des tableaux.

Cependant, l'interprétation abstraite souffre de deux principales limitations : tout d'abord, les approximations réalisées peuvent provoquer de nombreux faux positifs, signalant une erreur éventuelle dans un code correct. La précision de l'analyse peut parfois être améliorée en choisissant un domaine abstrait plus raffiné, mais cela a généralement un coût au niveau des performances. De plus, l'interprétation abstraite ne permet pas d'obtenir un contre-exemple menant vers une erreur indiquée, ou toute autre information indiquant la cause de cette erreur. Il peut alors être complexe de déterminer si une erreur reportée est un faux positif ou non, et de comprendre sa cause pour la corriger.

Le model-checking

Les techniques de model-checking se basent sur l'exploration d'un modèle du système à vérifier. Ce modèle prend généralement la forme d'un système de transitions, il peut être créé manuellement ou extrait automatiquement à partir du code ou d'une représentation de haut niveau du système. Les outils de model-checking vont alors explorer la partie accessible du modèle de manière exhaustive.

Le modèle doit à la fois représenter fidèlement le système vis-à-vis des propriétés à vérifier, et être suffisamment simple pour qu'il soit possible de l'explorer en un temps raisonnable. S'il est possible d'atteindre un état, ou d'emprunter une succession d'états ne respectant pas la propriété désirée, une erreur est reportée. Un contre-exemple — une trace d'exécution menant à cette erreur — peut alors être généré. Il a en général une grande valeur pour comprendre la source de l'erreur et la corriger, et constitue un des plus grands atouts du model-checking.

La distinction entre le model-checking et l'interprétation abstraite est principalement historique. L'analyse statique a été conçue pour vérifier des propriétés simples sur les programmes. Les algorithmes ont souvent mis l'accent sur les performances quitte à perdre en précision. D'autre part, le model-checking a été développé pour prouver des propriétés complexes sur le design de circuits. La priorité a été la précision des algorithmes et la capacité à vérifier des propriétés liées au programme. Cependant, les analyseurs statiques ont désormais gagné en précisions et sont capables vérifier des spécifications. En parallèle, les model-checkers ont développé des abstractions pour gagner en performances. La distinction reste donc principalement une question d'intention dans l'orientation de l'outil.

Le model-checking reste limité par le problème de *l'explosion combinatoire*. La taille des mo-

dèles et la durée des tâches de vérification augmentent généralement de manière exponentielle avec la complexité du système et des propriétés à vérifier.

1.1.3 Programmes concurrents

Les lois de Moore sont des conjectures empiriques qui prédisent que la puissance de calcul des processeurs double chaque année. Cependant, la miniaturisation des transistors approche ses limites physiques, annonçant la fin des lois de Moore dans leur interprétation traditionnelle. Cependant, afin continuer à gagner en puissance de calcul, les processeurs ont commencé à évoluer dans une autre direction : plutôt que d'augmenter la puissance, c'est le nombre de cœurs qui va croître. Pour tirer parti de ces processeurs multicœurs, il est cependant nécessaire de réaliser des programmes parallèles.

Plusieurs modèles de programmation concurrente existent. Par exemple, ils peuvent être basés sur des évènements (SystemC), ou sur des algorithmes de type map / reduce (MPI). Dans ce mémoire, nous allons nous concentrer sur les fils d'exécution suivant la norme POSIX (*pthread*). Dans ce modèle, le programme est constitué de plusieurs fils d'exécutions, ou *threads*, qui s'exécutent simultanément. Ils partagent un même espace mémoire, qu'ils peuvent utiliser pour communiquer, et ils sont coordonnés à l'aide de primitive de synchronisation comme les mutex et les sémaphores.

La concurrence introduit tout un nouveau panel de difficultés dans la conception d'un programme. Les accès à la mémoire partagée par plusieurs fils d'exécutions doivent être protégés pour éviter les *race conditions*, qui peuvent fausser le résultat d'une opération. Il est aussi nécessaire de synchroniser les fils d'exécution et de tenir compte des différents entrelacements possibles entre les instructions. Ces difficultés provoquent souvent l'apparition d'erreurs subtiles. En particulier, les erreurs dépendant de l'entrelacement des instructions sont souvent complexes à détecter et à diagnostiquer, car elles sont difficiles à reproduire : en effet, l'entrelacement des instructions lors d'une exécution est non déterministe.

Les programmes concurrents représentent un défi pour les méthodes de vérification. Les méthodes reposant sur l'exécution du programme, comme le test, n'ont pas de contrôle sur l'entrelacement des instructions. Il devient alors difficile de tester certains chemins critiques et de diagnostiquer les erreurs rencontrées. Les techniques de model-checking possèdent ce contrôle — elles ont initialement été conçues pour vérifier des systèmes électroniques concurrents. Cependant, dans le cas du model-checking logiciel, la concurrence amplifie l'explosion combinatoire de manière exponentielle. Il est nécessaire d'explorer tous les entrelacements entre les instructions d'un programme. Or, pour un programme composé de n threads, contenant k instructions chacun, il existe $\frac{(nk)!}{(k!)^n}$ entrelacements. Pour $k = 100$ instructions et $n = 2$

threads, on obtient déjà près de 10^{59} entrelacements !

1.2 Domaine d'étude

Dans ce mémoire, nous allons restreindre le champ de notre étude aux outils de model-checking pour des programmes écrits en C.

Le langage C est un langage bas niveau parmi les plus utilisés et les plus présents dans les bases de code actuelles. Il est en particulier utilisé dans les applications embarquées, les applications systèmes et les applications temps réel, qui nécessitent un contrôle bas niveau et des performances élevées. Ces types d'applications sont souvent critiques et ne peuvent tolérer un dysfonctionnement. Le C est donc un langage privilégié par les méthodes de vérification formelle, supporté par de nombreux outils de model-checking.

Quand il sera question de concurrence, nous nous intéresserons en particulier aux programmes multithreads suivant la norme POSIX, et utilisant la bibliothèque *pthread*. Elle implémente une gestion bas niveau de la concurrence, souvent utilisée dans les programmes embarqués ou les codes systèmes.

Deux questions ouvertes forment actuellement le cœur de la recherche sur le model-checking. La première, et la principale, est la question des performances et de la lutte contre l'explosion combinatoire. La seconde est la question de la spécification, les propriétés qu'il est possible de vérifier sur un système, et la manière de les exprimer. Dans ce mémoire, nous allons nous intéresser à ce second point.

Les assertions sont le mécanisme de spécification le plus utilisé dans le cadre de la vérification logicielle. Une assertion prend la forme d'une instruction dans le code, prenant en paramètre une expression booléenne pure — sans effet de bord — qui joue le rôle d'un invariant. Si cet invariant n'est pas respecté, l'assertion est déclenchée (ce qui met fin au programme dans le cas d'une exécution, ou permet de faire remonter une erreur dans le cas du model-checking). La plupart des outils de model-checking logiciel supportent une spécification sous la forme d'assertions.

Cependant, dans le cas de programmes multi threads, les assertions présentent des faiblesses. En particulier, l'invariant d'une assertion ne peut utiliser des valeurs hors de son contexte (une variable d'un autre thread par exemple). De plus, une assertion est bloquante : l'exécution ou l'exploration du programme stoppe une fois une assertion atteinte. Il n'est pas possible d'exprimer des relations temporelles (simultanéité ou succession par exemple) entre des assertions pour déclencher une erreur dans ces cas seulement.

Ma question de recherche sera donc composée des deux parties suivantes :

— Comment peut-on améliorer la spécification des programmes concurrents et la rendre plus expressive ?
— Comment peut-on rendre les outils de model-checking actuels compatibles avec une spécification plus expressive ?

Pour améliorer l'expressivité de la spécification des programmes concurrents, nous allons nous intéresser aux logiques temporelles, et en particulier aux formules LTL (Linear Temporal Logic). Ces dernières ont déjà été utilisées dans le cadre de model-checking logiciel, dans un cadre restreint. Nous proposons une extension de ce cadre, permettant en particulier de faire référence à des positions du code et à des variables locales dans les formules. En particulier, notre extension permet d'englober les assertions classiques et de créer des relations temporelles entre ces dernières.

Les outils de model-checking sont généralement complexes et optimisés pour le type de propriétés qu'ils prennent en charge. Afin de rendre les outils existants compatibles avec notre spécification, nous mettons en place une transformation de source à source du système. Nous construisons ainsi un nouveau programme spécifié par des assertions, dont la validité est équivalente à celle du système d'origine. Cette méthode nous permet en particulier de tester notre approche avec différents outils et de profiter de leurs optimisations.

1.3 Plan du mémoire

La partie 2 de ce mémoire consiste en un état de l'art. Nous présentons d'abord les spécifications à base d'assertions et la logique Linear Temporal Logic (LTL). Nous introduisons ensuite les principaux algorithmes de model-checking logiciels, ainsi que les outils les implémentant et leurs spécificités.

Dans la partie 3, nous détaillons un nouveau formalisme de spécification. Il constitue une extension de la logique LTL a des propositions portant sur les variables globales et locales et sur les positions dans le code du programme. Nous le comparons aux autres tentatives allant dans le même sens et nous mettons en lumière les principaux problèmes résolus ainsi qu'encore ouverts.

Dans la partie 4, nous présentons une transformation de source à source destinée à rendre les outils existants compatibles avec la spécification présentée dans la partie 3. Le but de cette transformation est de ramener la vérification d'une spécification, écrite dans le formalisme proposé, sur le code originel à la vérification d'assertions sur le code transformé. Nous évaluons ensuite les performances obtenues pour la vérification d'exemples simples.

CHAPITRE 2 REVUE DE LITTÉRATURE

Le *model-checking* rassemble des techniques permettant de vérifier et valider des systèmes. Historiquement, le model-checking a tout d'abord été utilisé pour la vérification de composants électroniques. Les techniques ont ensuite été adaptées afin de permettre la validation de programmes. On parle alors de model-checking logiciel. Nous allons par la suite nous intéresser à celui-ci uniquement.

Le principe du model-checking est de mener une recherche exhaustive à travers une modélisation du système plutôt que de construire une preuve formelle de validité. Alors qu'une telle preuve peut demander une certaine part d'intuition, le model-checking permet une approche plus systématique, qui peut être plus facilement automatisée.

De nombreux algorithmes de model-checking ont été développés en fonction des propriétés à vérifier sur le système, du formalisme dans lequel est décrit le système et de la complexité de celui-ci. Dans ce chapitre, nous présentons les principales techniques de model-checking logiciel et nous identifions leurs avantages et inconvénients respectifs.

Nous identifions les outils les plus performants implémentant chacune de ces techniques, à l'heure actuelle. Nous restreignons cependant cette étude aux outils qui nous intéresseront pour la suite de ce rapport. Ces outils sont donc susceptibles d'être utilisés en arrière-plan (backend) pour la vérification des codes sources considérés dans ce mémoire. Nous ne considérerons donc que des outils capables de vérifier des programmes concurrents en C, suivant le modèle d'exécution défini par la norme POSIX (pThread).

Structure : Dans la section 2.1, nous définissons le model-checking logiciel ainsi que les formalismes associés. Dans la section 2.2, nous présentons les types de propriétés des programmes vérifiables grâce au model-checking et les formalismes utilisés pour les exprimer et les représenter formellement. Enfin, dans la section 2.3, nous présentons différentes techniques utilisées pour le model-checking de programmes multi thread ainsi que les outils qui les implémentent.

2.1 Model-checking logiciel

Le model-checking permet de prouver qu'un modèle d'un système vérifie une spécification. Avant de nous intéresser davantage aux différentes techniques de model-checking, nous allons présenter les notions de modèle et de spécification. Nous définissions aussi un certain nombre de notions et termes techniques liés à cette problématique.

2.1.1 Modèle

Le point caractéristique du model-checking est l'utilisation de *modèles* des systèmes à vérifier. Un modèle est une représentation abstraite du système. Au lieu de vérifier un système directement, les techniques de model-checking s'appliquent sur le modèle du système. On considère un système comme correct lorsque son modèle ne contient pas d'erreurs.

L'origine des modèles est liée au domaine d'application initial du model-checking : la vérification du design de composants électroniques. Il était nécessaire de représenter ces derniers de manière abstraite afin de les vérifier, ce qui a donné naissance à la notion de modèles.

L'utilisation d'un modèle permet de simplifier les tâches de vérification : un modèle bien conçu capture le fonctionnement du système en retirant les détails n'ayant pas d'impact sur les propriétés à vérifier. Cette version simplifiée du système permet ainsi de réaliser des tâches de vérification complexes, qui ne pourraient aboutir autrement. Cependant, le modèle doit être suffisamment fidèle pour préserver les propriétés à vérifier. Un modèle trop simple pourrait provoquer l'apparition de faux positifs ou pire, ne pas contenir une erreur pourtant bien présente dans le système.

La phase de modélisation est donc une phase délicate et propice aux erreurs. Il est nécessaire de bâtir un équilibre entre l'abstraction du système et le respect de son fonctionnement. Cet équilibre dépend des propriétés que l'on souhaite vérifier. La modélisation demande donc une bonne connaissance du système et des outils de model-checking utilisés. Les tendances actuelles visent à réduire l'intervention humaine dans la phase de modélisation en automatisant celle-ci.

2.1.2 Système de transitions et trace d'exécution

Un modèle prend souvent la forme d'un système de transitions.

Système de transitions Formellement, un système de transitions est un triplet $(S, s_0, \rightarrow)$, avec :

- S l'ensemble des états du système ;
- $s_0 \in S$ l'état initial du système ;
- $\rightarrow \subset S \times S$ la relation de transitions du système ;

Intuitivement, les états d'un système de transitions représentent un statut possible du système, tandis que les transitions représentent les actions qui peuvent le faire évoluer.

Trace d'exécution Une trace d'exécution est un chemin dans le système de transition représentant le modèle, possiblement infini, dont le premier état est l'état initial du système de transition. Une trace d'exécution est donc formée d'une suite $s_0, s_1, .. \in S$ d'états, avec s_0 l'état initial, et d'une suite $t_0, t_1, ... \in \rightarrow$ de transitions telles que $\forall i, t_i = (s_i, s_{i+1})$.

Le model-checking consiste à examiner l'ensemble des traces d'exécution du modèle afin de vérifier si elles respectent la spécification.

2.1.3 Vocabulaire technique

Nous allons définir ici un certain nombre de termes liés au model-checking ou aux méthodes formelles de manière générale, que nous réutiliserons par la suite.

Faux positif / Faux négatif Lorsqu'un outil de vérification indique une erreur dans un système alors que ce dernier est en fait correct, on parle de *faux positif*. À l'inverse, si un outil reporte un système comme correct alors qu'il contient une erreur, on parle de *faux négatif*.

Complet / Correct Un outil ou une technique de vérification est complet s'il est conçu de sorte à ne jamais faire de faux négatif : si une erreur existe dans un système, elle sera toujours signalée. Un outil ou une technique de vérification est correct s'il est conçu de sorte à ne jamais faire de faux positif : si une erreur est signalée dans un système, elle est réellement présente dans ce système.

Concevoir un outil de model-checking logiciel à la fois complet et correct est extrêmement difficile, voire impossible, dans la plupart des cas. En effet, le théorème de Rice indique que toute propriété sémantique non triviale d'un programme est indécidable. Une technique de vérification ne respecte donc généralement qu'une seule de ces caractéristiques, ou aucune des deux.

2.1.4 Le cas du model-checking logiciel

Le model-checking logiciel, ou *software model-checking*, est un cas particulier de model-checking. Le système à vérifier est un programme, et le modèle va être extrait automatiquement à partir de son code source. On automatise ainsi la conception du modèle, ce qui présente les avantages suivants :

— on réduit le risque d'erreur lors de la modélisation du système, en supprimant l'intervention humaine ;

— on automatise davantage le procédé de vérification, ce qui le rend plus facilement utilisable en pratique.

Cependant, le code doit encore être compilé afin d'obtenir le système, ce qui constitue une encore une certaine distance entre le modèle et le système. Afin de réduire cette distance, certains model-checkers logiciels se basent sur des représentations intermédiaires plus proches de l'assembleur, comme la représentation intermédiaire de LLVM [1].

La programmation impérative consiste à considérer un programme comme un état que l'on fait évoluer à l'aide d'instructions. L'état du programme est principalement défini par l'état de la mémoire du programme. On peut alors modéliser un programme par un système de transition. L'ensemble des états du système de transition est une partition de l'ensemble des états du programme. Les états du système de transition sont donc caractérisés par :

— la configuration du tas du programme ;
— la valeur des variables globales du programme ;
— la configuration de la pile de chaque thread ;
— la valeur du pointeur d'instruction de chaque thread.

Les instructions permettant de faire évoluer l'état du programme sont représentées par des transitions entre les états de son modèle. Une transition représente plus précisément l'action d'une instruction atomique du programme sur les trois zones mémoires citées précédemment. Les transitions peuvent aussi dépendre de facteurs externes, comme les entrées du programme.

Dans le cadre du model-checking logiciel, deux hypothèses sont fréquemment faites sur le système :

— la consistance séquentielle. On considère alors que les écritures et les lectures mémoires ont lieu dans l'ordre où les instructions sont rencontrées dans une trace d'exécution. En pratique, cette hypothèse n'est pas toujours vérifiée. Les compilateurs et les processeurs peuvent modifier l'ordre de ces opérations pour optimiser un programme. On parle alors de modèle mémoire *faible*.
— la sémantique d'entrelacement. Dans le cas d'un système multi thread, on considère que les traces d'exécution possibles sont produites par un entrelacement des instructions atomiques de chaque thread. Cette hypothèse n'est pas valide si le programme est exécuté par plusieurs processeurs : il est alors possible d'exécuter plusieurs instructions simultanément, ce qui peut produire des traces d'exécution supplémentaires.

1. L'utilisation de cette représentation est aussi expliquée par des raisons de compatibilité. De nombreux langages, dont C, C++ et C# peuvent en effet être compilés vers la représentation intermédiaire de LLVM, un model-checker utilisant cette dernière est alors compatible avec tous ces langages.

Exemple Considérons le programme suivant, formé de deux threads.

Listing 2.1 Thread 1

```
a <- 1
a <- 2
```

Listing 2.2 Thread 2

```
c <- 1
d <- a + c
```

Ce programme contient trois variables globales, `a`, `c` et `d`, et deux threads. Les états du système sont donc définis par les valeurs de ces variables et la position des pointeurs d'instruction de chaque thread. On peut modéliser ce programme par le système de transitions en Figure 2.1. L'étiquette de chaque état représente la valeur des variables. Les transitions représentent l'exécution d'une instruction. On remarque l'existence de nombreuses traces d'exécution, pouvant mener à trois résultats différents.

2.1.5 Non-déterminisme

Un système est rarement complètement déterministe. De nombreux facteurs (les données en entrée du système, l'instant où un évènement a lieu, la réussite des allocations mémoires, la concurrence) tendent généralement à rendre l'évolution d'un système non déterministe.

Face à un choix non déterministe, les techniques de model-checking doivent explorer toutes les alternatives possibles.

Lors de la modélisation, certaines sources de non-déterminisme sont ignorées. Elles sont alors remplacées par une exécution déterministe. C'est généralement le cas pour les appels système ayant la possibilité d'échouer (les allocations de mémoire, par exemple).

Pour représenter les évènements et les paramètres extérieurs du programme, une fonction simulant le non-déterminisme est généralement mise à disposition de l'utilisateur par les outils de model-checking.

2.1.6 Explosion combinatoire

Le nombre d'états d'un programme augmente exponentiellement selon de nombreux paramètres (nombre de variables du programme, taille des types de données, degré de concurrence...). Il peut éventuellement être infini si des appels de fonctions ou de l'allocation dynamique de mémoire entrent en jeu. Cependant, afin de vérifier un système, un model-checker doit explorer l'ensemble de ces états. Cette tâche devient donc extrêmement coûteuse en temps et en mémoire quand le nombre d'états devient trop important. Ce problème, nommé l'explosion combinatoire, est la principale limite du model-checking.

Le non-déterminisme est une des principales raisons de l'explosion combinatoire : le nombre

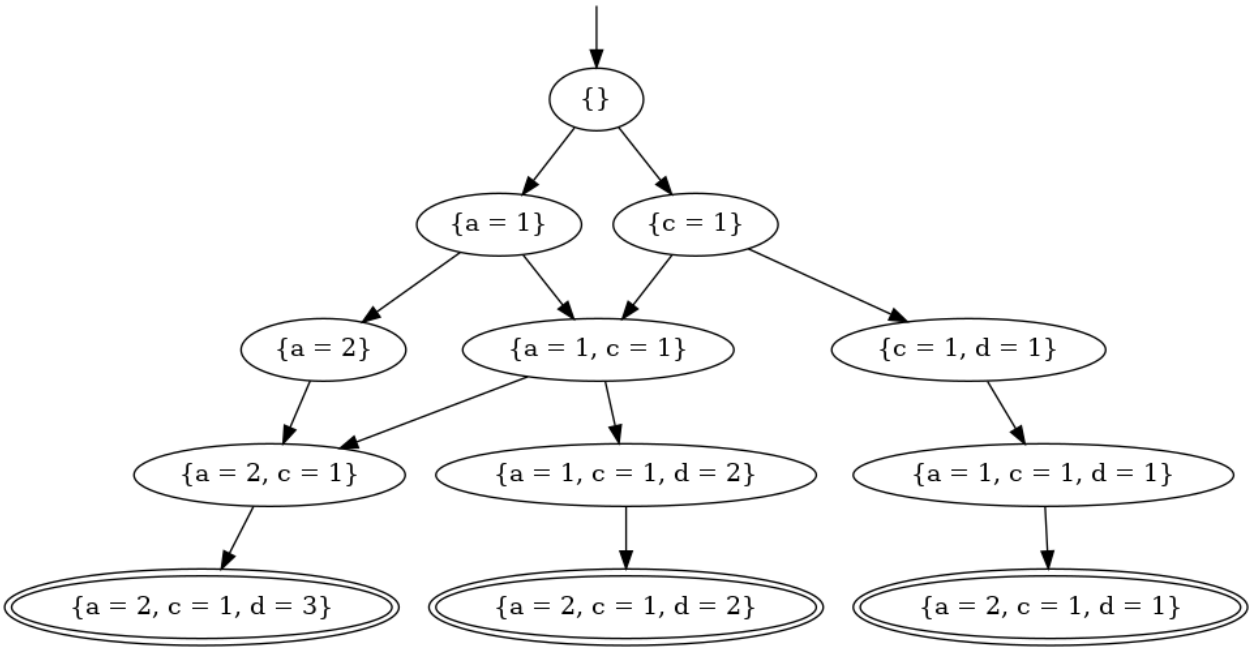

Figure 2.1 Système de transition modélisant un programme simple.

d'exécutions est exponentiel selon le nombre de choix non déterministe.

Cela explique les difficultés rencontrées par les techniques de model-checking face aux programmes concurrents : l'ordonnancement entre les instructions est non-déterministe. Il existe un nombre d'entrelacements exponentiel en fonction du nombre d'instructions du programme.

Les différents algorithmes de model-checking utilisent tous des techniques afin de limiter l'explosion combinatoire. Elle reste cependant le plus grand obstacle rencontré par le model-checking et limite le passage à l'échelle de la plupart des techniques.

2.2 Formalismes de spécification

Afin de vérifier un système, il faut tout d'abord établir ce que signifie être correct pour ce système. Un ingénieur responsable du design d'un système a généralement une connaissance informelle de la manière dont le système doit se comporter. Il est alors nécessaire de traduire cette connaissance d'une manière non ambiguë et compréhensible par des outils. On réalise pour cela une spécification du système.

Une spécification est constituée d'un ensemble de propriétés, qui représentent des invariants logiques que le système doit respecter. Si l'un de ces invariants est brisé, une erreur est

présente : le système ne se comporte pas de la manière attendue. Un outil de vérification peut alors inspecter un système afin de vérifier si l'ensemble de la spécification est respecté.

Une spécification peut prendre de nombreux aspects : il existe des langages de spécification (UML par exemple), elle peut être constituée de formules logiques portant sur les variables du système ou prendre la forme d'un système de transitions. Un langage de programmation constitue en lui même une spécification bas niveau du comportement d'un programme.

Les propriétés d'une spécification se répartissent en plusieurs catégories. Une propriété peut être un comportement généralement attendu pour le type de système concerné. Par exemple, on attend généralement d'un programme qu'il ne contienne pas de comportements indéterminés. Une propriété peut aussi être spécifique au système et à son comportement. On peut par exemple spécifier qu'une variable ne doit jamais dépasser un certain seuil.

2.2.1 Propriétés "built-in"

Certaines propriétés sont fortement — si ce n'est toujours — désirables pour une catégorie de systèmes. Un model-checker ciblant cette catégorie de systèmes peut alors implémenter la vérification de ces propriétés nativement, sans qu'il soit nécessaire de les spécifier. On parle alors de propriétés *built-in*.

Dans le cas du model-checking logiciel, des propriétés built-in classiques sont la vérification de la validité des pointeurs et des opérations arithmétiques, l'initialisation des variables et, de manière plus générale, l'absence de tout comportement ne respectant pas la norme du langage. Dans le cas de programmes concurrents, on cherche généralement à s'assurer de l'absence de dreadlocks (lorsque tous les threads du programme sont bloqués par une condition, et que le système ne peut plus évoluer) et de data-races (plusieurs accès simultanés à une même adresse mémoire, dont au moins un en écriture).

Les propriétés built-in sont souvent des propriétés qu'il serait difficile à l'utilisateur de spécifier, parce qu'elles impliquent la plupart des variables du système.

Pour vérifier des propriétés plus spécifiques au système (des propriétés fonctionnelles par exemple, qui portent sur le résultat que le système doit produire) il est cependant nécessaire de permettre à l'utilisateur d'exprimer lui-même la spécification.

2.2.2 Assertions

Dans un article de son blog, John Regehr aborde le sujet des assertions[48]. Il reprend en particulier la définition suivante :

> An assertion is a Boolean expression at a specific point in a program which will be true unless there is a bug in the program. [48]

Cette définition présente les assertions comme un mécanisme de spécification exécutable. Si une assertion n'est pas respectée, une erreur est présente dans le programme. Les assertions sont présentes dans la plupart des langages de programmation. Elles ont été naturellement reprises par les méthodes de vérification formelle, dont le model-checking logiciel.

Les assertions permettent de vérifier des propriétés d'accessibilité. Ces propriétés consistent à vérifier s'il est possible d'atteindre un état donné du modèle. Cet état est appelé un état d'erreur. On peut aussi considérer l'accessibilité d'un ensemble d'états d'erreurs.

Les propriétés d'accessibilité sont des propriétés relativement simples : déterminer si un état est un état d'erreur ne dépend que de cet état. Un programme contient donc une erreur dès qu'il est possible d'atteindre un état d'erreur, indépendamment des états et des transitions empruntées au cours de l'exécution. Une propriété d'accessibilité peut ainsi être déterminée par une exploration (en largeur par exemple) du système.

Dans un programme, une assertion prend la forme d'une instruction telle que : `assert(condition);`. Si elle est exécutée et que la condition est évaluée à vrai, alors l'exécution du programme continue normalement. Sinon, le programme a atteint un état d'erreur. Son comportement est alors indéterminé, bien que le comportement le plus fréquent soit de stopper son exécution.

Plus formellement, l'assertion `assert(c)` désigne comme étant des états d'erreur tous les états tels que :

- un des pointeurs d'instruction du programme pointe sur l'assertion
- l'expression `c` s'évalue à faux

Dans le code suivant, une assertion permet de spécifier que la variable `b` doit être non nulle. Dans le cas contraire, une erreur arithmétique (division par zéro) pourrait avoir lieu à la ligne suivante.

```
int int_div(int a, int b) {
    assert(b != 0);
    return a / b;
}
```

Les assertions sont très utilisées en raison de leur simplicité. Elles peuvent de plus être placées en tant que spécification pour un outil de vérification, ou permettre de signaler une erreur dans une version exécutable du programme (il ne faut cependant pas les confondre avec un

mécanisme de gestion d'erreurs).

Cependant, le pouvoir d'expression des assertions est limité. Toutes les propriétés d'accessibilité ne peuvent pas être exprimées par une assertion : il n'est par exemple pas possible d'utiliser dans la condition d'une assertion des variables hors du contexte courant. Il n'est pas non plus possible d'exprimer un problème d'exclusion mutuelle : il faudrait pour cela référer à la position des autres pointeurs d'exécution du code, ce qui n'est pas possible à l'aide d'une assertion.

2.2.3 Logique temporelle

Pour exprimer des propriétés plus complexes et en particulier des propriétés qui portent sur l'ensemble d'une trace d'exécution et l'ordre d'apparition de certains évènements, il est plus approprié d'utiliser une logique temporelle.

Supposons que notre système soit le code d'un distributeur automatique. On voudrait s'assurer que le distributeur ne délivre jamais le produit avant que le client ait payé, ce qui va revenir dans le code à vérifier que la fonction `livrer_produit` (qui commanderait au système de donner au client l'objet commandé) n'est jamais appelée avant la fonction `accepter_paiement` (qui validerait que le paiement a été correctement effectué), au cours d'une transaction. Des assertions permettent de déterminer si chacune de ces fonctions est atteinte, mais sans introduire des variables auxiliaires, il n'est pas possible de déterminer l'ordre des appels.

Les logiques temporelles permettent d'exprimer ce type de propriétés. On peut ainsi spécifier, pour un programme, des propriétés sur la succession des états d'une trace d'exécution.

Ci-dessous sont listés quelques-un des schémas de propriétés les plus courantes. Davantage sont présenté par [1]. Soit p une propriété portant sur les états du système.

- propriété de sûreté : tous les états atteints pendant l'exécution vérifient la propriété p.
- propriété d'accessibilité : en un temps fini, un état vérifiant la propriété p est atteint.
- propriété d'équité : on atteindra infiniment souvent un état vérifiant la propriété p.

Les logiques temporelles les plus utilisées sont : Linear Temporal Logic (LTL) et Computation Tree Logic (CTL).

LTL et CTL différent principalement par leur vision de l'ensemble des traces d'exécution. LTL considère chaque trace indépendamment. Un système est valide par rapport à une propriété LTL si toutes ses traces d'exécution respectent la propriété. CTL considère l'ensemble des traces comme un arbre et permet de quantifier universellement ou existentiellement sur les successeurs de chaque nœud.

Les pouvoirs d'expressivités de LTL et CTL ne sont ni équivalents, ni même comparables. En effet, LTL ne permet pas de quantifier existentiellement : une formule CTL utilisant une quantification existentielle n'a donc pas toujours d'équivalent en LTL. La réciproque peut être présentée à l'aide de la propriété LTL $F(p \wedge Xp)$ (pour toutes les traces, on atteint un état vérifiant p et dont le successeur vérifie p) n'a pas d'équivalent dans CTL. La formule CTL $AF(p \wedge AXp)$ (quelque soit le chemin emprunté, on atteint un état vérifiant p et dont tous les successeurs vérifient p) pourrait sembler un bon candidat, mais la Figure 2.2 présente un modèle vérifiant $F(p \wedge Xp)$ mais ne vérifiant pas $AF(p \wedge AXp)$.

CTL est considérée comme plus difficile à comprendre par les ingénieurs, plus habitués à penser à une unique exécution linéaire plutôt qu'à un arbre d'exécution[8]. CTL est par conséquent moins utilisée que LTL dans le cadre du model-checking logiciel. Par la suite, nous allons donc nous concentrer sur LTL uniquement.

2.2.4 LTL : Logique Temporelle Linéaire

La définition de LTL[46] ajoute deux opérateurs temporels à la logique classique, *next* (X) et *until* (U). La syntaxe d'une formule LTL est définie de la manière suivante, pour ϕ et ψ deux formules LTL :

$$\phi, \psi := \text{true} \mid \text{false} \mid \text{p} \mid \phi \wedge \psi \mid \neg\phi \mid X\phi \mid \psi U \phi$$

p est une proposition sur l'état du système. On nommera par la suite ces propriétés des *propositions atomiques*. On identifiera aussi une proposition atomique avec sa fonction d'évaluation, c'est-à-dire la fonction qui indique si un état du système vérifie la propriété ou non.

Étant donnée une trace d'exécution infinie $s = (s_0, s_1, ...)$, LTL a la sémantique suivante :

$$\begin{aligned} s \models p &\equiv s_0 \models p \\ s \models X\phi &\equiv (s_1, s_2, \dots) \models \phi \\ s \models \phi U \psi &\equiv \exists k, (s_k, s_{k+1}, \dots) \models \psi \wedge \forall i <= k, (s_i, s_{i+1}, \dots) \models \phi \end{aligned}$$

$\neg$, $\wedge$, $true$ et $false$ s'interprètent de la manière usuelle.

Une trace d'exécution est un modèle d'une proposition atomique si son premier état est un modèle de la proposition atomique. L'opérateur *next* signifie donc que la propriété LTL passée en paramètre doit être valide sur la trace privée de son premier état. L'opérateur *until* signifie que la première propriété passée en paramètre doit être vérifiée par tout les états de la trace jusqu'à ce qu'un état vérifie la seconde formule.

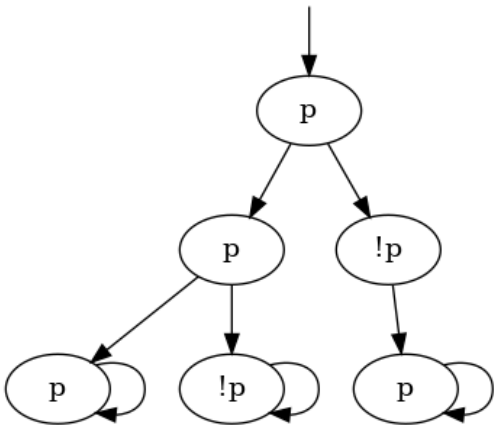

Figure 2.2 Modèle vérifiant $F(p \wedge Xp)$ mais pas $AF(p \wedge AXp)$.

À partir de ces opérateurs base, on définit les opérateurs $\vee$, $\Longrightarrow$, ... de la manière classique. On définit aussi les opérateurs temporels *always* (G), signifiant qu'une propriété est vraie pour tous les états d'une trace et *finally* (F), signifiant qu'un état vérifiant une propriété est atteint dans le futur.

$$Fp \equiv \text{true}Up$$
$$Gp \equiv \neg F(\neg p)$$

Enfin, on définit un système comme étant valide par rapport à une formule LTL si toutes les exécutions de ce système sont des modèles de la formule.

2.2.5 Automates de Büchi

Vérifier si un système respecte une proposition LTL demande d'être capable de manipuler efficacement ces dernières. On utilise pour cela les automates de Büchi. Ils permettent de représenter une propriété LTL. Sous cette forme, elles sont plus simples à manipuler pour un model-checker. Toute formule LTL peut être représentée par un automate de Büchi. Il existe des algorithmes pour construire efficacement et automatiquement cet automate[28].

Afin de vérifier si un système respecte une propriété LTL, une méthode classique est de construire un automate de Büchi qui représente la négation de cette formule LTL et d'explorer la composition entre cet automate et le système de transitions modélisant le système. Un chemin acceptant représente alors une exécution du système qui viole la propriété.

Automates de Büchi Un automate de Büchi est un automate qui accepte un langage de mots infinis. Formellement, un automate de Büchi est un quintuplé $B = (S, \Sigma, I, \delta, F)$, avec :

- S un ensemble d'états ;
- Σ un alphabet ;
- $I \subset S$ est un ensemble d'états initiaux ;
- $\delta \subset (S, \Sigma, S)$ est la relation de transition ;
- $F \subset S$ est l'ensemble des états finaux.

Un calcul (les mots chemin ou trace sont aussi utilisés) dans B est une suite infinie de transitions consécutives $c \in \delta^\omega$, dont l'état de départ est un état initial :

$$c = (s_0, a_0, s_1)(s_1, a_1, s_2) \dots (s_n, a_n, s_{n+1}) \dots$$

L'étiquette de ce chemin est le mot infini $a = (a_0, a_1, \dots, a_n) \in \Sigma^\omega$. Le chemin c est réussi si et seulement s'il passe une infinité de fois par un état final de B.

L'automate B accepte un mot a si et seulement s'il existe un calcul réussi dans B ayant le mot a pour étiquette.

Pour représenter une formule LTL, on prend $\Sigma = 2^P$, avec P l'ensemble des propositions atomiques de la formule. Une lettre de l'alphabet représente ainsi une configuration des propositions atomiques du système.

Exemples La figure 2.3 présente les automates de Büchi pour trois propriétés LTL simples : Gp, $F(p \vee q)$ et $G(Fp)$. Seule la partie accessible des automates est représentée, et les transitions ayant les mêmes sources et destinations sont fusionnées, leurs étiquettes étant remplacées par une garde sous la forme d'une expression logique.

Automate produit Soient un automate de Büchi $B = (S_B, \Sigma, I_B, \delta, F_B)$ représentant une propriété LTL ϕ et un système de transitions $T = (S_T, \rightarrow, I_B)$ modélisant un système.

L'automate produit de B et T est définit comme étant $P = (S_P, \Sigma, I_P, \delta_P, F_P)$, avec :

- $S_P = S_B \times S_T$
- $I_P = I_B \times I_T$
- $F_P = F_B \times S_T$
- $((p, q), s, (p', q')) \in \delta_P$ si et seulement si :
  - $(p, p') \in \rightarrow$
  - $(q, s, q') \in \lambda$

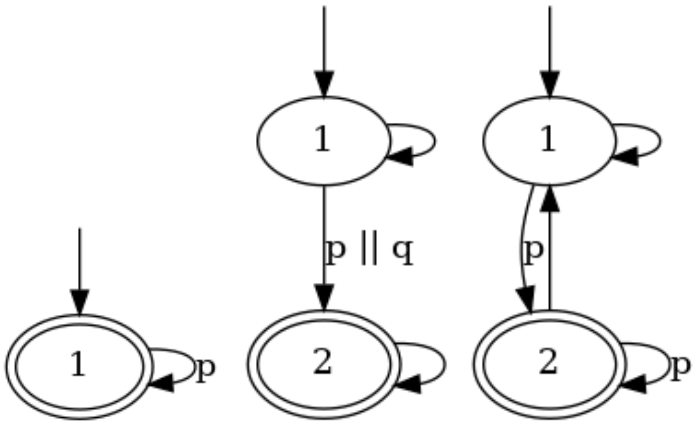

Figure 2.3 Automates de Büchi pour les formules LTL Gp, $F(p \vee q)$ et GFp.

— $p \models s$

Un mot est accepté par l'automate produit si et seulement s'il représente une exécution valide dans le modèle du système et qu'il appartient au langage de l'automate de Büchi. L'automate produit P reconnaît donc exactement le langage des exécutions du modèle T qui vérifient la propriété LTL représentée par l'automate de Büchi B.

Vérifier une propriété LTL Vérifier si le modèle T respecte la propriété ϕ, revient à calculer si le langage des exécutions valide de T est inclus dans le langage de l'automate de Büchi B. Cependant, vérifier une inclusion est une opération complexe. Il est plus simple de calculer si langage est vide ou non. On va donc reformuler le problème : on va chercher à déterminer si le langage L des exécutions valides dans T et ne respectant *pas* ϕ est vide. Si c'est le cas, le système est correct. Sinon, une erreur est présente et les éléments de L représentent des contre-exemples.

Pour calculer le langage L, on va construire le produit entre le système de transitions du modèle et l'automate de Büchi représentant la *négation* de ϕ. Le langage de l'automate produit est alors constitué des exécutions valides du modèle qui ne respectent pas ϕ, soit L.

Il suffit alors d'explorer l'automate produit. S'il est possible, à partir de l'état initial, d'atteindre un cycle contenant au moins un état final, alors, L n'est pas vide : il contient au moins le mot composé par l'étiquette du chemin allant jusqu'au cycle et d'une infinité de répétitions de l'étiquette du cycle. Ce mot correspond à une exécution du système qui viole

la propriété ϕ, et pourra servir de contre-exemple à l'utilisateur.

L'encodage des propriétés LTL à l'aide des automates de Büchi permet ainsi de vérifier des propriétés complexes sur les modèles. Cependant, la taille d'un automate de Büchi augmente exponentiellement avec la profondeur de la formule qu'il représente. Dans le cas de formules de grande taille, il peut donc devenir problématique de générer l'automate. Sa taille et son aspect non déterministe viennent renforcer le problème d'explosion combinatoire déjà rencontré par les techniques de model-checking.

2.3 Techniques et outils de model-checking

Différents algorithmes sont utilisés par les outils de model-checking afin d'explorer l'ensemble des états d'un modèle. Afin de lutter contre l'explosion combinatoire, ils établissent des compromis au niveau de la précision de la vérification et du type de propriétés qu'ils prennent en charge. Leurs performances dépendent aussi fortement de la structure du programme : les boucles, le non-déterminisme ou la gestion de la concurrence sont plus ou moins bien supportés selon les algorithmes.

À l'exception des techniques de séquentialisation, ces algorithmes ont été initialement appliqués à des programmes séquentiels. Le support des programmes concurrents est venu ensuite [2]. Ce dernier n'est pas forcément complexe, d'un point de vue théorique : un programme multi thread peut s'exprimer comme un système de transitions non déterministe. Les model-checkers sont capables de vérifier des modèles non déterministes — ils sont utilisés pour simuler les entrées possibles d'un système par exemple — les algorithmes de model-checking peuvent donc théoriquement être étendus. Le problème est bien plus complexe en pratique, en raison des contraintes de temps et de mémoire que rencontrent les model-checkers. En raison de l'explosion combinatoire provoquée par le nombre exponentiel d'entrelacements possibles entre les threads, il est difficile de concevoir un algorithme performant.

Nous présentons dans cette partie les principaux algorithmes utilisés dans le cadre du model-checking de logiciel multi thread, ainsi que certains des outils qui les implémentent. Pour chaque algorithme, nous avons choisi les outils qui nous ont parus les plus aboutis et ceux qui présentaient une approche originale. Pour chaque algorithme et outil, nous tentons de mettre en valeur ses points forts, ses faiblesses et ses spécificités.

2. Les algorithmes inspirés du model-checking de composants électroniques supportent des modèles concurrents depuis longtemps. Cependant, le support de la concurrence pour le model-checking logiciel a été ralenti par l'explosion combinatoire.

2.3.1 Model-checking explicite

Le model-checking explicite consiste à énumérer individuellement les états accessibles du modèle afin de les explorer. Le modèle prend la forme d'un système de transitions et son exploration est réalisée à l'aide d'algorithmes de graphes (exploration en largeur ou en profondeur...).

Les algorithmes de model-checking explicite modélisent le programme par un système de transitions. Ils construisent et explorent ce système de transitions à la volée : partant de l'état de départ, les successeurs sont déterminés à partir des instructions du programme. Ils sont alors visités à leur tour, leurs successeurs sont calculés et ainsi de suite. Pour ne pas explorer plusieurs fois un même état (l'exploration entrerait alors dans un cycle infini), il est nécessaire de stocker les états du chemin emprunté.

Le model-checking explicite est extrêmement vulnérable à l'explosion combinatoire. Les états accessibles du système sont examinés individuellement. S'ils sont en nombre infinis, une analyse peut ne pas terminer. Le stockage des états explorés peut aussi devenir problématique. Plusieurs techniques existent pour réduire l'impact de l'explosion combinatoire.

Stateless Model-checking Certains outils choisissent de ne pas maintenir la liste des états visités afin de ne pas être limités par l'utilisation de la mémoire. On parle alors de *stateless model-checking*. Ces outils ne sont pas capables de détecter un cycle dans le programme à analyser. Il est donc nécessaire que toutes les exécutions du programme à analyser terminent.

State Hashing Le *state hashing* consiste à ne conserver qu'une valeur de hachage des états visités, et non pas l'état complet. L'utilisation de la mémoire est ainsi réduite, et les performances peuvent être améliorées par l'utilisation de structures de données capables de rechercher un élément de manière efficace (ensemble ordonné, ...). Cependant, il est possible (bien que très peu probable) que deux états aient la même valeur de hachage. Ces états sont alors considérés comme égaux, ce qui peut mener à un résultat faux de l'analyse. Le state hashing rend donc les model-checkers incomplets.

Runtime model-checking Certains outils explorent l'espace d'états en se basant sur des exécutions réelles du programme. Le modèle est alors le programme lui-même. Son exécution est contrôlée par le model-checker. Ce dernier contrôle le non-déterminisme de chaque exécution. Cette approche permet de bénéficier des performances réelles du programme lors de l'exploration, cependant le backtracking est plus compliqué : les outils de runtime model-checking relancent en général une exécution du programme à tester depuis l'état initial chaque

fois qu'il est nécessaire de revenir à un état précédent. Il est aussi complexe de mémoriser les états explorés, cette technique est donc souvent combinée avec le *stateless model checking*.

Réduction par ordre partiel Les techniques de réduction par ordre partiel visent à réduire le nombre de chemins à explorer dans un système concurrent. Elles permettent de supprimer les chemins équivalents par rapport aux propriétés à vérifier, souvent en supprimant des changements de contextes entre des instructions indépendantes. Un cas simple de réduction consiste à regrouper en un bloc atomique une série d'instructions manipulant uniquement des variables locales. L'entrelacement de ces instructions avec d'autres threads n'a pas d'impact sur leur résultat, on peut donc conserver uniquement l'ordonnancement consistant à les exécuter sans changement de contexte.

La plupart des variantes ci-dessus sont aussi utilisées par les autres algorithmes que nous présentons par la suite, en particulier les techniques de réduction par ordre partiel. Cependant, le model-checking explicite est particulièrement dépendant de ces techniques afin de gagner en performances.

La principale force du model-checking explicite est sa précision. Explorer l'ensemble des états de manière explicite permet de vérifier la plupart des propriétés, spécifiées à l'aide d'assertions ou par une formule LTL. Cependant, il est extrêmement dépendant des techniques de réduction et a plus de difficultés face à des programmes de grande taille.

Outils

SPIN[33] est l'un des premiers projets de model-checking explicite. Il permet de vérifier des propriétés de la logique temporelle LTL sur des modèles exprimés en PROMELA. SPIN supporte nativement les modèles concurrents et implémente de nombreuses méthodes de réduction (ordre partiel, bit state hashing...). SPIN ne supporte pas le langage C nativement, cependant des travaux ont été menés afin de traduire C vers PROMELA[35], ce qui permet à SPIN de vérifier un programme en C de manière indirecte. Cependant, cette traduction ne supporte que partiellement les manipulations de la mémoire (pointeurs, allocation dynamique...).

Pancam[52] se base sur SPIN pour vérifier du bytecode LLVM [3]. Pancam exécute le bytecode LLVM dans une machine virtuelle et utilise SPIN comme un moniteur d'exécution afin de générer les différents entrelacements à explorer.

3. le bytecode LLVM peut être produit à partir d'un code C à l'aide d'un compilateur tel que CLANG ou GCC

CHESS[43] est un outil de runtime model-checking. Étant donné un scénario de test, il permet d'explorer l'ensemble des entrelacements à l'aide d'un ordonnanceur. Celui-ci a la particularité de ne pas autoriser un changement de contexte en tout point du programme : un changement de contexte est possible lorsque une primitive de synchronisation ou un point défini par l'utilisateur sont atteints ou lorsqu'un thread est dans une situation de livelock (son exécution boucle en attendant une action d'un autre thread). Il permet ainsi de limiter l'explosion combinatoire due à la concurrence.

`inspect`[51] vérifie du code C et C++ multi thread en se basant sur un algorithme de runtime model-checking avec une exploration *stateless*. `inspect` instrumente le programme à vérifier par des instructions lui permettant de communiquer avec un ordonnanceur, selon une architecture client/serveur. Le programme est ensuite exécuté pour chaque combinaison d'entrelacements, afin d'explorer l'ensemble des traces possibles.

Divine[6] est un model-checker capable de vérifier des propriétés LTL et des propriétés d'accessibilités. Il se base sur un langage interne, DVE. Il est capable de traiter du bytecode LLVM en le traduisant vers DVE, ce qui lui permet de supporter des langages comme C et C++. La particularité de Divine est de mettre en place une analyse concurrente et distribuée afin d'améliorer ses performances. Il utilise aussi des méthodes de réduction de l'espace d'état (compression de chemins, réduction par ordre partiel).

On retrouve à travers ces outils le besoin de performance des outils de model-checking explicite. Divine est implémenté dans une architecture concurrente pour améliorer ses performances, alors que `inspect` et Pancam se basent sur du runtime model-checking. SPIN, `inspect`, Pancam et Divine sont tous capables de vérifier des violations d'assertions et des propriétés built-in, comme les deadlocks et les data-races, cependant, seul Divine et SPIN sont capables de vérifier des propriétés LTL. `inspect` nécessite un système fermé, et n'est pas capable de gérer le non-déterminisme. Il est donc dépendant d'une suite de tests pour fixer les paramètres du programme à vérifier. Divine est arrivé sixième de la catégorie portant sur les programmes concurrents lors de l'édition de 2016 de la compétition de vérification logicielle SV-COMP[11] (les autres outils présentés ci-dessus n'ont pas participé à ces compétitions).

2.3.2 Model-checking symbolique

Les algorithmes de model-checking symbolique manipulent des ensembles d'états du modèle représentés de manière abstraite, plutôt que d'énumérer chaque état individuellement. Il est ainsi possible de manipuler des ensembles d'états importants ou infinis de manière efficace. Contrairement au model-checking explicite, le model-checking symbolique n'est donc pas limité à un modèle fini.

L'abstraction utilisée pour représenter les ensembles d'états est un élément clef du model-checking symbolique. Cette représentation doit permettre de réaliser les opérations classiques sur les ensembles (union, intersection, comparaison) de manière efficace. Les abstractions suivantes sont les plus utilisées :

— les BDD (*Binary Decision Diagrams*, Diagrammes de Décision Booléens) ;
— les formules de la logique propositionnelle ;
— automates finis.

Ces représentations permettent de définir des opérations efficaces sur les ensembles. L'opération la plus complexe est généralement la comparaison de deux ensembles, qui nécessite le choix d'une représentation canonique.

Cependant, la taille de la représentation d'un ensemble peut varier très fortement selon le choix de la représentation canonique. Le défi du model-checking symbolique est donc de construire et de maintenir efficacement une représentation canonique de taille raisonnable.

Représentation symbolique par des BDD

Nous allons utiliser l'exemple des BDD pour illustrer les forces et les défis rencontrés par les représentations symboliques.

Un BDD permet de représenter une fonction booléenne. On va donc représenter un ensemble d'états par une fonction prenant en paramètre un état (encodé par des variables booléennes) et indiquant s'il est dans l'ensemble représenté par la fonction ou non.

Encoder les états La première étape est d'encoder les états du système par des variables booléennes. Une manière simple de le faire dans le cas d'un ensemble fini d'états est de les numéroter, puis d'utiliser une variable booléenne pour chaque chiffre de la représentation binaire (il faut donc $log_2 n$ variables booléennes pour n états).

Prenons l'exemple d'un système à huit états. En numérotant les états de 0 à 7 et en prenant leur représentation binaire (sur trois bits), on obtient les représentations suivantes : (000, 001, 010, ..., 111).

Pour représenter l'ensemble composé des états 110 et 111, il suffit de prendre sa fonction caractéristique, c'est-à-dire la fonction qui s'évalue à vrai pour les éléments de l'ensemble et à faux sinon.

$$f\colon \{0,1\}^3 \to \{0,1\} \tag{2.1}$$

$$(x_1, x_2, x_3) \mapsto \begin{cases} 1 & \text{si } (x_1, x_2, x_3) \in \{(1,1,0),(1,1,1)\} \\ 0 & \text{sinon} \end{cases} \tag{2.2}$$

Nous allons ensuite construire un BDD pour représenter cette fonction.

La construction d'un BDD repose sur la propriété suivante :

Décomposition des fonctions booléennes Soit $k > 0$ et $f : \mathbf{B}^k \to \mathbf{B}$ une fonction booléenne. Alors il existe deux fonctions booléennes $g, h : \mathbf{B}^{k-1} \to \mathbf{B}$ telles que :

$$f(x_1, ..., x_k) = \begin{cases} g(x_2, ..., x_k) \text{ si } x_0 \text{ est vrai} \\ h(x_2, ..., x_k) \text{ si } x_0 \text{ est faux} \end{cases}$$

Il est ainsi possible de définir une fonction booléenne par récursion sur le nombre de ses variables. On va se servir de cette propriété pour représenter la fonction booléenne par un arbre binaire de décision. On fixe tout d'abord un ordre des variables (dans notre exemple, on prend le bit de poids le plus fort en premier). On construit alors l'arbre par induction :

- si la fonction est constante, on la représente par une feuille étiquetée par $vrai$ ou $faux$.
- sinon, on décompose f entre g et h selon la propriété. On construit alors l'arbre dont le fils gauche est la représentation de g et le fils droit celle de h (construites selon la même méthode). On étiquette l'arrête vers le fils gauche par 0 et celle vers le fils droit par 1.

Les étiquettes d'un chemin de la racine de l'arbre à une feuille représentent alors une valuation des variables de la fonction. L'étiquette de la feuille indique la valeur de la fonction pour cette valuation.

Pour notre exemple, nous obtenons l'arbre de la Figure 2.4.

La représentation par un arbre de décision contient $2^{n+1} - 1$ nœuds pour une fonction booléenne à n variables, on ne peut donc pas encore parler de représentation efficace. Cependant, cet arbre de décision contient aussi beaucoup de sous-arbres redondants. On va donc réduire cet arbre en un BDD en fusionnant tous les sous-arbres identiques et en supprimant les états n'ayant qu'un unique successeur. Dans notre exemple, on obtient alors le BDD de la Figure 2.5.

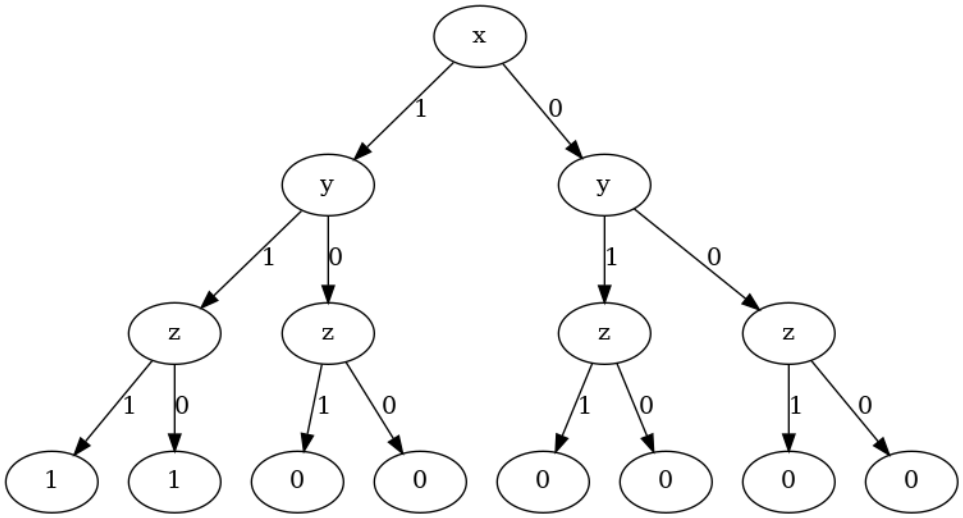

Figure 2.4 Arbre binaire de la fonction $(x, y, z) \mapsto x \wedge y$.

Cependant, la taille du BDD est très dépendante de l'ordre choisi pour les variables[16]. Dans cet exemple, l'ordre n'a pas une grande importance mais lorsque le nombre de variables et la complexité de la fonction augmentent, la taille du BDD peut exploser [4].

À partir de la représentation sous forme de BDD, on peut évaluer la fonction f efficacement, mais aussi réaliser des opérations logiques performantes, en construisant le BDD résultant à partir des feuilles et en réduisant au fur et à mesure. [16] explique cet algorithme plus en détail . Dans le cas (simple) de la négation, il suffit par exemple d'échanger les deux feuilles du BDD.

Vérification à l'aide des BDD On peut définir l'ensemble des états vérifiant une propriété Φ comme le point fixe d'une fonction (dépendant de Φ). On calcule alors ce point fixe de manière itérative, en utilisant les opérations définies sur les BDD. Si l'état initial du système fait partie des états vérifiant la propriété, le système est valide.

Par exemple, dans le cas d'une assertion, l'ensemble des états valides est l'ensemble des états à partir desquels il n'est pas possible de déclencher l'assertion. On peut le définir comme étant le point fixe de la fonction $X \mapsto E \cup X \cup Pre(X)$, avec X un ensemble d'états, E l'ensemble des états d'erreurs et Pre la fonction qui associe à un ensemble d'états ses prédécesseurs dans le modèle de système.

Tout comme les BDD, les autres représentations symboliques sont fortement dépendantes

4. Pour s'en convaincre, on peut calculer les BDD de la fonction $x, y, z, t \mapsto x.z + y.t$, avec les ordres $x < z < y < t$, puis $x < y < z < t$.

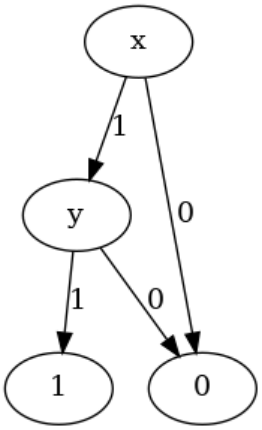

Figure 2.5 BDD de la fonction $(x, y, z) \mapsto x \wedge y$.

d'un ordre des variables afin de conserver une taille raisonnable. Les formules de la logique propositionnelle sont très utilisées actuellement en raison des progrès des solveurs SAT.

Les techniques de model-checking symbolique sont efficaces afin de prouver la validité d'une propriété (ce qui est le point faible du model-checking de manière générale). Elles supportent bien l'explosion combinatoire mais sont très dépendantes des performances des représentations symboliques utilisées.

Outils

CIVL[53] est un framework pour la vérification de programmes concurrents. Il définit un langage intermédiaire, CIVL-C, proche du C11, vers lequel plusieurs langages et types de concurrence peuvent être traduits (dont pThread, OpenMP et CUDA). CIVL utilise un mélange de model-checking explicite et symbolique : les différents entrelacements d'un programme sont explorés de manière explicite et une analyse symbolique est utilisée pour analyser chacun de ces entrelacements. Les états sont représentés à travers des formules SMT, manipulées à l'aide de la bibliothèque SARL[2]. CIVL a pour ambition de constituer un point de jonction commun entre plusieurs frontends et backends, afin de découpler les problèmes de modélisation d'un langage et les problèmes de vérification. CIVL adopte une approche conservatrice : il se veut correct mais il n'est pas complet, il peut donc signaler des faux positifs.

SymDivine[42] permet la vérification de propriétés LTL sur du bytecode LLVM concurrent (et donc, par extension, pour des programmes en C). Tout comme CIVL, SymDivine utilise une combinaison de model-checking explicite et symbolique. Le non-déterminisme au niveau des données est géré de manière symbolique tandis que celui au niveau du contrôle est traité

explicitement : tous les entrelacements sont explorés individuellement. SymDivine peut utiliser les formules SMT ou des BDD pour représenter les ensembles abstraits, les premières étant plus efficaces dans la plupart de leurs expérimentations.

SymDivine et CIVL utilisent des méthodes de vérification semblables, mais diffèrent par leurs langages de modélisation. CIVL utilise CIVL-C, un langage relativement haut niveau pour faciliter la conception de modèles et la prise en charge de plusieurs langages et paradigmes de concurrence. SymDivine privilégie le bytecode LLVM, plus proche du programme compilé final afin de réduire la distance entre le modèle et le système. CIVL a terminé troisième (respectivement quatrième) lors des éditions 2016 (respectivement 2017) de SV-COMP[11, 12], dans la catégorie portant sur la concurrence.

2.3.3 Predicate abstraction

L'abstraction par prédicat (*predicate abstraction*) est une technique proche de l'interprétation abstraite, dont nous avons parlé dans l'introduction de ce mémoire. Elle consiste à construire un modèle du programme dans un domaine abstrait, déterminé en partitionnant l'ensemble des états du système selon un ensemble de prédicats. Le domaine abstrait utilisé par l'interprétation abstraite est en général fixé par l'outil et indépendant du programme analysé. Les techniques d'abstraction par prédicat diffèrent sur ce point : elles construisent un modèle sur mesure pour le système considéré et la propriété à vérifier[29, 26].

L'objectif des techniques d'abstraction par prédicat est de construire une abstraction du programme exacte par rapport à la propriété à vérifier. Cette abstraction est réalisée en choisissant un certain nombre de prédicats portant sur les états du programme. Le choix des prédicats est une étape cruciale, puisque la précision de l'abstraction en dépend. Si l'abstraction n'est pas suffisamment précise, des faux positifs ou des faux négatifs peuvent avoir lieu.

Une fois les prédicats choisis, les états de l'abstraction sont définis comme les classes d'équivalence obtenues en partitionnant l'ensemble des états du système par l'ensemble des prédicats. Une transition existe entre un état abstrait A et un état abstrait B si une transition existe dans le système d'origine entre un des états de A et un des états de B.

On peut représenter le système abstrait comme un programme booléen, c'est-à-dire un programme possédant les structures de contrôle classique du C, mais uniquement des variables booléennes. Les variables de ce programme représentent chacune un prédicat, et leur valeur dans un état du programme représente si le prédicat est vérifié dans cet état ou non. On peut alors vérifier le système abstrait à l'aide d'un model-checker pour programmes booléens

(ceux-ci utilisent fréquemment des méthodes de model-checking symbolique).

Exemple Considérons l'exemple du listing 2.3. L'erreur qui a lieu lorsque `x = 0` pourrait ici correspondre à une division par zéro. Considérons les prédicats $p_1 = (y == 1)$ et $p_2 = (x == 0)$.

On détermine quatre états abstraits pour chaque position du programme, correspondant aux différentes valeurs de (p_1, p_2). On peut ensuite déduire le programme booléen associé en supprimant les variables du programme initial et en rajoutant des instructions pour actualiser les prédicats. On obtient le programme booléen présenté dans le listing 2.4.

Listing 2.3 Code initial

```
1 x = 3;
2 y = 2;
3 if (y == 1)
4     x = 0;
5 if (x==0)
6     assert(false);
```

Listing 2.4 Programme booléen

```
1 p_2 = false;
2 p_1 = false;
3 if (p_1)
4     p_2 = true;
5 if (p_2)
6     assert(false);
```

Ce programme peut facilement être analysé par un model-checker pour programmes booléens.

Il reste cependant à déterminer automatiquement les prédicats utilisés. La méthode la plus connue pour construire choisir les prédicats et construire une abstraction dans le cas du model-checking logiciel est la méthode Counter-Example Guided Abstraction Refinement (CEGAR)[26].

Counter-Example Guided Abstraction Refinement (CEGAR)

La méthode CEGAR permet de construire une abstraction en itérant quatre phases : Abstraction, Vérification, Simulation et Raffinement. Une première abstraction (peu précise) du programme est réalisée. Si la vérification fournit un contre-exemple, celui-ci est simulé dans le système initial afin de vérifier qu'il y existe (afin d'éviter un faux-positif). Si ce n'est pas le cas, on utilise cette information pour construire une abstraction plus précise qui permet de rejeter cette trace et on itère le processus.

Reprenons le programme du listing 2.3 comme exemple.

1) **Abstraction :** Dans un premier temps, on considère une abstraction basée sur un ensemble de prédicats vide. Il n'y a donc qu'une seule classe d'équivalence, contenant tous les états du programme. On ne conserve que les structures de contrôle du

programme. Une analyse statique du programme peut aussi être utilisée pour définir ce premier ensemble de prédicats. Toutes les variables sont rendues abstraites, et leurs valeurs dans les structures de contrôle sont non déterministes. On obtient alors l'abstraction présentée dans le listing 2.5.

2) **Vérification :** Un model-checker pour programmes booléens permet de trouver la trace exécutant les lignes 1, 2, 3, 5, et 6 qui mène à une erreur dans l'abstraction.
3) **Simulation :** Une simulation symbolique permet de constater que la trace n'est pas réalisable. En effet, `x` est non-nul lors du test à la ligne 5 donc la transition de la ligne 5 à la ligne 6 n'est pas réalisable.
4) **Raffinement :** L'incohérence présente dans la trace est utilisée pour raffiner l'abstraction, de manière à éliminer les chemins invalides. Ici, un prédicat `x = 0` indiquant si `x` est nul est ajouté. On obtient alors l'abstraction plus précise du listing 2.6.
5) **Vérification :** Le model-checker trouve cette fois l'exécution passant par les lignes 1, 2, 3, 4, 5 et 6.
6) **Simulation :** cette trace est encore une fois incorrecte, puisque `y` ne vaut pas `1` à la ligne 3.
7) **Raffinement :** un prédicat `y = 1` est ajouté. On obtient l'abstraction du listing 2.7.
8) **Vérification :** Le model-checker ne trouve pas de trace menant à une erreur dans l'abstraction. L'absence d'erreur est donc prouvée pour le programme.

Listing 2.5 Abstraction initiale

```
1 ;
2 ;
3 if (*)
4     ;
5 if (*)
6     assert(false);
```

Listing 2.6 Premier raffinement

```
1 {x = 0} = {false};
2 ;
3 if (*)
4     {x = 0} = true;
5 if ({x = 0})
6     assert(false);
```

Listing 2.7 Second raffinement

```
1 {x = 0} = false;
2 {y = 1} = false;
3 if ({y = 1})
4     {x = 0} = true;
5 if ({x = 0})
6     assert(false);
```

La phase de raffinement est critique pour les performances de la méthode CEGAR, puisque améliorer le choix des nouveaux prédicats peut éviter des itérations supplémentaires de l'algorithme. Des techniques d'analyse statique sont souvent utilisées pour trouver les prédicats les plus forts éliminant un chemin impossible du modèle. On peut en particulier calculer les plus faibles préconditions d'un chemin impossible — c'est-à-dire la plus faible propriété qui implique que ce chemin va être emprunté.

Lazy Abstraction with Interpolant (IMPACT)

Les méthodes d'abstraction par prédicat, dont l'algorithme CEGAR, se basent sur le calcul d'une ou plusieurs abstractions du modèle. Calculer cette abstraction, et en particulier les successeurs de chaque état abstrait est une tâche coûteuse.

L'algorithme IMPACT[39] permet d'éviter le calcul des successeurs des états abstraits. Au lieu de raffiner une abstraction de manière itérative, IMPACT raffine son modèle au fur et à mesure de sa construction, seulement lorsque nécessaire. Pour ce faire, il déroule la relation de transition d'un programme. L'arbre ainsi obtenu est ensuite exploré. Lorsqu'un état d'erreur est atteint, les états du chemin vers cette erreur sont annotés par des invariants prouvant que le chemin est impossible dans le programme d'origine (si de tels invariants n'existent pas, une erreur est reportée). Le cœur de l'algorithme IMPACT consiste à utiliser ces invariants pour éviter d'explorer des branches entières de l'arbre. Si deux nœuds A et B partagent la même location et que l'invariant de B est plus fort que celui de A (l'invariant de B implique l'invariant de A), alors il est inutile d'explorer les fils de B : une erreur accessible à partir de B est accessible à partir de A aussi.

Gestion de la concurrence

Les techniques d'abstraction par prédicat reposent sur le fait que le problème d'accessibilité dans un programme booléen séquentiel est décidable. On peut donc déterminer si une trace menant à une erreur existe dans le modèle abstrait. Cependant, le problème d'accessibilité dans un programme booléen concurrent n'est pas décidable. C'est une grande difficulté pour les méthodes d'abstraction par prédicats, qui reposent sur la terminaison et la rapidité de l'analyse du programme booléen. C'est plus vrai encore pour l'algorithme CEGAR, qui effectue plusieurs vérifications successives de ce programme.

Deux alternatives existent pour intégrer la concurrence dans les techniques d'abstraction par prédicat :

1) Considérer les entrelacements du programme lors de la vérification du programme

booléen. Pour pallier le problème de terminaison, il est possible de construire une sur approximation du programme booléen. Cette approximation permet d'assurer la terminaison du programme booléen, mais elle peut générer des faux positifs. Il reste possible de détecter ces faux positifs dans la partie *Simulation* de l'algorithme CEGAR[22]. Une autre solution est de limiter la profondeur d'exploration et le nombre de changements de contexte. Cependant, cette approche rend nécessairement la vérification incomplète.

2) Utiliser un raisonnement de type *rely-garantee*[32] : les threads sont examinés un par un, de manière indépendante, et une approximation de l'effet des autres threads est construite pour examiner les différentes traces possibles.

Outils

Impara[50] adapte l'algorithme *IMPACT*[39] et le combine à des techniques de réduction d'ordre partiel afin de l'étendre à des programmes multi thread. Il réduit ainsi l'explosion combinatoire présente dans l'algorithme IMPACT : dérouler naïvement la relation de transition d'un modèle concurrent, comme le fait une extension directe de l'algorithme IMPACT, revient à énumérer un nombre exponentiel d'entrelacements.

La plupart des model-checkers utilisant la méthode CEGAR basent leurs procédures de décision sur des prouveurs limités à l'arithmétique linéaire sur des nombres réels uniquement. Ils ne peuvent alors pas raisonner sur des tableaux de bits. Satabs[7][20] implémente ses procédures de décision à l'aide d'un solveur SAT. Il parvient ainsi à manipuler des tableaux de bits, et à représenter des structures plus complexes du langage C. Il gagne en précision sur l'arithmétique des pointeurs en particulier. Satabs est ainsi capable de vérifier un certain nombre de propriétés built-in (validité des pointeurs, erreurs arithmétiques, accès hors des bornes d'un tableau), en plus de propriétés spécifiées par des assertions.

Threader[47][31] se base aussi sur la méthode CEGAR. Cependant, il utilise des techniques basées sur des raisonnements de type *Rely-guarantee* afin de rechercher une preuve modulaire du programme. Threader gagne ainsi en performance lorsqu'une preuve modulaire existe, mais il peut être plus lent qu'une méthode classique dans les autres cas.

SKINK[18] permet vérifier des propriétés exprimées par des assertions à partir de la représentation intermédiaire de LLVM. SKINK se base sur une approche de type CEGAR, mais il raffine itérativement une abstraction de l'ensemble des traces d'exécution au lieu de travailler sur une abstraction des états. Les interpolants renvoyés par les solveurs SAT sont utilisés pour construire un automate qui vient restreindre le modèle. SKINK prends en charge la concurrence en construisant le produit entre les automates représentant chaque threads.

Des méthodes d'ordre partiel sont utilisées pour limiter l'explosion combinatoire.

CPAchecker[14] est un framework qui a pris la suite de l'outil BLAST [15]. Il permet une analyse configurable travers des CPAs (*Configurable program analysis*). Un CPA définit un domaine abstrait (BDD, analyse de valeurs, analyse d'intervalles...). CPAchecker est capable de mener une analyse d'accessibilité sur une combinaison arbitraire de CPAs. La concurrence est prise en charge à l'aide d'un CPA qui définit une exploration explicite des entrelacements[13]. De manière similaire à CIVL, CPAchecker vise à réduire les efforts de développement nécessaires pour tester de nouvelles approches pour les méthodes basées sur l'abstraction par prédicat. CPAchecker permet de construire une spécification sous la forme d'un automate, à l'aide d'un formalisme similaire à celui utilisé par BLAST.

Les techniques d'abstraction par prédicat sont principalement tournées vers les propriétés d'accessibilité spécifiées par des assertions. Parmi les outils supportant le C multi thread, tous prennent en charge les assertions. Cependant, seul Satabs est orienté vers la vérification de propriétés built-in et aucun ne permet de vérifier des propriétés LTL.

De manière générale, les techniques d'abstraction par prédicat sont très efficaces dans le cas du model-checking de logiciels séquentiels, car elles supportent bien l'explosion combinatoire liée aux données. Cependant, elles ont plus de difficultés à traiter des structures de données complexes (tableaux et allocations dans le tas). La vérification de programmes concurrents constitue aussi une difficulté majeure.

2.3.4 Model-checking borné

Le model-checking borné, ou Bounded Model Checking (BMC), est l'une des techniques les plus utilisées dans l'industrie des semi-conducteurs. Elle a été adaptée dans le cas du model-checking logiciel. Le model-checking borné se base sur les avancées des solveurs SAT et SMT pour effectuer les tâches de vérification.

Au lieu d'explorer entièrement un modèle, les algorithmes de model-checking bornés explorent uniquement des préfixes finis des exécutions possibles du modèle. Une profondeur maximale d'exploration `k` est fixée. Les `k` premiers pas de chaque exécution du système sont alors encodée dans une formule de la logique propositionnelle. On construit ensuite la conjonction entre cette formule et la négation d'une formule représentant la spécification du programme, et on transmet le résultat à un solveur SAT ou SMT. Si la formule est satisfiable, il existe une exécution valide ne respectant pas la spécification (puisqu'elle satisfait la négation de la spécification). Une erreur est donc reportée, la valuation qui a satisfait la formule représente le contre-exemple.

Forme SSA Afin d'encoder les exécutions d'un programme dans une formule, on met tout d'abord ce programme dans la forme Single Static Assignment (SSA). Elle consiste à transformer le programme de telle sorte que chaque variable ne soit assignée qu'une seule fois. Pour ce faire, les variables sont dupliquées, une nouvelle copie étant utilisée à chaque assignation. Les structures de contrôle sont remplacées par des expressions conditionnelles une fois que les différents branchements se rejoignent. Les fonctions sont inlinées — leur corps est inséré directement à l'emplacement de l'appel — et les boucles sont déroulées un nombre de fois limité par la profondeur d'exploration, en une cascade de blocs `if`.

Le passage en forme SSA est illustré par l'exemple présenté dans les listings 2.8 et 2.9.

Listing 2.8 Code initial

```
int a = 0, b = 1, c;
a = a + b;
if (a > 0)
    c = 2;
else
    c = 3;
```

Listing 2.9 Forme SSA

```
int a_0 = 0;
int b_0 = 1;
int a_1 = a_0 + b_0;
int c_0 = 2;
int c_1 = 3;
int c_2 = a_1 > 0 ? c_0 : c_1;
```

Dépliage de la relation de transition Dans le cadre du BMC, le programme est vu comme une relation de transition R. Étant donnés deux états s et q du programme, $R(s, q)$ s'évalue à vrai si et seulement s'il existe une transition dans le programme permettant de passer de l'état s à l'état q. On peut alors établir la formule représentant les k premiers pas d'une exécution par :

$$F = I(s_0) \wedge \bigwedge_{i \in \{1..k\}} R(s_{i-1}, s_i)$$

Le solveur va tenter de résoudre la formule en trouvant une suite d'états s_i qui la satisfait. Cette suite représente alors une exécution possible dans le programme : $I(s_0)$ impose que l'exécution commence dans un état initial, et $\bigwedge_{i \in \{1..k\}} R(s_{i-1}, s_i)$ impose que les transitions entre deux états successifs de la trace soient valides.

Partant de la forme SSA, la relation de transition se déplie facilement en construisant la conjonction des instructions. Les variables de la forme SSA peuvent directement être considérées comme des variables logiques. La principale difficulté consiste à établir une théorie pour représenter les instructions, en particulier les utilisations dynamiques de la mémoire.

Dans notre exemple, la relation de transition se déplierait en :

$$\underbrace{a_0 = 0 \wedge b_0 = 1}_{\text{État initial}} \wedge \underbrace{a_1 = a_0 + b_0}_{\text{Instruction 1}} \wedge \underbrace{c_0 = 2 \wedge c_1 = 3}_{\text{Branches du tests}}$$
$$\wedge \underbrace{(a_1 > 0 \implies c_2 = c_0) \wedge (\neg a_1 > 0 \implies c_2 = c1)}_{\text{Fusion des branches}} \quad (2.3)$$

Afin de vérifier une propriété Φ du programme, on construit la conjonction entre la formule représentant les exécutions du système et la négation de la propriété.

$$F' = \neg\Phi \wedge I(s_0) \wedge \bigwedge_{i \in \{1..k\}} R(s_{i-1}, s_i)$$

Un modèle de cette formule représente donc les k premiers pas d'une exécution valide, et qui ne respecte pas la propriété Φ. Il s'agit donc d'une exécution contenant une erreur.

Le BMC explore un système sur une profondeur bornée, il est donc nécessairement incomplet. Il permet de trouver des erreurs, ou de prouver des propriétés d'accessibilité, mais il ne peut prouver des propriétés de sûreté ou trouver des erreurs à des propriétés de vivacité : une erreur ou l'évènement attendu pourraient être présents à une profondeur plus importante que la profondeur maximale d'exploration.

Lorsque le modèle contient un nombre d'états finis, il est possible de rendre le BMC complet : il suffit de fixer une profondeur maximale d'exploration suffisamment grande pour que toutes les traces du programme n'affichent plus de nouveaux comportements une fois cette profondeur atteinte. Cette profondeur est appelée le seuil de complétion (*completeness threshold*). Calculer la valeur exacte du seuil de complétion est un problème difficile. On utilise donc des sur approximations. Cependant, l'exploration d'un système jusqu'au seuil de complétion est souvent trop coûteuse en temps et en mémoire pour être réalisable. Le BMC est donc souvent utilisé pour détecter des erreurs dans un système plutôt que pour prouver leur absence, en effectuant plusieurs explorations avec une profondeur maximale croissante.

Support du multi threading Plusieurs approches ont été développées pour prendre étendre le BMC à des logiciels multi thread[23].

Deux techniques sont principalement utilisées. La première consiste à encoder tout les entrelacements possibles entre les threads du programme dans la relation de transition, et donc dans la formule de la logique propositionnelle fournie à un SAT-solveur ou un SMT-solveur. Cette technique permet au solveur d'optimiser sa preuve en utilisant les similarités entre dif-

férents entrelacements. Cependant, les formules générées peuvent être extrêmement grandes et dépasser les capacités des solveurs.

La seconde approche se base sur l'exploration explicite des entrelacements du programme : une formule est passée au solveur pour chaque entrelacement, le procédé étant interrompu dès qu'une erreur est détectée. Les formules générées sont ainsi plus petites et plus simples à résoudre pour les solveurs, cependant, un nombre exponentiel d'appels au solveur peut être nécessaire, avec des formules parfois très semblables.

Le nombre de changements de contexte autorisés dans une exécution est généralement borné de manière indépendante par rapport à la profondeur d'exploration. On évite ainsi d'explorer un nombre trop élevé d'entrelacements. Cette seconde borne repose sur l'observation empirique : la plupart des erreurs dues au parallélisme sont peu *profondes*, elles peuvent être atteintes en un nombre réduit de changements de contexte.

Le model-checking borné est la méthode la plus efficace pour trouver des erreurs peu profondes. Elle supporte bien le multi threading. Cependant, cette méthode est peu efficace pour prouver l'absence d'erreur et ne peut généralement valider un programme que si sa profondeur est faible. Elle est particulièrement sensible aux boucles dont le nombre d'itérations n'est pas borné.

Outils

CBMC[19] est le premier model-checker logiciel à avoir utilisé le model-checking borné. Il est orienté vers les applications embarquées, et permet de vérifier un programme par rapport à différents modèles d'exécution. Il se base sur des SAT-solveurs afin de vérifier les conditions de vérification qu'il génère. CBMC remplace les opérateurs arithmétiques par la description de circuits équivalents afin de construire une formule logique (*bit-flattening*). CBMC supporte les programmes multi thread en encodant symboliquement les entrelacements dans la formule passée au model-checker. Comme la majorité des model-checkers, CBMC supporte la consistance séquentielle (i.e. un ordre total entre les instructions est considéré). Cependant, CBMC est aussi capable de traiter des modèles mémoires faibles (i.e. un ordre partiel entre les instructions est considéré). En effet, les multiprocesseurs modernes ne respectent pas la consistance séquentielle : d'autres comportements peuvent apparaître, liés à la propagation des valeurs à travers les niveaux de cache.

ESBMC[24] a été conçu en se basant sur CBMC, dont il reprend le frontend. ESBMC utilise un solveur SMT (SAT Modulo Theory) au lieu d'un solveur SAT pour vérifier les formules générées et utilise des théories différentes de celles utilisées par CBMC afin de créer ces

formules. ESBMC supporte les programmes multi thread en énumérant explicitement tous les entrelacements possibles du programme, et en effectuant une exploration symbolique sur chacun d'entre eux. ESBMC améliore le support des boucles par l'utilisation de preuves par induction[40]. Les boucles sont déroulées sur un nombre k d'itérations. Si cela ne suffit pas à trouver un contre-exemple, ESBMC va tenter d'établir un invariant en montrant qu'il est maintenu tout au long de la boucle par une preuve par induction.

SMACK[17] permet de traduire un programme en C vers le langage de modélisation Boogie. Il utilise pour cela la représentation intermédiaire LLVM-IR et implémente des simplifications du code source pour réduire le modèle. Le modèle en Boogie est ensuite vérifié à l'aide d'un model-checker utilisé en backend. Par défaut, Coral est utilisé. Il s'agit d'un model-checker borné se basant sur des solveurs SMT pour les programmes en Boogie, supportant la concurrence. Ceci permet à SMACK de tirer parti de backends éprouvé et des performances des solveurs. SMACK reste cependant modulaire, il peut ainsi utiliser d'autres algorithmes que du BMC pour le backend. SMACK est capable de vérifier des spécifications sous forme d'assertions uniquement.

2.3.5 Séquentialisation

Les techniques de model-checking logiciel pour des programmes séquentiels ont largement progressé au niveau de leurs performances et de leur précision. Cependant, les model-checkers restent bien plus limités dans le cas de programmes concurrents. La séquentialisation réduit le problème de la vérification d'un programme multi thread à celui de la vérification d'un programme séquentiel pour tirer partit des performances des outils dans ce cas.

La séquentialisation consiste à traduire un programme concurrent en un programme séquentiel non déterministe équivalent. Le non-déterminisme lié au contrôle (les différents entrelacements) est remplacé par un non-déterminisme lié aux données, ce qui permet de mieux le gérer. Les techniques de séquentialisation ne s'appliquent donc qu'à un programme multi thread, il est nécessaire de les combiner avec un outil de model-checking adapté au cas séquentiel pour réaliser les tâches de vérification.

La séquentialisation permet d'attaquer deux problèmes simultanément : améliorer la prise en charge de la concurrence en se ramenant au cas séquentiel, et proposer une interface de gestion de la concurrence compatible avec plusieurs model-checkers, sans que ceux-ci n'aient à modifier leurs algorithmes.

Nous allons énumérer les idées clefs des algorithmes de séquentialisation, tout en les illustrant par l'exemple de la transformation réalisée par Lazy-CSeq[34].

Partons d'un programme concurrent très simple (nous nous inspirons ici de l'exemple présenté dans [34], auquel on peut se référer pour plus de détails) :

```
int y = 0;
void* T0(void *d) {
  int x;
  x = y;
  y = x + 1;
}
void* T1(void *d) {
  y = 3;
}
int main() {
  pthread_t t0, t1;
  pthread_create(t0, NULL, T0, NULL);
  pthread_create(t1, NULL, T1, NULL);
}
```

Les idées clefs de la séquentialisation sont les suivantes :

1) Remplacer les structures concurrentes par des structures séquentielles (un thread est remplacé par une fonction...)
2) Instrumenter le code afin de simuler les changements de contexte.
3) Ajouter un ordonnanceur non-déterministe. Il dirige l'exécution et permet de générer tous les entrelacements (généralement de type circulaire) du programme.
4) Recourir à un model-checker pour explorer le résultat.

Dans notre exemple, le premier point consiste à changer la fonction `main`, qui représente le thread principal, en une fonction séparée `main_thread`. Les autres threads sont déjà représentés par une unique fonction, il n'est donc pas utile de les modifier (si deux threads exécutaient la même fonction, il serait nécessaire de la dupliquer). La seconde étape est réalisée en introduisant un saut conditionnel avant chaque instruction du programme, à travers la macro `J`. Cette macro permet d'interrompre et de reprendre l'exécution d'une fonction à partir d'une instruction donnée, en sautant les instructions qui précèdent ou suivent. La variable `cs` contient à chaque instant le numéro du label correspondant à la prochaine interruption, et le tableau `pc` contient le numéro du label de la dernière instruction atteinte lors de l'exécution précédente. Pour conserver les valeurs des variables locales pendant les

interruptions, ces dernières sont rendues statiques. Enfin, la fonction `main` contient l'ordonnanceur. Il va simuler `k` passes d'un ordonnancement de type round-robin, en choisissant de manière non déterministe le nombre d'instructions à exécuter dans chaque thread avant de changer de contexte.

On obtient alors le code séquentialisé du listing 2.10. Des structures de contrôle supplémentaires permettent de représenter la synchronisation des threads.

La séquentialisation ne préserve pas nécessairement toutes les propriétés du code, selon les transformations effectuées. Les propriétés d'accessibilité sont généralement préservées, cependant la plupart des approches déplient les boucles et bornent la profondeur d'exploration et le nombre de changements de contexte possibles.

Outils

CSeq[27] suit le schéma de séquentialisation de Lal/Reps. Ce dernier consiste à considérer K copies de la mémoire partagée, pour K phases d'un ordonnancement *round-robin*. La fonction représentant chaque thread va mettre à jour la $i^{ème}$ copie jusqu'au $i^{ème}$ changement de contexte, avant de la passer au thread suivant. Enfin, toutes les exécutions incohérentes (lorsque l'état de la mémoire à la fin d'un cycle ne correspond pas à l'état deviné au début du cycle suivant) sont supprimées. CSeq utilise des model-checkers bornés pour mener la vérification. Il est compatible avec CBMC, ESBMC et LLBMC.

Lazy-CSeq[34] se base sur le framework mis en place par CSeq mais prend une approche paresseuse (*lazy*). Il ajoute des instructions de contrôle aux fonctions représentant les threads, ce qui permet d'interrompre ou de reprendre l'exécution d'une fonction chaque fois qu'un changement de contexte peut avoir lieu. L'ordre d'exécution des instructions dans un entrelacement est ainsi préservé, ce qui permet de réduire considérablement l'utilisation du non-déterminisme. Lazy-CSeq est compatible avec CBMC, ESBMC et LLBMC.

Mu-CSeq[49] se base sur le concept de *memory unwinding* (déroulement de la mémoire, MU). Un MU consiste en une séquence d'écriture dans la mémoire partagée du programme. Mu-CSeq choisit un MU de manière non déterministe et le confronte à une simulation du programme afin de vérifier si les écritures correspondent. Si c'est le cas, une exécution valide a été détectée. Cette approche permet d'éviter d'explorer des entrelacements équivalents, seul l'entrelacement des écritures étant pris en compte. Mu-CSeq utilise CBMC comme backend.

Unbounded-Lazy-CSeq[45] se base sur l'algorithme utilisé par Lazy-CSeq, mais permet de ne pas limiter le nombre de changements de contexte. Il est aussi capable de traiter des programmes non bornés. Pour cela, UL-CSeq conserve les boucles mais se base sur CPAChecker[14]

Listing 2.10 Code séquentialisé

```
int cs,ct,pc[T],size[T]={2,4,7};
#define J(A,B) if(pc[ct]>A||A>=cs) goto _##B;
int y=0;
void T0(void *arg) {
          static int x;
_0:J(0,1) x = y;
_1:J(1,2) y = x + 1;
_2: ;
}
void T1(void *arg) {
_3:J(3,4) y = 3;
_4: ;
}
int main_thread() {
          static pthread_t t0,t1;
_5:J(5,6) pthread_create(&t0,NULL,T0,0,1);
_6:J(6,7) pthread_create(&t1,NULL,T1,0,2);
_7: ;
}
int main() {
  int K = 10; // Check over 10 rounds
  int r;
  for(r=1; r<=K; r++) {
    cs=pc[0] + nondet_uint();
    assume(cs<=size[0]);
    main_thread();
    pc[0]=cs;
    cs=pc[1] + nondet_uint();
    assume(cs<=size[1]);
    T1();
    pc[1]=cs;
    cs=pc[2] + nondet_uint();
    assume(cs<=size[2]);
    T1();
    pc[2]=cs;
  }
}
```

en backend et non sur des model-checkers bornés.

De manière générale, les outils de séquentialisation ont obtenu d'excellentes performances ces dernières années. Lazy-CSeq a obtenu la médaille d'or de la catégorie *concurrence* lors des éditions 2014 et 2015 de la SVCOMP et une médaille d'argent pour les éditions 2016 et 2017, tandis que Mu-CSeq a obtenu la médaille d'or lors de l'édition 2016 et la médaille d'argent lors des éditions 2014 et 2015 [11, 10, 9, 12].

2.3.6 Bilan

Différents outils et techniques de model-checking ont été développés pour tenter de résoudre le problème principal du model-checking : l'explosion combinatoire. Ces techniques ont différents points forts et défauts, qui dépendent des propriétés types à vérifier, de la structure du modèle (boucles, non-déterminisme sur les données ou le contrôle...) et du résultat attendu (détection d'erreurs, preuve de correction...).

Afin de gagner en performances, les différents outils implémentent des optimisations et des techniques de réduction qui provoquent en général une certaine perte de précision. La plupart des outils restreignent ainsi le type de propriétés et de spécification qu'ils supportent afin de gagner en performances. La figure A.1, en annexe, récapitule les types de propriétés qu'il est possible de vérifier avec chaque outil (selon les articles que nous avons passés en revue). On remarque en particulier que si la plupart des outils supportent une spécification à l'aide d'assertions, le support pour LTL est bien plus réduit.

La gestion de la concurrence reste encore un problème ouvert. Elle est actuellement prise en charge par différents mécanismes : les différents entrelacements peuvent être intégrés directement dans le modèle, ils peuvent aussi être énumérés explicitement. Une autre approche est de mener une preuve modulaire, en examinant chaque thread individuellement et en simulant ses interactions avec le reste du système. Enfin, les techniques de séquentialisation permettent de ramener le non-déterminisme dû aux entrelacements à du non-déterminisme dû aux données que les model-checkers savent mieux traiter.

Par la suite, nous allons utiliser des model-checkers en tant que backend afin de vérifier du code instrumenté. L'objectif sera alors de considérer le model-checker comme une boîte noire, dont il est inutile de connaître le fonctionnement interne. Connaître les techniques utilisées restera cependant utile pour comprendre les résultats et les performances obtenus.

CHAPITRE 3 LTL AVEC SUPPORT DES POSITIONS ET DES VARIABLES LOCALES

3.1 Limitations des méthodes de spécification actuelles

À travers la revue de littérature, nous avons dégagé deux principales limitations concernant les méthodes de spécification utilisées pour model-checking de programmes concurrents.

La première limitation concerne les assertions. Les assertions sont le mécanisme de spécification le plus utilisé dans le cadre du model-checking logiciel. Cependant, les propriétés que l'on peut spécifier par des assertions sont limitées à un sous-ensemble des propriétés de sûretés — une assertion spécifie qu'une certaine condition doit être vraie à une certaine position du code. Dans le cas de programmes concurrents, ce type de propriétés est souvent insuffisant.

En effet, l'entrelacement entre les instructions d'un programme concurrent est généralement complexe. Il est donc fréquent de vouloir vérifier des propriétés portant sur l'ordre d'apparition de certains évènements au cours d'une exécution du programme. Les assertions ne sont pas adaptées pour spécifier ce type de propriétés.

Exemple L'absence de conditions de concurrence dans un programme est une propriété généralement désirable. Elle fait partie de l'ensemble plus vaste des propriétés d'exclusion mutuelle : il s'agit de vérifier que deux instructions accédant à une même variable partagée ne peuvent être exécutées simultanément. Les listings 3.1 et 3.2 présentent un cas très simple de data-race.

Listing 3.1 Thread 1

```
int p = 0;
void* thread1(void* d) {
  ...
  p += 1;
  ...
}
```

Listing 3.2 Thread 2

```
void* thread2(void* d) {
  ...
  p += 1;
  ...
}
```

Si les instructions `p += 1` des deux threads ont lieu simultanément, la valeur finale de `p` est indéterminée.

Spécifier l'absence de data-races dans ce programme à l'aide d'assertions n'est pas immédiat : les assertions ne permettent pas d'exprimer la notion de simultanéité. Il est nécessaire

d'introduire des variables supplémentaires, et donc de modifier le système. La spécification la plus simple que nous avons pu produire est présentée dans les listings 3.3 et 3.4 [1].

Listing 3.3 Thread 1

```
int p = 0;
int flag = 0;
void* thread1(void* d) {
  ...
  assert(!flag);
  flag = 1;
  p += 1;
  flag = 0;
  ...
}
```

Listing 3.4 Thread 2

```
void* thread2(void* d) {
  ...
  assert(!flag);
  flag = 1;
  p += 1;
  flag = 0;
  ...
}
```

Cet exemple souligne le fait que, dans un programme concurrent, on s'intéresse fréquemment au lien entre plusieurs *assertions* plutôt qu'au déclenchement d'une assertion simple. On aimerait pouvoir formuler des propriétés telles que "si deux assertions sont violées simultanément, une erreur est présente" ou "si une assertion est violée, puis une seconde assertion est violée, une erreur est présente".

La seconde limitation concerne les spécifications utilisant LTL. Les model-checkers logiciels supportant LTL restreignent généralement les propositions atomiques à des fonctions booléennes portant sur la valeur des variables globales du programme. Si toutes les variables globale ont la même valeur dans deux états distincts du programmes (qui diffèrent par leur position dans le code et leurs variables locales), alors toutes les formules LTL vont s'évaluer de la même manière dans ces deux états : il n'est pas possible de les distinguer dans la spécification, ce qui limite le type de propriétés qu'il est possible de spécifier.

En particulier, ces restrictions rendent impossible d'exprimer une assertion par une formule LTL. Alors que les logiques temporelles permettent d'exprimer facilement des relations d'ordre entre différents évènements, LTL n'est donc pas une solution aux limitations rencontrées par les assertions.

Exemple Reprenons l'exemple précédent. La propriété d'exclusion mutuelle que l'on veut spécifier peut s'écrire $G\neg(p_1 \wedge p_2)$ où p_1 (respectivement p_2) est la proposition atomique

1. On remarque que cette spécification permet d'atteindre une race-condition sans violer les assertions. Ce n'est pas un problème ici, car s'il est possible d'atteindre une race-condition, alors au moins un entrelacement viole les assertions, un model-checker signalera donc correctement l'erreur.

indiquant si le thread 1 (respectivement le thread 2) est dans sa zone critique. Mais on ne peut pas exprimer p_1 et p_2 directement, sans mentionner des positions du programme. Encore une fois, il est nécessaire d'introduire des variables supplémentaires et de modifier le système. Les listings 3.5 et 3.6 présentent une solution, associée avec la spécification $G\neg(flag1 == 1 \wedge flag_2 == 1)$.

Listing 3.5 Thread 1

```
int p = 0;
int flag1 = 0;
void* thread1(void* d) {
  ...
  assert(!flag);
  flag1 = 1;
  p += 1;
  flag1 = 0;
  ...
}
```

Listing 3.6 Thread 2

```
int flag2 = 0;
void* thread2(void* d) {
  ...
  assert(!flag);
  flag2 = 1;
  p += 1;
  flag2 = 0;
  ...
}
```

Les deux mécanismes de spécification les plus utilisés par le model-checking logiciel sont donc limités, sans être complémentaires. Certaines propriétés sont alors complexes à spécifier. En pratique, le cas des data-races présenté en exemple est suffisamment pour être implémenté en tant que propriété built-in, mais ce n'est pas le cas pour d'autres propriétés (d'exclusion mutuelle ou non).

Dans ce chapitre, nous allons présenter un nouveau formalisme de spécification permettant de remédier à ce problème. Il se base sur une restriction de LTL plus souple que celle utilisée classiquement par les model-checkers logiciels. Nous introduisons le concept de *zones de validités* afin de permettre l'utilisation de variables locales et de positions dans les propositions atomiques. Nous produisons ainsi un formalisme de spécification qui englobe les assertions et les propositions LTL (telles que classiquement utilisées par les model-checkers logiciel), tout en permettant de surmonter les limitations rencontrées par les utilisations classiques de LTL : l'utilisation des variables locales et des positions du code source dans la spécification.

Nous allons tout d'abord présenter les différents obstacles à résoudre afin d'atteindre cet objectif. Nous présenterons ensuite les solutions (parfois partielles) que nous apportons à ces problèmes, ainsi que le formalisme de spécification que nous avons établi. Enfin, nous justifierons nos choix et nous les comparerons aux autres travaux connexes.

3.2 Comment spécifier les propriétés d'un code ?

Différents problèmes doivent être surmontés afin de permettre l'utilisation de variables locales et de positions dans une spécification. Limiter LTL à des propositions atomiques portant sur les variables globales a permis d'éviter ces problèmes, qui sont spécifiquement liés aux variables locales et aux positions.

3.2.1 Désignation des variables d'un programme

Pour utiliser une variable dans la spécification d'un programme, il est nécessaire de la désigner de manière unique. Le language C n'autorise qu'une seule variable globale portant un nom donné dans un fichier [2]. Une variable globale peut donc être identifiée de manière unique par son nom dans la plupart des cas (en ajoutant le nom du fichier où elle est déclarée si nécessaire). Ce n'est pas aussi simple lorsqu'on considère des variables locales : deux variables locales portant le même nom peuvent cohabiter sans faire référence au même emplacement mémoire.

Deux variables locales distinctes (ne faisant pas référence à la même adresse mémoire), mais ayant le même nom peuvent survenir dans diverses situations, parmi lesquelles :

- une variable `foo` est définie dans plusieurs contextes distincts (des fonctions différentes, des blocs différents d'une même fonction...). La connaissance du contexte de la définition permet de les distinguer.
- une fonction `bar` contenant une variable `foo` est appelée plusieurs fois séquentiellement dans le programme. La variable `foo` dépend alors de l'appel à la fonction `bar`.
- la variable `foo` est définie dans des threads distincts. Dans ce cas, les deux instances de la variable peuvent exister simultanément. L'instance dépend du thread dans lequel elle est définie.
- la variable `foo` est définie dans une fonction récursive `bar`. Dans ce cas, une nouvelle instance de `foo` est définie à chaque appel récursif de `bar`.

On peut répartir ces cas en deux catégories :

- les deux variables sont lexicalement distinctes, elles sont issues de deux définitions distinctes dans le code source. Dans ce cas, on peut différencier les variables en ajoutant des informations supplémentaires à leurs noms, pour identifier les contextes de déclaration de manière unique (fichier, fonction, bloc...).
- les deux variables sont lexicalement identiques, mais elles sont redéfinies. Ce cas est

2. Les mots clefs `extern` et `static` permettent de déclarer plusieurs variables globales portant le même nom dans un programme

plus complexe : les appels de fonctions et les créations de threads peuvent être dynamiques et dépendre de l'exécution. Ils sont donc difficiles à identifier de manière statique, et il en va de même pour les variables locales associées.

3.2.2 Portée des propositions atomiques

Dans une formule LTL, une proposition atomique est intrinsèquement globale : pour tout état s du programme, il peut être nécessaire de déterminer si s est un modèle de la proposition atomique. Cela est généralement déterminé selon la valeur prise par une expression booléenne dans l'état s. L'expression est évaluée en utilisant la valeur des variables dans l'état s.

Cependant, si cette expression dépend de variables locales, elle ne peut être évaluée que si toutes ces variables locales existent dans l'état s. Une proposition atomique n'est donc correctement définie que dans le domaine où sont définies les variables locales dont elle dépend.

Pour obtenir une définition correcte d'une proposition atomique dépendant de variables locales, il est donc nécessaire de prolonger sa définition à l'ensemble des états.

3.2.3 Désignation des positions d'un programme

Tout comme pour les variables, il est nécessaire de désigner de manière unique une position du code pour l'utiliser dans une spécification.

Il faut donc concevoir un moyen de désigner une position dans le code qui soit à la fois pratique pour l'utilisateur et robuste par rapport aux modifications du code. Il est aussi nécessaire de tenir compte de la précision recherchée. Sur ce point, nous nous limiterons à désigner une instruction. En effet, si l'on veut désigner une expression au sein d'une instruction, il est toujours possible de la transformer en une instruction indépendante.

Plusieurs options peuvent être envisagées pour résoudre ce problème :

— indiquer la position à l'aide du numéro de la ligne et du nom du fichier ;
— placer des marqueurs dans le code ;

La première méthode souffre d'un inconvénient majeur : la spécification doit être corrigée dès qu'une ligne est ajoutée ou enlevée dans le code. Nous avons donc retenu la seconde option. Elle impose cependant de rajouter une instrumentation dans le code, ce que nous désirons minimiser.

Une fois la manière de désigner une position établie, il faut déterminer comment l'utiliser dans la spécification. Nous dirons par la suite qu'un programme a atteint une position dans

le code lorsqu'un des pointeurs d'instruction du programme pointe sur l'instruction désignée par la position.

Il est cependant nécessaire de définir comment et dans quelle mesure une position peut intervenir dans une proposition atomique. Nous avons considéré les options suivantes :

1) une proposition atomique désigne une position ou une condition sur les variables du programme, de manière exclusive. On fait le lien entre les positions et les valeurs des variables du programme à l'aide des opérateurs de la logique des prédicats. Si `pos1` et `pos2` désignent deux positions du programme, alors on spécifie que la variable `x` est non nulle entre `pos1` et `pos2` par $G(\{\text{pos1}\} \implies \{x \neq 0\} U \{\text{pos1}\})$.
2) une proposition atomique est constituée d'une condition sur les variables du programme et d'une condition sur les positions. En reprenant l'exemple précédent, on écrirait alors $G\neg p$, avec p la proposition atomique qui est vérifiée par tous les états ayant une position entre `pos1` et `pos2` et où `x` est nulle.
3) les positions viennent contraindre les opérateurs temporels. On pourrait envisager une approche semblable à la logique MTL[36], où les opérateurs temporels sont restreints à des intervalles de temps, mais en appliquant la restriction à des positions du programme. On écrirait alors $G_{[pos1,pos2]}\neg\{x \neq 0\}$.

Nous avons décidé de ne pas explorer l'option 3. En effet, elle demande de modifier le comportement standard des opérateurs LTL et nous avons préféré limiter nos modifications aux propositions atomiques seulement.

3.3 Syntaxe et sémantique de la spécification proposée

3.3.1 Syntaxe

Nous allons tout d'abord présenter la grammaire de la spécification que nous avons finalement retenue, avant de justifier nos choix.

Notre spécification est composée de deux parties. D'une part, une formule LTL classique. D'autre part, la définition des propositions atomiques utilisées par la formule LTL.

La définition d'une proposition atomique est composée des éléments suivants :

— **un nom** : il permet de référer facilement à la proposition atomique.
— **une zone de validité** : elle délimite un bloc d'instructions dans le code (au sens du langage C). Une zone de validité est délimitée par deux labels (placés dans le code par l'ingénieur réalisant la spécification). L'entrée et la sortie d'une zone de validité

doivent être dans le même contexte et tout pointeur d'exécution atteignant le label de début doit atteindre le label de fin avant de sortir du contexte (i.e. un branchement ne doit pas permettre de sortir d'une zone de validité en évitant l'instruction portant le label de fin).

— **une fonction d'évaluation** : il s'agit d'une fonction booléenne pure, écrite en C. Cette fonction est utilisée pour déterminer si un état dans la zone de validité vérifie la proposition atomique ou non.

— **une liste de paramètres** : ils désignent les variables qui seront passées en argument de la fonction d'évaluation lorsque celle-ci est évaluée. Une variable locale est préfixée par le nom de la fonction dans laquelle elle est définie. Tous les paramètres doivent être définis dans la zone de validité de la proposition atomique.

— **une valeur par défaut** : lorsque la proposition atomique n'est pas dans sa zone de validité, elle s'évalue à sa valeur par défaut. Cette valeur par défaut permet de prolonger la définition de la proposition atomique en dehors de sa zone de validité.

Le tableau 3.1 résume la grammaire de notre spécification.

Tableau 3.1 Grammaire des propositions atomiques

<atomic-proposition>	: :=	<proposition-id> <evaluation-function> <parameters> <default> <validity-area>
<proposition-id>	: :=	*name of the proposition*
<evaluation-function>	: :=	*C pure boolean function*
<parameters>	: :=	<parameter> <parameters> | *nil*
<parameter>	: :=	<global-parameter> | <local-parameter>
<global-parameter>	: :=	*variable name*
<local-parameter>	: :=	*function name* : : *variable name*
<default>	: :=	*boolean*
<validity-area>	: :=	<label> <label>
<label>	: :=	*name of a C label*

La spécification est alors constituée d'une formule LTL portant sur des propositions atomiques définies selon cette grammaire.

3.3.2 Sémantique

Les opérateurs de la logique classique et de la logique temporelle suivent la sémantique usuelle de la logique propositionnelle et de LTL.

Les propositions atomiques s'évaluent selon la valeur des pointeurs d'instruction du programme. On considère qu'un état p est dans la zone de validité d'une proposition atomique

lorsque, dans l'état p, un pointeur d'instruction du programme pointe sur une instruction comprise entre le label de début de la zone de validité de la proposition et le label de fin. Cela signifie que, quelque soit l'exécution menant à l'état p, le pointeur d'exécution a déjà atteint l'instruction désignée par le label de début, mais pas encore celle désignée par le label de fin. Pour que la spécification soit valide, il est nécessaire que, quelque soit l'exécution, si un pointeur d'instruction désigne l'instruction marquée par un label de début, alors il atteigne celle marquée par le label de fin avant de sortir du contexte courant.

On évalue alors la proposition atomique en un état p par :

— si l'état p n'est pas dans la zone de validité de la proposition atomique, alors la proposition atomique s'évalue à sa valeur par défaut.
— si l'état p est dans la zone de validité de la proposition atomique, alors elle s'évalue à la valeur renvoyée par l'évaluation de sa fonction d'évaluation, avec comme paramètres la liste de variables de la proposition atomique, dans l'ordre.

3.3.3 Exemple

Nous avons choisi par la suite de représenter cette spécification en JSON (JavaScript Object Notation)[4]. Ce format est relativement verbeux, mais présente l'avantage d'être facile à parser tout en étant lisible pour l'être humain. Établir une syntaxe plus concise pourrait représenter un développement futur.

Étant donné le code suivant, constitué de deux threads s'exécutant simultanément (pour des raisons de concision, la fonction `main` a été omise), nous allons spécifier le fait que la variable `a` du premier thread n'est jamais nulle en même temps que la variable `a` du second thread.

Listing 3.7 Thread 1

```
void* thread1(void* d) {
  int a
b_p1:
  a = 0;
  a = 1;
e_p1:
  pthread_exit(NULL);
}
```

Listing 3.8 Thread 2

```
void* thread2(void* d) {
  int a;
b_p2:
  a = 1;
  a = a - 1;
e_p2:
  pthread_exit(NULL);
}
```

Il est nécessaire de rajouter des fonctions d'évaluation pour les propositions atomiques :

Listing 3.9 Proposition atomique p1

```
int f_ev_1(int x) {
    return x == 0;
}
```

Listing 3.10 Proposition atomique p2

```
int f_ev_2(int x) {
    return x == 0;
}
```

On obtient alors la spécification présentée dans le listing 3.11.

Listing 3.11 Spécification

```
{ "ltl": "G(! (p1 && p2))",
  "pa": [
    {"name": "p1",
      "default": false,
      "expr": "f_ev_p1",
      "span": ["b_p1", "e_p1"],
      "params": ["thread1::a"]
    },
    { "name": "p2",
      "default": false,
      "expr": "f_ev_p2",
      "span": ["b_p2", "e_p2"],
      "params": ["thread2::a"]
    }
  ]
}
```

Dans cet exemple, le code ne respecte pas la spécification. L'exécution $t1 : a = 0 \rightarrow t2 : a = 1 \rightarrow t2 : a = a - 1$ permet d'obtenir un état qui ne satisfait pas la propriété.

3.4 Motivation de nos choix

Nous allons maintenant expliquer comment et dans quelle mesure ce formalisme de spécification permet de surmonter les obstacles détaillés précédemment.

3.4.1 Identification des variables

Nous avons choisi d'identifier une variable globale par son nom et une variable locale par le nom de la fonction dans laquelle elle est définie et le nom de la variable. Cela n'apporte qu'une solution partielle au problème de l'identification des variables.

Cas de deux variables lexicalement distinctes Notre solution permet de distinguer des variables lexicalement distinctes tant qu'elles ne sont pas définies dans une même fonction. Nous avons jugé que ce cas est suffisamment peu fréquent pour ne pas le traiter. Nous laissons donc le soin à l'utilisateur de renommer les variables concernées.

Cas de plusieurs instances d'une variable lexicalement identique Nous ne proposons pas de solution dans ce cas de manière générale. Cependant, il existe des solutions pour quelques cas particuliers.

L'un des cas où identifier une variable dépend d'une notion dynamique a lieu lorsqu'une même fonction est utilisée pour créer plusieurs threads. En effet, les threads peuvent être créés de manière dynamique, leur nombre et leur ordre de création peuvent donc dépendre de l'exécution. Cependant, il est fréquent de créer les threads d'un programme de manière statique, en créant toujours le même nombre de threads dans le même ordre (dans la fonction `main` par exemple). Dans ce cas, il est possible d'identifier un thread par un indice correspondant à l'ordre de création des threads. On pourra alors identifier de manière unique une variable dans un thread uniquement en rajoutant un préfixe à son nom, qui prendra alors la forme `thread_id::fonction::variable` [3].

Un autre cas est lié à l'identification d'une variable locale à une fonction appelée plusieurs fois dans un programme. Une instance de la variable est créée à chaque appel. Si les appels à la fonction sont séquentiels (il existe donc plusieurs instances de la variable dans le programme, mais une seule existe à chaque instant), considérer que ces différentes instances correspondent à la même variable dans la spécification est généralement le comportement attendu.

Cependant, lors d'appels récursifs ou concurrents, plusieurs instances de la variable existent simultanément : pour évaluer une proposition atomique, il faut alors décider quelle instance doit être utilisée. Dans ce cas, nous ne proposons pas de solution. Il est cependant possible de reprendre l'approche de Divine[5], en l'adaptant à notre système de valeurs par défaut[3]. Nous présentons cette approche plus en détail dans la section 3.6.

3.4.2 Identification des positions dans le code

Identifier une position dans le code sans y placer un marqueur n'est pas une solution viable : elle est trop sensible aux modifications du code, d'autant plus que lors de la vérification d'un programme, on s'attend à plusieurs itérations avec des modifications mineures du code, afin de corriger les erreurs détectées.

3. Cette fonctionnalité n'est pas implémentée dans l'outil présenté dans le Chapitre 4.

Nous avons donc utilisé des marqueurs. Nous avons choisi des labels, puisqu'ils permettent de désigner une instruction du code. Des alternatives auraient été d'utiliser des commentaires ou des macros afin de marquer les instructions. Nous avons préféré les labels pour des raisons d'implémentation, les commentaires n'étant pas conservés par de nombreuses bibliothèques de transformation de code, mais aussi en raison de leur signification : un label a pour rôle de désigner une instruction.

Cependant, la précision d'une position à l'aide d'un marqueur reste de, au mieux une instruction. On ne peut donc pas désigner une position plus précisément, au niveau d'une expression par exemple. Nous avons considéré que ce cas est suffisamment peu fréquent pour qu'il soit possible de l'ignorer. Il reste de plus possible de décomposer une instruction afin d'en désigner ses parties.

Nous avons choisi de former des intervalles de positions, car ils représentent un bon compromis entre une énumération explicite et des constructions plus complexes (union ou intersections d'intervalles...). Dans les cas où une zone de validité plus complexe est nécessaire, il est possible de la simuler en utilisant plusieurs propositions atomiques et les opérateurs de la logique classique pour les combiner.

3.4.3 Définition des propositions atomiques

Afin de définir correctement les propositions atomiques dans tous les états du programme, nous avons choisi d'utiliser des valeurs par défaut combinées à des zones de validité.

L'intérêt des zones de validité est double. L'utilisateur peut spécifier manuellement quand une proposition doit utiliser sa fonction d'évaluation. Cela permet en particulier d'attendre que les variables soient correctement initialisées. Les zones de validité permettent aussi d'utiliser des positions du code dans la spécification, et de lier directement ces positions à une proposition sur les valeurs des variables du programme. Il n'est donc pas nécessaire d'utiliser deux propositions atomiques `b` et `e` pour définir le début et la fin d'un intervalle de position, une troisième proposition atomique `p` pour spécifier une propriété sur les valeurs des variables et enfin, d'utiliser une formule telle que $G(b \implies (pUe))$ pour spécifier que `p` doit être valide entre `b` et `e`. Gp est suffisant, la zone de validité étant incluse dans la proposition atomique `p`. On obtient ainsi une formule LTL plus simple.

Nous avons choisi d'utiliser une valeur par défaut que l'utilisateur peut fixer, plutôt qu'imposer une valeur par défaut. Il serait aussi possible d'imposer la valeur par défaut (à *faux* par exemple), on pourrait alors reproduire les propositions atomiques ayant *vrai* comme valeur par défaut en utilisant une double négation. Par exemple, pour une proposition atomique p

ayant *vrai* comme valeur par défaut et f comme fonction d'évaluation, et pour la formule Gp, on pourrait utiliser de manière équivalente la proposition atomique q, avec $faux$ comme valeur par défaut et $\neg f$ comme fonction d'évaluation. La formule LTL serait alors $G\neg q$.

Cependant, il nous a semblé plus pratique et confortable de permettre à l'utilisateur de fixer cette valeur de manière cohérente avec l'utilisation de la proposition atomique. Ainsi, dans le cas d'une propriété de sûreté Gp, il est plus naturel de fixer la valeur par défaut de p à *vrai* plutôt que considérer une double négation.

3.5 Expressivité de la spécification proposée

Notre spécification est une restriction de LTL plus faible que celle utilisée par la plupart des model-checkers logiciels. Exprimer une formule LTL ne portant que sur des variables globales est donc très simple. Il suffit de désigner le programme complet comme zone de validité pour toutes les propositions atomiques. Le reste de la spécification est inchangé.

On vérifie ainsi que notre formalisme de spécification englobe bien les formules LTL classiquement utilisées par les model-checkers.

Notre spécification permet aussi de spécifier l'équivalent d'une assertion. En effet, une assertion signifie : "si un pointeur d'instruction du programme pointe sur l'assertion et que l'état ne vérifie pas la condition exprimée, alors il y a une erreur".

On peut donc représenter cela par une propriété de sûreté. On exprime une assertion par la formule Gp, avec p une proposition atomique telle que :

- sa zone de validité correspond à l'emplacement de l'assertion dans le programme ;
- sa fonction d'évaluation est la condition utilisée dans l'assertion, elle s'évalue donc à la valeur de cette expression dans la zone de validité ;
- sa valeur par défaut est *vrai*e, elle s'évalue donc à *vrai* hors de la zone de validité.

Notre formalisme de spécification englobe donc bien les spécifications à l'aide d'assertions et les formules LTL classiquement supportée, tout en tirant parti des forces de ces deux formalismes.

3.6 Travaux similaires

Des travaux portant sur la prise en charge des variables locales ont été réalisés, autour du model-checker Divine[6]. Ils sont présentés dans [5].

Divine met en place un ensemble de macros permettant de définir les propositions atomiques

utilisées dans une formule LTL. Deux macros, `ap(proposition)` et `ap_set(proposition, value)` permettent respectivement de modifier directement la valeur d'une proposition atomique passée en argument. La macro `ap` met la proposition atomique à *vrai* uniquement pendant l'exécution de la macro, tandis que `ap_set` change la valeur d'une proposition atomique de manière durable, jusqu'à une prochaine modification.

Divine permet aussi de lier une proposition à une fonction d'évaluation à l'aide de deux autres macros, `ap_global(proposition, fonction, N, arg1, ..., argN)` et `ap_local(proposition, fonction, N, arg1, ..., argN)`.

La macro `ap_global` doit être appelée dans l'espace global. Elle permet de lier une proposition atomique à une fonction booléenne dont tous les paramètres sont des variables globales.

La macro `ap_local` lie une proposition atomique à une fonction dont certains des paramètres peuvent être locaux. Les paramètres sont identifiés dans le contexte d'appel selon la norme du C, et peuvent ensuite être masqués. Le lien entre la fonction et la proposition atomique dure jusqu'à la sortie du contexte (où les paramètres locaux sont détruits). Ensuite, la proposition atomique prend la valeur par défaut *faux*.

Exemple Reprenons l'exemple d'une data-race (Listing 3.1). Les listings 3.12 et 3.13 présentent la spécification de l'absence de data-race dans le formalisme de Divine. La formule LTL correspondante est $G(\neg(P1 \wedge P2)))$.

Listing 3.12 Thread 1

```
void* thread1(void* d) {
  ...
  ap_set(P1, 1);
  p += 1;
  ap_set(P1, 0);
  ...
}
```

Listing 3.13 Thread 2

```
void* thread2(void* d) {
  ...
  ap_set(P2, 1);
  p += 1;
  ap_set(P2, 0);
  ...
}
```

Exemple Nous allons illustrer l'utilisation de `ap_local` en spécifiant un programme pour éviter une division par zéro. On considère pour cela la formule LTL GP, avec P la proposition atomique indiquant que la variable locale p, dans le listing 3.14 est non nulle. P est définie à l'aide de la macro `ap_local`.

L'usage de macro permet à Divine de mettre en place une spécification concise et intuitive. La désignation des paramètres à passer aux fonctions d'évaluation, en particulier, est à la

Listing 3.14 `ap_local`

```
int non_nul(int p) {
  return p != 0;
}
void* thread1(void* d) {
  int p;
  ap_local(P, non_nul, p);
  ...
  c = 1/p;
  ...
}
```

Listing 3.15 Fonction récursive

```
int p(int a) {
  return a == 8;
}
int count(int a) {
  ap_local(P, p, a);
  if (a < 10)
    return count(a + 1);
  return a;
}
int main() {
    count(6);
    return 0;
}
```

fois moins verbeuse et plus précise — elle permet de designer une variable dont le nom est réutilisé dans la même fonction.

Divine est aussi confronté au problème posé par de multiples instances d'une variable lexicalement identique. Divine choisi d'évaluer une proposition atomique à vrai dès qu'il existe un contexte dans lequel elle s'évalue à vrai. On considère donc la disjonction des valeurs prise par la proposition dans l'ensemble des contextes actifs. Cette solution est justifiée par le fait qu'on est généralement intéressé par l'instant où une proposition atomique ne prend pas sa valeur par défaut (imposée à *faux* par Divine).

Exemple Le code du listing 3.15 présente une fonction récursive. Une nouvelle instance de la variable `a` est donc déclarée à chaque appel récursif, et les différentes instances existent simultanément.

Supposons que l'on veuille vérifier que la variable `a` ne vaut jamais 8 dans la fonction `count`. On cherche donc a vérifier formule $G\neg P$. P est définie de sorte à être vrai si et seulement si `a` vaut 8. En déroulant les trois premiers appels, on obtient trois contextes, où `a` vaut respectivement 6, 7 et 8. Pour $a = 6$ et $a = 7$, P s'évalue à $faux$. Pour $a = 8$, P s'évalue à $vrai$. On retient donc cette valeur pour la proposition atomique. La formule est alors invalide, ce qui permet de trouver une erreur.

Cette solution peut être adaptée dans le cas de notre formalisme de spécification, en prenant la conjonction des valeurs si la valeur par défaut est *faux*, et la conjonction sinon.

Cependant, Divine ne propose pas l'équivalent d'un intervalle de validité. Limiter la véri-

fication d'une propriété à un intervalle de ligne dans le code demande donc de définir des propositions atomiques supplémentaires, dont la valeur est gérée par `ap` et `ap_set`, et d'utiliser les opérateurs LTL. Cette approche produit des formules plus complexes, et donc plus difficiles à vérifier.

CHAPITRE 4 INTÉGRATION DE LA SPÉFICICATION PROPOSÉE DANS LE PROGRAMME

Nous avons présenté un formalisme de spécification basé sur LTL dans le Chapitre 3. Notre objectif est maintenant d'établir un outil permettant de vérifier un programme concurrent en C, spécifié dans ce formalisme.

Construire un outil de model-checking performant et adapté à notre spécification dépasse de loin le cadre de cette maîtrise. Nous allons donc nous baser sur les outils existants. Cependant, la plupart des outils de model-checking pour des programmes concurrents en C ne supportent pas LTL, alors que notre spécification est basée sur cette logique temporelle. Adapter la logique interne d'un model-checker représente une tâche complexe.

Cependant, presque tous les model-checkers logiciels supportent les assertions. Nous avons par conséquent mis en place une transformation source à source du système. À partir des sources d'un système et de sa spécification, elle produit un programme instrumenté, contenant des assertions, de telle sorte que la validité de ce programme instrumenté par rapport aux assertions corresponde à celle du programme initial par rapport à sa spécification.

Construire une transformation de source à source présente de plus un certain nombre d'avantages. Elle est en grande partie indépendante du model-checker utilisé pour effectuer la vérification, ce qui permet d'obtenir un outil compatible avec plusieurs model-checker en arrière-plan. On peut ainsi les comparer et sélectionner le plus adapté pour un problème spécifique. Utiliser une instrumentation du code simplifie aussi les tâches d'implémentation, en nous évitant de composer avec le code d'un système préexistant et la représentation interne d'un model-checker. Cependant, ce choix a aussi des inconvénients. En particulier, une instrumentation représente une perte de précision et de performance par rapport à un traitement direct : des variables et des instructions sont ajoutées dans le système et vont être traitées comme n'importe quelle instruction par les model-checkers en backend, alors qu'elles suivent une logique qui pourrait être utilisée pour les traiter plus efficacement.

La transformation de source à source que nous proposons s'inspire fortement de celle mise en place dans [41]. Elle se base sur la composition d'un automate de Büchi représentant la négation de la propriété LTL à prouver avec le programme à valider. Un model-checker doit ensuite être utilisé pour explorer le produit du système avec l'automate. La composition du système avec un automate de Büchi est une technique classique pour le model-checking de propriétés LTL[30]. La particularité de notre approche est de construire ce produit au niveau du code source du système, et non pas au niveau de la représentation interne d'un

model-checker (où le système et l'automate sont vus comme des systèmes de transitions).

La figure 4.1 présente le fonctionnement de l'outil que nous avons conçu, nommé *baProduct*. Trois phases principales apparaissent :

1) Construire un automate de Büchi à partir de la spécification, et le convertir en code C ;
2) Instrumenter le code source du système en entrée pour construire le produit entre le système et l'automate de Büchi ;
3) Vérifier le code produit à l'aide d'un model-checker et analyser les résultats.

Dans la suite de ce chapitre, nous allons détailler ces trois étapes. Nous présentons ensuite notre implémentation, puis une analyse de performances sur une série d'exemples avec différents model-checkers utilisés pour les tâches de vérification.

Hypothèses Nous réalisons un ensemble d'hypothèses sur les systèmes que nous manipulons afin de simplifier la transformation de source à source. Il s'agit de restrictions classiques, souvent utilisées par les model-checkers logiciels.

Nous supposons que le système vérifie les propriétés suivantes :

— **Instructions atomiques** : nous supposons que les changements de contexte n'ont lieux que entre deux instructions. Ils ne peuvent pas avoir lieu pendant l'évaluation d'une instruction, même si elle est composée de plusieurs expressions.
— **Consistance séquentielle** : les écritures et les lectures sont supposées avoir lieu dans l'ordre où les instructions correspondantes sont rencontrées dans une trace d'exécution. Les optimisation du compilateur et du processeur, qui peuvent changer cela, ne sont pas considérées.

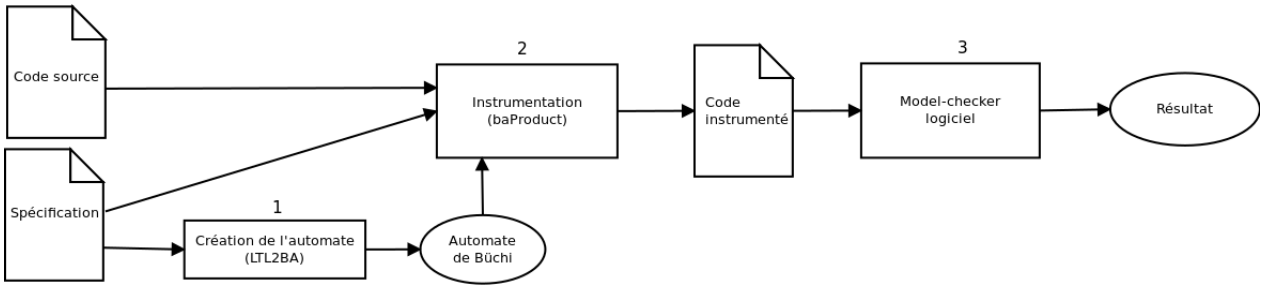

Figure 4.1 Schéma fonctionnel de baProduct

— **Sémantique d'entrelacement** : on suppose qu'une trace est un entrelacement des instructions atomiques de chaque thread, mais que deux instructions ne sont jamais exécutées simultanément. Cela correspond à l'exécution d'un programme concurrent sur un seul cœur du processeur. Dans le cas d'un système multi-cœur, cette hypothèse n'est généralement pas respectée.

4.1 Construction de l'automate de Büchi

Nous reprenons une technique classique de vérification des propriétés LTL, utilisée en particulier par SPIN[33] : construire un automate de Büchi représentant la négation de la propriété LTL et le composer avec le système. Une exécution acceptée par le système résultant est un contre-exemple à la spécification.

Mais alors que de manière générale, le programme à vérifier est représenté par un système de transitions avant d'être composé avec l'automate de Büchi, nous allons ici représenter l'automate de Büchi en C avant de le composer avec le programme.

Des algorithmes efficaces et éprouvés permettent de construire l'automate de Büchi représentant une formule LTL. En particulier, l'outil LTL2BA[28] est largement reconnu et utilisé : cet outil propose un algorithme de construction de l'automate notablement plus performant que les algorithmes utilisé auparavant, et il est éprouvé de part son utilisation dans de nombreux projets de recherche. Nous allons donc utiliser LTL2BA afin de générer la structure des automates et nous concentrer sur l'implémentation de l'automate en C.

Cependant, vouloir vérifier des propriétés LTL à l'aide d'une instrumentation à une conséquence : il n'est possible de vérifier que des traces finies. En effet, les model-checkers utilisent la présence de cycles dans les traces infinies pour les traiter : une fois le cycle atteint, le comportement du programme est totalement connu et il est possible de conclure si oui ou non, il vérifie une spécification. Mais du point de vue d'une instrumentation, il n'est pas possible de détecter un cycle, et il n'est donc pas possible de déterminer à quel moment prononcer une conclusion pour une trace infinie : l'ensemble des comportements du programme n'a peut être pas encore eu lieu, un nouveau comportement pourrait modifier le verdict. Dans le cas d'une trace finie, par contre, il suffit de conclure à la fin de la trace. Pour une instrumentation, cela revient à placer une fonction de "conclusion" juste avant les points de sortie du programme.

Nous allons donc pour la suite nous restreindre à des traces finies. Puisque LTL est définie sur des traces infinies, nous allons voir tout d'abord comment interpréter une formule LTL dans le cas d'une trace finie, avant d'expliquer notre construction en C de l'automate de Büchi associé.

4.1.1 Traces finies et logique à 4 valeurs de vérité

Les propriétés LTL ne sont pas définies sur les traces finies. Plusieurs variantes de LTL existent pour répondre à ce problème (LTL3, FLTL, RV-LTL, infinite extension...), aucune ne faisant consensus.

Nous avons choisi l'approche de l'extension infinie (*stuttering*) : l'état final de la trace finie est répété à l'infini afin d'étendre la trace de manière infinie. Cette approche correspond bien avec un programme qui, lorsqu'il se termine, cesse de voir son état évoluer. Elle a de plus l'avantage de ne pas modifier les opérateurs LTL, ce qui nous permet d'utiliser les techniques de vérification pour LTL sans modifications. Une trace finie est un modèle d'une propriété LTL dans la sémantique avec *stuttering* si et seulement si son extension infinie est un modèle de la propriété.

Exemple Soient p et q deux états d'un système. Soit la trace finie $t = pqpqpqp$. Alors, l'extension infinie de t est la trace $t' = pqpqppppppp\ldots$.

t' ne vérifie pas Gp mais vérifie $F(Gp)$, il en va donc de même pour la trace finie t.

Il faut cependant noter que les différentes variantes de LTL pour des traces finies diffèrent principalement par la prise en charge de l'opérateur *next*. Dans le cadre du software model-checking, le sens donné à celui-ci est très variable et il est généralement déconseillé de l'utiliser.

Une trace finie ne correspond pas toujours à une exécution réelle du système. Il peut s'agir seulement d'un préfixe d'une exécution plus longue, par exemple, le nombre d'itérations d'une boucle a pu être limité. La prolongation de la sémantique d'extension infinie ne reflète donc pas forcément le comportement réel du système.

Exemple La trace $t = pqpqpqp$ de l'exemple précédent est peut-être le préfixe d'une trace $t'' = pqpqpq\ldots pq\ldots$, qui aurait été interrompue. Cette trace ne vérifie pas la propriété $F(Gp)$, contrairement à t', l'extension infinie de t.

Pour tenir compte de ce comportement, il est possible de raffiner davantage les résultats en faisant la distinction entre deux types de traces finies :

- le préfixe fini est suffisant pour déterminer si la trace est un modèle de la formule ou non, quelque soit l'extension (*stuttering* ou autre)
- le préfixe fini n'est pas suffisant, le résultat dépend de l'extension

Exemple Le préfixe fini $t = pqpqpqp$ est suffisant pour établir que t ne vérifie pas Gp, quelque soit l'extension. Pour $F(Gp)$, cependant, nous avons vu que le résultat dépend de

l'extension choisie.

Nous avons de plus utilisé une logique à 4 valeurs de vérité, comme décrite dans [41]. Nous pourrons ainsi, selon la catégorie d'une trace, affecter des valeurs de vérité différentes.

Étant donné une propriété et un automate de Büchi associé à cette propriété, on définit les quatre valeurs de vérité par :

- **True** : utilisée lorsque, étant donnée une trace finie `t`, toutes les traces ayant `t` pour préfixe sont acceptées par l'automate.
- **Maybe true** : utilisée lorsque, étant donnée une trace finie `t`, l'extension infinie de `t` en répétant son dernier état est acceptée par l'automate.
- **Maybe false** : utilisée lorsque étant donnée une trace finie `t`, l'extension infinie de `t` en répétant son dernier état est rejeté par l'automate mais que au moins une trace ayant `t` pour préfixe est acceptée.
- **False** : utilisée lorsque, étant donnée une trace finie `t`, toutes les traces ayant `t` pour préfixe sont rejetées par l'automate.

Les deux valeurs *True* et *False* permettent de conclure de manière certaine, lorsque le résultat ne dépend pas de l'extension. Si le résultat dépend de l'extension, les valeurs *Maybe true* et *Maybe false* permettent d'indiquer que le résultat est relatif à la sémantique d'extension infinie.

Dans les cas d'une conclusion *Maybe true* ou *Maybe false*, savoir si le programme a terminé ou non son exécution (i.e on a vérifié l'exécution complète ou la vérification a été interrompue parce qu'une borne sur la profondeur d'exploration a été atteinte) peut permettre de préciser le résultat.

4.1.2 Génération de l'automate de Büchi

La génération d'un automate de Büchi en C est la première étape réalisée par notre outil (étape 1 dans la Figure 4.1). Cet automate doit représenter la négation de la propriété LTL présente dans la spécification, et permettre de déduire un résultat dans la logique à 4 valeurs de vérité. L'implémentation de cet automate reprend l'approche décrite dans [41].

Les états de l'automate sont indicés selon un ordre arbitraire. On réfère à un état dans le code par son indice. La partie principale de l'implémentation de l'automate est sa fonction de transition. Elle est implémentée à travers une fonction C, qui permet de déterminer l'évolution de l'état de l'automate selon l'état courant et les valeurs des propositions atomiques.

Nous utilisons les variables et fonctions suivantes pour construire l'automate :

— une variable globale, `ltl2ba_state_var`, contient l'état courant de l'automate sous la forme d'un entier (les états sont numérotés arbitrairement)

— des variables globales, `ltl2ba_atomic_name` (où name est l'identifiant d'une proposition atomique), contiennent la valeur courante de chaque proposition atomique. Ces valeurs sont maintenues à jour pendant le déroulement du programme. Elles sont actualisées par des instructions ajoutées dans le code du programme que nous détaillerons dans la partie sur l'instrumentation.

— une fonction `_ltl2ba_transition` représente la fonction de transition de l'automate et fait évoluer l'état stocké dans `ltl2ba_state_var`. Étant donné l'état courant de l'automate, elle choisit de manière non déterministe une transition valide partant de cet état et met à jour l'état courant. Si aucune transition n'est valable, le mot est rejeté (et l'exploration courante est stoppée). Si l'état courant est un état puits acceptant, le mot est accepté et une erreur est immédiatement remontée (on rappelle que l'automate représente la négation de la propriété à vérifier, une trace acceptée par l'automate est donc un contre-exemple à la propriété).

  Cette fonction permet d'explorer l'automate de manière non déterministe. Une valeur non déterministe est produite l'aide de la fonction intrinsèque au model-checker, `nondet_int()`. La transition à emprunter est désignée à l'aide de cette valeur. Le model-checker va alors explorer l'ensemble des évolutions possibles de l'automate. Les évolutions invalides sont immédiatement interrompues à l'aide d'hypothèses (commande `assume`).

— une fonction `_ltl2ba_result` permet de déterminer le résultat d'une exploration. Elle est appelée après la dernière instruction du `main`. Elle se base sur l'état courant de l'automate et la valeur des propositions atomiques pour déterminer la valeur de vérité associée à cette exécution. Si une erreur est possible, elle est remontée au model-checker à l'aide d'une assertion. Les valeurs de vérité sont déterminées en utilisant des données précalculées par une analyse d'accessibilité dans l'automate de Büchi [41].

Notre implémentation de l'automate de Büchi en C diffère de celle de [41] par un point d'importance : dans [41], l'automate est implémenté dans un thread supplémentaire, utilisé comme un thread observateur. La fonction de transition de l'automate est placée dans une boucle, synchronisée avec le reste du code à l'aide d'une commande interne du model-checker. Cette commande a été implémentée pour l'occasion. Dans notre cas, la fonction de transition est représentée par une fonction dans le code, appelée quand nécessaire par l'instrumentation. Cela nous permet de ne pas nous baser sur une commande interne pour synchroniser l'automate avec le code et ainsi, de supporter plus facilement différents model-checkers en backend.

Exemple L'automate pour la propriété LTL $G(p \implies Fq)$ (représenté en Figure 4.2) est implémenté par le code du listing 4.1.

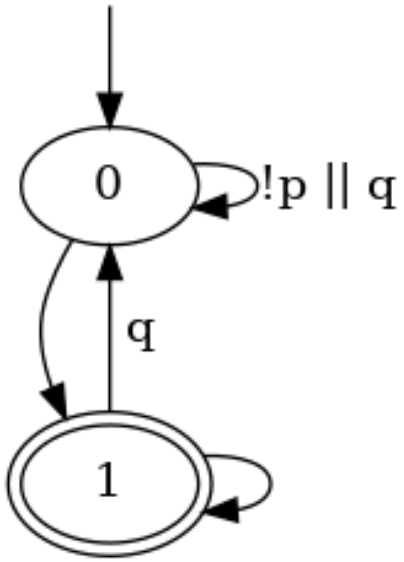

Figure 4.2 Automate de Büchi pour la propriété $G(p \implies Fq)$

Listing 4.1 Code de l'automate représentant $G(p \implies Fq)$

```
/* Status of atomic propositions */
_Bool _ltl2ba_atomic_p = 0;
_Bool _ltl2ba_atomic_q = 0;
/* Current state in the automaton */
int _ltl2ba_state_var = 0;
/* Transition function of the automaton */
void _ltl2ba_transition() {
  /* Non determinist choice of a transition */
  int choice = nondet_uint();
  /* Perform an action according to the current state */
  switch (_ltl2ba_state_var) {
  case 0: // From state 0
    if (choice == 0) { // Take the first transition
      /* Check if the guard is verified */
```

```
      assume(!_ltl2ba_atomic_p);
      /* Take the transition and update the current state */
      _ltl2ba_state_var = 0;
    } else if (choice == 1) { // Take the second transition
      /* Check if the guard is verified */
      assume(1);
      /* Take the transition and update the current state */
      _ltl2ba_state_var = 1;
    } else if (choice == 2) { // Take the third transition
      assume(_ltl2ba_atomic_q);
      _ltl2ba_state_var = 0;
    } else {
      assume(0);
    }
    break;
  case 1: // From state 1
    if (choice == 0) {
      assume(1);
      _ltl2ba_state_var = 1;
    } else if (choice == 1) {
      assume(_ltl2ba_atomic_q);
      _ltl2ba_state_var = 0;
    } else {
      assume(0);
    }
    break;
  }
}
/* States of the automaton from which every suffix is
   accepted by the automaton */
_Bool _ltl2ba_surely_accept[2] = {0, 0};
/* States of the automaton from which every suffix is
   rejected by the automaton */
_Bool _ltl2ba_surely_reject[2] = {0, 0};
/* For each combination of automaton state and atomic
   proposition value give the sttuter acceptance result. */
```

```
_Bool _ltl2ba_stutter_accept[8] = {1,0, 0,0, 1,1, 1,1,};
unsigned int _ltl2ba_sym_to_id() {
  unsigned int id = 0;
  id |= (_ltl2ba_atomic_p << 0);
  id |= (_ltl2ba_atomic_q << 1);
  return id;
}
/* Report results to the model-checker using assertions */
void _ltl2ba_result() {
  /* The automaton reject the execution: there is no error */
  _Bool reject_sure = _ltl2ba_surely_reject[_ltl2ba_state_var];
  /* The automaton accept the execution: there is an error */
  assume(!reject_sure);
  _Bool accept_sure = _ltl2ba_surely_accept[_ltl2ba_state_var];
  assert(!accept_sure, "ERROR SURE");
  /* The automaton stutter accept the execution:
     there may be an error */
  unsigned int id = _ltl2ba_sym_to_id();
  _Bool accept_stutter =
    _ltl2ba_stutter_accept[id * 2 + _ltl2ba_state_var];
  assert(!accept_stutter, "ERROR MAYBE");
  assert(accept_stutter, "VALID MAYBE");
}
```

Cet exemple permet de remarquer que le seul moyen de faire remonter un résultat est à travers le déclenchement d'une assertion. La violation d'une assertion met fin à une trace, il est donc important d'ordonner les assertions de manière à ce que le résultat le plus fort soit le premier à être reporté pour qu'il ne soit pas masqué. De plus, certains model-checkers arrêtent d'explorer le modèle dès qu'une erreur est reportée. Dans ce cas, il peut être nécessaire de réaliser plusieurs vérifications successives en désactivant les assertions les plus faibles, pour qu'un résultat faible ne masque pas un résultat plus fort présent dans une trace future.

4.2 Instrumentation du code

Une fois l'automate implémenté, la seconde étape du processus est de construire le produit entre le système et l'automate (étape 2 dans la Figure 4.1). Cela est réalisé par une instru-

mentation du code, afin de maintenir à jour la valeur des propositions atomiques et d'appeler la fonction de transition de l'automate.

Afin de minimiser l'instrumentation, la valeur d'une proposition atomique est mise à jour uniquement lorsqu'un des paramètres dont elle dépend est modifié (valeur d'une variable, entrée ou sortie de la zone de validité). La fonction de transition de l'automate est appelée uniquement à ces occasions. L'opérateur LTL *Next* a alors pour signification "lors du prochain appel à la fonction de transition", c'est à dire la prochaine fois qu'une proposition atomique est susceptible d'être modifiée.

Différents emplacements du code vont nécessiter une instrumentation, qui va aussi dépendre du type de paramètre des propositions atomiques. Nous allons expliquer l'instrumentation nécessaire dans ces différents cas.

4.2.1 Frontière des zones de validité

Les premiers points nécessitant une instrumentation sont les frontières des zones de validité des propositions atomiques. Lorsqu'un pointeur d'instruction atteint l'instruction portant le label d'entrée (respectivement de sortie) d'une zone de validité, la proposition atomique doit prendre la valeur calculée par sa fonction d'évaluation (respectivement sa valeur par défaut).

Les listings 4.2 et 4.3 présentent l'instrumentation correspondant à l'entrée et à la sortie des zones de validités. Ici, la proposition atomique `p` débute au label `lbegin` et se termine au label `lend`.

Listing 4.2 Entrée d'une zone de validité (1)

```
1 __atomic_begin();
2 ltl2ba_atomic_p = fp(..);
3 _ltl2ba_transition();
4 __atomic_end();
5 lbegin: ....;
```

Listing 4.3 Sortie d'une zone de validité (1)

```
1 __atomic_begin();
2 ltl2ba_atomic_p = {v_default};
3 _ltl2ba_transition();
4 __atomic_end();
5 lend: ....;
```

Toutes les instructions sont regroupées dans un bloc atomique (ouvert par l'instruction `__atomic_begin()`, ligne 1) et fermé par l'instruction `atomic_end()`, ligne 4) afin de ne pas générer d'entrelacements supplémentaires dans le code et créer des chemins d'exécutions pouvant mener à une conclusion erronée : toutes les instructions de l'instrumentation doivent être réalisées de manière atomique (sans changement de contexte possible) pour garantir un état cohérent de l'automate. Si plusieurs entrées et sorties d'une zone de validité ont lieu au même label, les instructions d'instrumentations sont toutes réunies dans le même bloc atomique. La fonction de transition n'est appelée qu'une seule fois (à la ligne 3), après

l'actualisation de toutes les propositions atomiques (à la ligne 2).

4.2.2 Variables locales

Passons à la seconde raison pouvant amener à modifier la valeur d'une proposition atomique. Il s'agit de la modification d'une variable dont dépend la proposition atomique, dans sa zone de validité (en dehors de la zone de validité, la proposition atomique s'évalue toujours à sa valeur par défaut, la modification d'une variable n'a donc pas d'impact). Nous allons tout d'abord nous concentrer uniquement sur le cas des variables locales à une fonction.

Si une variable locale est modifiée dans la zone de validité d'une proposition atomique et que la valeur de la proposition atomique dépend de la variable, il est toujours nécessaire d'instrumenter l'affectation. Instrumenter ces appels est aussi suffisant (car nous ne considérons pas les accès indirects aux variables). La variable étant locale, elle ne peut s'échapper hors de son contexte : elle ne peut donc pas être modifiée par une instruction hors de la zone de validité alors qu'un pointeur d'instruction est dans la zone de validité.

On peut alors ajouter une instrumentation autour de l'affectation. Après la modification de la variable, les propositions atomiques qui en dépendent sont évaluées à nouveau. On tente ensuite d'effectuer une transition dans l'automate de Büchi.

Listing 4.4 Instrumentation pour une variable locale

```
1 __atomic_begin();
2 v = ....;
3 ltl2ba_atomic_p = fp(v, ...);
4 _ltl2ba_transition();
5 __atomic_end();
```

L'affectation de la variable à sa nouvelle valeur et l'instrumentation sont placées dans un même bloc atomique, afin de répercuter la modification de l'état sans changement de contexte possible.

Contrairement à l'instrumentation à l'entrée et la sortie des zones de validité, ici, aucun nouveau chemin d'exécution n'est créé : une affectation est une instruction, que nous considérons comme atomique. Il est donc possible de l'inclure dans un bloc atomique pour le model-checker.

4.2.3 Variables globales

Passons maintenant au cas où la proposition atomique ne dépend que de variables globales.

Tout comme une variable locale, si une variable globale est modifiée dans la zone de validité d'une proposition atomique et que cette proposition atomique dépend de la variable globale, il est nécessaire d'actualiser la valeur de la proposition atomique et d'effectuer une transition dans l'automate. Mais une variable globale peut aussi affecter la valeur d'une proposition atomique lors d'une affectation hors de la zone de validité de cette proposition : si un pointeur d'exécution du programme est dans la zone de validité et qu'une variable globale dont dépend la proposition est modifiée dans un autre thread, cette modification a un impact sur la valeur de la proposition atomique.

Exemple Les listings 4.5 et 4.6 illustrent ce problème.

On désire s'assurer que la variable `a` n'est pas nulle dans la zone de validité allant de `begin` à `end` dans le second thread. L'affectation `a = b;` est susceptible de faire évoluer cette propriété, il est donc nécessaire de l'instrumenter, tout comme dans le cas des variables locales.

L'affectation `a = 0` dans le premier thread peut aussi avoir un impact sur la propriété, mais cela dépend de l'ordre d'exécution des instructions :

— si l'ordre d'exécution est `a = b; a = 0; r = 10 / a`, alors l'affectation `a = 0;` du second thread doit impacter la valeur de la proposition atomique : `a` est maintenant

Listing 4.5 Thread 1

```
1 int a = 1;
2 /* Atomic proposition */
3 int pred(int a) {
4     return a != 0;
5 }
6 void* thread1(void* d) {
7     a = 0;
8 }
```

Listing 4.6 Thread 2

```
1 void* thread2(void* d) {
2     int b = *(int *)d;
3     int r;
4     begin: ;
5     a = b;
6     r = 10 / a;
7     end: ;
8 }
```

nulle dans la zone de validité.

— si le second thread s'exécute entièrement avant le premier, quand `a = 0;` est exécuté, la proposition atomique n'est pas dans sa zone de validité (le pointeur d'exécution du premier thread n'a pas encore atteint le label `begin`). Dans ce cas, la valeur de la proposition atomique ne doit pas être modifiée.

Il est donc nécessaire d'instrumenter aussi les affectations à des variables globales hors de la zone de validité d'une proposition. Cependant, une affectation en dehors d'une zone de validité ne doit pas non plus systématiquement donner lieu à une actualisation des propositions atomiques : il ne faut actualiser la valeur de la proposition atomique que lorsqu'un pointeur d'exécution est dans sa zone de validité. C'est donc une notion dynamique, qui dépend de l'exécution.

Pour n'actualiser la valeur que lorsque c'est nécessaire, l'instrumentation prendra la forme d'une condition. Nous ajoutons une variable globale pour chaque proposition atomique, qui indique si le programme est dans sa zone de validité. On maintient la valeur de cette variable à l'entrée et à la sortie de chaque zone de validité. On peut alors tester la valeur de cette variable pour définir s'il faut actualiser ou non la proposition atomique.

La valeur de cette nouvelle variable globale doit être actualisée à l'entrée et à la sortie des zones de validités. L'instrumentation de ces zones devient donc :

Listing 4.7 Entrée d'une zone de validité (2)

```
1 __atomic_begin();
2 _ltl2ba_active_p = 1;
3 ltl2ba_atomic_p = fp(..);
4 _ltl2ba_transition();
5 __atomic_end();
6 lbegin: ....;
```

Listing 4.8 Sortie d'une zone de validité (2)

```
1 __atomic_begin();
2 _ltl2ba_active_p = false;
3 ltl2ba_atomic_p = {v_default};
4 _ltl2ba_transition();
5 __atomic_end();
6 lend: ....;
```

On remarque, à la seconde ligne des listings 4.3 et 4.8, que la variable `_ltl2ba_active_p` est mise à jour.

On utilise alors cette information dans l'instrumentation lors de la modification d'une variable globale, qui prend alors la forme suivante :

1) Pour des instructions situées dans la zone de validité de la proposition :

Listing 4.9 Instrumentation pour une variable globale dans la zone de validité

```
1 __atomic_begin();
2 g = ....;
3 ltl2ba_atomic_p = fp(g, ...);
4 _ltl2ba_transition();
5 __atomic_end();
```

2) Pour des instructions situées hors de la zone de validité de la proposition :

Listing 4.10 Instrumentation pour une variable globale hors de la zone de validité

```
1   __atomic_begin();
2   g = ....;
3   if (_ltl2ba_active_p) {
4     ltl2ba_atomic_p = fp(g, ...);
5     _ltl2ba_transition();
6   }
7   __atomic_end();
```

On retrouve la même instrumentation que dans le cas des variables locales, mais la troisième ligne du listing 4.10 utilise la variable `_ltl2ba_active_p` pour déterminer comment évaluer la proposition atomique.

4.2.4 Combinaison de variables globales et locales

Enfin, plaçons-nous dans le cas où une proposition atomique dépend à la fois de variables globales et de variables locales.

Un nouveau problème apparaît : lorsqu'une variable globale est modifiée hors de la zone de validité, il peut être nécessaire de mettre à jour la proposition atomique à l'aide de sa fonction d'évaluation, comme nous l'avons vu précédemment. Cette fonction prend en paramètre les variables locales dont dépend la proposition atomique. Mais ces variables locales ne sont pas

dans le contexte de l'appel ! Celui-ci est réalisé depuis un autre thread, depuis lequel ces variables locales ne sont pas accessibles.

Exemple Dans le code suivant, on veut vérifier que les variables `a` et `b` sont bien égales durant tout l'intervalle de validité de la proposition.

Listing 4.11 Thread 1

```
1 int a = 1;
2 int pred(int a, int b) {
3   return a == b;
4 }
5 void* thread1(void* d) {
6   a = 0;
7 }
```

Listing 4.12 Thread 2

```
1 void* thread2(void* d) {
2   int b = *(int *)d;
3   int r;
4   begin: a = b;
5   r = 10 / a;
6   end: ;
7 }
```

Le second thread serait instrumenté de la manière suivante :

Listing 4.13 Thread 2 instrumenté

```
1  void* thread2(void* d) {
2    __atomic_begin();
3    a = 0;
4    if (_ltl2ba_active_p) {
5      // Here, b is not in the context
6      ltl2ba_atomic_p = pred(a, b);
7      _ltl2ba_transition();
8    }
9    __atomic_end();
10 }
```

On remarque que la variable `b` n'est pas accessible lorsqu'il est nécessaire d'appeler la fonction d'évaluation, à la ligne 6.

Cependant, on sait que lorsque cet appel a lieu, un pointeur d'exécution du programme est dans la zone de validité de la proposition : toutes les variables locales dont dépend la proposition sont donc dans la pile. On peut alors y accéder de manière indirecte, en utilisant des pointeurs globaux contenant les adresses de ces variables. Nous allons donc compléter l'instrumentation pour maintenir des adresses de toutes les variables utilisées comme paramètre d'une proposition atomique.

Pour chaque variable locale utilisée comme paramètre d'une proposition atomique, une variable globale supplémentaire est créée. À l'entrée dans la zone de validité d'une proposition, cette variable globale est assignée avec l'adresse du paramètre auquel elle correspond. Il est alors possible d'accéder à la valeur de la variable en déréférençant ce pointeur global.

Exemple On obtient enfin la version finale de l'instrumentation pour l'exemple précédent.

Listing 4.14 Instrumentation finale pour des variables locales et globales

```
1  int a = 1;
2  // Are we in p validity area ?
3  int _ltl2ba_active_p = 0;
4  // Pointer to local variable b
5  int *_ltl2ba_ptr_b;
6  // Atomic proposition p current value
7  int _ltl2ba_atomic_p = {v_default};
8  // Evaluation function
9  int pred(int a, int b) { return a == b; }
10
11 void* thread1(void* d) {
12   // Open atomic bloc
13   __atomic_begin();
14   a = 0;
15   // a is global: check if p is in its validity area
16   if (_ltl2ba_active_p) {
17     // Update p value and take a transition.
18     // We get the local variable b through a pointer
19     ltl2ba_atomic_p = pred(a, *_ltl2ba_ptr_b);
20     _ltl2ba_transition();
21   }
22   __atomic_end();
23 }
24 void* thread2(void* d) {
25   int b = *(int *)d;
26   int r;
27   // Begining of a validity area
28   begin: __atomic_begin();
29   // Set the address of b in the global pointer
```

```
30    _ltl2ba_ptr_b = &b;
31    // Mark the proposition as active, compute its value
32    // and take a transition
33    _ltl2ba_active_p = 1;
34    ltl2ba_atomic_p = pred(a, *_ltl2ba_prt_b);
35    _ltl2ba_transition();
36    __atomic_end();
37    a = b;
38    r = 10 / a;
39    // End of the validity area
40    __atomic_begin();
41    // Mark the proposition as inactive, set its value to
42    // the default value and take a transition
43    _ltl2ba_active_p = 0;
44    ltl2ba_atomic_p = {default_val};
45    _ltl2ba_transition();
46    __atomic_end();
47    end: ;
48 }
```

À ce niveau, l'instrumentation est terminée. Le code source obtenu, composé du système instrumenté et de l'automate de Büchi peut être vérifié.

4.3 Vérification à l'aide d'un model-checker

Nous en sommes enfin à l'étape 3 du processus présenté par la Figure 4.1. Nous avons produit, à partir de la spécification et du code du système, un nouveau code spécifié par des assertions.

Un model-checker permet alors de vérifier ce code. Selon les assertions que le code va violer, il est possible de déterminer si le programme contient une erreur ou non. Le contre-exemple fourni par le model-checker permet de retrouver l'erreur dans le programme instrumenté, et donc, en ignorant les lignes ajoutées par l'instrumentation, dans le système d'origine.

Certains model-checkers arrêtent l'exploration du système dès qu'une assertion est violée. Un résultat plus faible que le résultat attendu peut alors être remonté, si une trace violant l'assertion correspondant au résultat plus faible est explorée avant une trace prouvant le résultat plus fort. Il peut donc être nécessaire de réaliser plusieurs passes de vérification en désactivant certaines assertions pour s'assurer d'obtenir le résultat le plus fort.

4.4 Implémentation

Nous avons implémenté cette transformation de source à source dans un outil, `baProduct`.

Nous générons les automates de Büchi correspondant aux formules LTL en utilisant le logiciel LTL2BA[28]. LTL2BA implémente une méthode rapide afin de construire l'automate de Büchi associé à une formule LTL, en utilisant un co-automate de Büchi alternant très faible (*very weak alternating co-Büchi automaton*) et un automate de Büchi généralisé (*generalized Büchi automaton*) comme intermédiaires.

L'instrumentation est réalisée à l'aide la bibliothèque CIL[44], en OCaml[37]. CIL permet de parser du code C et de construire un arbre de syntaxe abstraite plus concis que celui du C mais d'un niveau plus haut que celui des langages intermédiaires utilisés dans les procédés de compilation. La bibliothèque ocamlgraph[21] est utilisée pour réaliser l'analyse d'accessibilité dans les automates.

Notre implémentation comprend :

— la version modifiée de LTL2BA. Son fonctionnement est identique à celui de la version d'origine, seule une option `-t` a été ajoutée afin de choisir le type de la sortie. Cette option peut prendre les valeurs `spin`, `c` ou `json` pour obtenir un automate en Promela, C et JSON respectivement.

— L'utilitaire `baProduct`. Il s'agit du programme réalisant l'instrumentation. Il prend en entrée un fichier C préprocessé et un fichier de spécification, et il produit un fichier instrumenté. Cet utilitaire a besoin de la version modifiée de LTL2BA pour fonctionner.

— Un script de lancement, `baProduct.py`. Il préprocesse les fichiers en entrée et appelle `baProduct` avec un ensemble d'options classiques.

Une utilisation classique de notre implémentation ressemble donc à l'appel suivant (avec ESBMC[24] utilisé en backend) :

```
~ ./baProduct.py -i test.c -s test.spec -o test_instr.c
~ esbmc --depth 100 test_instr.c
```

Le code de ces outils et une série de tests sont disponibles sur Github, aux adresses https://github.com/xNephe/baProduct et https://github.com/xNephe/ltl2ba.

4.5 Résultats expérimentaux

Afin de tester notre approche, nous avons mis en place une suite de tests permettant d'évaluer les performances de vérification du code instrumenté avec plusieurs model-checkers. Les tests mettent en jeu des erreurs liées au non-déterminisme de contrôle (i.e. les différents entrelacements possibles dans le système). Ils modélisent des versions très simplifiées de systèmes usuels, parmi lesquels un feu tricolore, un système producteur-consommateur ou un système d'alimentation.

Nous avons réalisé les tests sur un ordinateur muni de 6 Go de RAM, équipé d'un processeur Intel Core i7 (4 coeurs cadencés à 3,7GHz), sous Fedora 25 (64 bits). Nous avons utilisé CBMC[19] version 5.7 et ESBMC[24] version 4.2. ESBMC nécessite une borne sur le nombre de changements de contextes, que nous avons fixé à 3, sauf pour le test *prod_cons_simple*, où elle est de 7 (en raison d'un plus grand nombre de threads). En présence de boucles, nous avons limité l'exploration à trois itérations.

CBMC et ESBMC étant des model-checkers bornés, ils se combinent très bien avec notre approche, qui exige une exécution finie. Nous avons tenté d'utiliser d'autres types de model-checkers pour effectuer les tâches de vérification (en particulier Threader, Divine et Lazy CSeq). Cependant, ces essais n'ont pas été fructueux. Nous avons rencontré des difficultés techniques lors de l'installation ou la compilation de ces outils, ainsi que des erreurs de parsing du code instrumenté. Pour Threader, l'incompatibilité vient de l'approche choisie : Threader ne considère pas de fonction `main` dans un programme et vérifie uniquement les threads alors que notre transformation nécessite qu'un programme commence et se termine dans la fonction `main`.

4.5.1 Scénarios de test

Nous avons utilisé les scénarios de tests suivants lors de nos expérimentations. Nous nous sommes concentré sur des situations impliquant des problèmes de concurrence, et en particulier sur des erreurs liées à une absence ou une mauvaise utilisation des primitives de synchronisation.

battery_simple (50 lignes, 2 threads) Deux batteries, chacune représentée par un thread, sont supposées alimenter un système critique. Une variable locale représente le niveau d'énergie contenu dans chaque batterie, elle est décrémentée dans une boucle puis remise à sa valeur maximale lorsqu'elle atteint zéro. Dans ce test, on désire éviter que les deux batteries soient en recharge en même temps : il s'agit donc d'une propriété d'exclusion mutuelle.

battery_var (51 lignes, 2 threads) Dans la même situation que précédemment, on désire maintenant vérifier que les deux batteries ne contiennent pas un niveau d'énergie faible simultanément. La taille des zones de validité est alors plus importante, et la valeur des variables représentant le niveau d'énergie doit alors être pris en compte, ce qui augmente la taille du système.

Ces deux tests permettent de mettre en évidence l'intérêt de la méthode de spécification que nous avons présentée, en particulier dans le cas de propriétés d'exclusion mutuelle.

crossing_exclusive_green (88 lignes, 2 threads) Le système modélise deux feux de circulation à un croisement. Chaque feu est représenté par un thread. Les feux doivent prendre les couleurs rouge, orange et verte. Ils sont synchronisés à l'aide de mutex. Dans ce premier cas, on veut vérifier que les deux feux ne sont jamais verts de manière simultanée.

crossing_exclusive_mutex (88 lignes, 2 threads) Dans la même situation que précédemment, on veut maintenant vérifier que à tout moment, au moins un feu est rouge.

crossing_GF_green (88 lignes, 2 threads) Dans la même situation que précédemment, on veut désormais vérifier que chaque feu finira par devenir vert (il n'y a pas de famine dans le système).

crossing_order (94 lignes, 2 threads) Dans la même situation, on vérifie que chaque feu respecte l'ordre des couleurs : vert $\rightarrow$ orange $\rightarrow$ rouge.

Cette série de test sur le modèle du feu de circulation permet de tester un ensemble de propositions LTL classiques et fréquemment utilisées.

answer_simple (48 lignes, 2 threads) Deux threads modélisent un client et un serveur, qui échangent un message à travers deux variables partagées. Dans ce premier cas, des mutex sont utilisés de manière incorrecte pour protéger ces variables. On vérifie si l'envoie d'une requête par le client est bien systématiquement suivie d'une réponse du serveur.

answer_simple_valid (50 lignes, 2 threads) La même situation que précédemment est utilisée, mais les variables partagées sont correctement protégées par les mutex.

prod_cons_simple Ce scénario reprends une situation de producteur / consommateur, avec un seul producteur et un seul consommateur, qui incrémentent et décrémentent une

variable partagée. Dans ce cas de figure, on tente de synchroniser le système sans primitives de synchronisation. On vérifie que on ne consomme jamais plus d'éléments qu'il n'y a eu de production.

prod_cons_mutex On se place dans la même situation que précédemment, en protégeant la variable partagée à l'aide de mutex.

Les quatre tests précédents se basent sur des problèmes classiques, connu pour poser fréquemment des problèmes de synchronisation. Nous en étudions donc deux versions, une correcte et une erronée afin de vérifier la pertinence des résultats de notre outil.

race Deux threads sont en compétition afin de modifier, dans un ordre opposé, deux variables partagées. L'objectif est de déterminer la valeur contenue à l'état final dans la seconde variable, selon l'ordre dans lequel les thread modifient la première variable.

Cette situation donne l'opportunité de manipuler une formule LTL plus complexe que dans les scénarios précédents.

Les tableaux 4.1 et 4.2 présentent les résultats que nous avons obtenus.

Comme on peut le constater dans le tableau 4.1, CBMC et ESBMC permettent généralement de mener à bien la tâche de vérification et de trouver un résultat correct. On observe cependant quelques erreurs. CBMC indique une erreur possible dans un test valide, alors que ESBMC ne détecte pas une erreur dans un test incorrect. ESBMC échoue aussi sur le test *battery_var* après avoir consommé l'ensemble de la mémoire disponible. Ces résultats ne sont pas liés à notre instrumentation mais au fonctionnement interne des model-checkers, et sont probablement dus à une perte de précision dans les abstractions du système. Le test *battery_var* est l'un de ceux nécessitant la plus grande profondeur d'exploration et générant le plus d'entrelacement, ce qui explique le temps nécessaire à CBMC pour conclure, et l'erreur d'exécution de ESBMC.

Les résultats attendus et ceux fournis par les model-checkers sont généralement incertains (`MAYBE`). Cela est normal, puisqu'on considère des propriétés de sûreté ou d'accessibilité sur des programmes finis, pour lesquelles on ne peut pas conclure de manière certaine. Le fait de savoir que le programme a terminé son exécution dans ces cas permet de rendre le verdict définitif.

Pour les mesures de performances, nous ne tenons compte que du temps utilisé par le model-checker pour vérifier le fichier instrumenté. Le temps pris par l'instrumentation est négligeable (moins d'une seconde). Si plusieurs passes de vérification sont nécessaires pour s'assurer

Tableau 4.1 Résultats de la vérification d'un benchmark instrumenté par baProduct

Scénario de test	CBMC	ESBMC	Résulat attendu
answer_simple	ERROR MAYBE	ERROR MAYBE	ERROR MAYBE
answer_simple_valid	**ERROR MAYBE**	VALID MAYBE	VALID MAYBE
battery_simple	ERROR SURE	ERROR SURE	ERROR SURE
battery_var	ERROR SURE	* [1]	ERROR SURE
crossing_exclusive_green	VALID MAYBE	VALID MAYBE	VALID MAYBE
crossing_exclusive_mutex	VALID MAYBE	VALID MAYBE	VALID MAYBE
crossing_GF_green	ERROR MAYBE	ERROR MAYBE	ERROR MAYBE
crossing_order	VALID MAYBE	VALID MAYBE	VALID MAYBE
prod_cons_mutex	VALID MAYBE	VALID MAYBE	VALID MAYBE
prod_cons_simple	ERROR SURE	**VALID SURE**	ERROR SURE
race	ERROR MAYBE	ERROR MAYBE	ERROR MAYBE

Tableau 4.2 Performances pour la vérification d'un benchmark instrumenté par baProduct

	CBMC		ESBMC	
Scénario de test	Temps (s)	Mémoire (MB)	Temps (s)	Mémoire (MB)
answer_simple	0.87	42	5.57	56
answer_simple_valid	0.81	41	12.91	72
battery_simple	1.07	42	13.95	61
battery_var	666.04	954	N/A	OOM
crossing_exclusive_green	2.93	44	32.12	119
crossing_exclusive_mutex	1.73	44	32.20	120
crossing_GF_green	1.99	46	41.70	117
crossing_order	1.02	40	11.62	56
prod_cons_mutex	10,046.32	593	229,99	28
prod_cons_simple	1.70	52	454.00	2711
race	1.08	41	5.65	28

qu'une assertion n'en masque pas une autre, nous considérons la somme des temps d'exécution. Le tableau 4.2 permet de constater que CBMC est bien plus efficace sur ce benchmark que ESBMC. Nous interprétons cet écart de performances par la manière dont ces deux model-checkers prennent en charge la concurrence. CBMC construit une unique formule représentant tous les entrelacements, qu'il délègue ensuite à un SAT solveur, alors que ESBMC explore les entrelacements de manière explicite. Les erreurs dans nos tests sont principalement basées sur un entrelacement particulier des threads, ce qui est plus compatible avec l'approche de CBMC. On peut cependant noter une très bonne performance de ESBMC sur le test *prod_cons_mutex*, sur lequel CBMC est extrêmement lent. Dans ce cas, ESBMC a probablement eu la chance d'explorer rapidement un entrelacement permettant de conclure.

4.6 Limitations et pistes d'améliorations

La transformation de source à source et l'outil que nous présentons dans ce chapitre restent limités. Parmi ces limitations, nous pensons que certaines pourraient être surmontées relativement facilement par des travaux futurs qui sortaient du cadre de cette maîtrise. D'autres limitations sont liées au choix de construire une instrumentation du code plutôt qu'étendre directement un model-checker. Elles pourraient donc être contournées en changeant d'approche. Enfin, certaines sont des problèmes plus complexes et difficiles à surmonter.

La principale limitation de notre travail est qu'il est limité à l'analyse de programmes dont toutes les traces terminent, afin que les appels à la fonction d'analyse des résultats soient examinés. Cette restriction est inhérente à une instrumentation, une extension d'un model-checker ne rencontrerait pas ce problème : l'état final de l'automate peut être examiné une fois l'exploration du système terminée, qu'il contienne ou non des traces infinies.

Nous n'avons pas abordé la question des accès indirects aux variables (à travers un pointeur ou un tableau par exemple). Ces derniers pourraient être pris en charge à l'aide d'une instrumentation plus lourde, se basant sur une analyse statique des cibles possibles des pointeurs et procédant ensuite par disjonction de cas.

Notre instrumentation provoque potentiellement des pertes de précision dans la vérification, en introduisant des chemins dans le programme instrumenté qui n'étaient pas présent dans le programme initial. Ce phénomène est présent à l'entrée des zones de validité, où l'instrumentation ne peut être regroupée dans un bloc atomique avec l'instruction la précédant immédiatement dans une exécution. Cette perte de précision ne nous a pas posé de problèmes au cours de nos tests. Une procédure de vérification de notre spécification directement intégrée dans un model-checker ne rencontrerait probablement pas cette perte de précision.

Nous n'avons pas cherché à optimiser les performances de notre outil ou à lutter contre l'explosion combinatoire. En particulier, nous n'utilisons pas de techniques de réduction ou d'ordre partiels dans notre instrumentation. Celles utilisées par les model-checkers en backend n'ont pas connaissance de la logique de l'instrumentation et de la spécification, ce qui peut limiter leur efficacité. Une extension à notre travail serait de mettre en place des techniques de réduction tenant compte de la spécification et de l'instrumentation afin de limiter l'exploration de chemins d'exécution équivalents. Rendre atomiques les zones du programme sans impact sur les valeurs des propositions atomiques pourrait être un premier pas. Une autre approche pourrait être de restreindre l'automate de Büchi composé avec le système afin d'éviter l'exploration de branches entières de l'arbre des exécutions. La vérification ne serait cependant plus complète, mais certaines erreurs pourraient être détectées plus efficacement. Il est déjà possible de restreindre manuellement l'automate de Büchi dans le code instrumenté.

Nous n'apportons pas de solution au problème de désignation des variables locales, dans le cas où plusieurs instances d'une même variable lexicale cohabitent. Intégrer l'approche choisie par Divine[5] pourrait constituer une solution partielle.

Enfin, notre formalisme de spécification reste très verbeux, ce qui nuit à sa clarté. Mettre en place une syntaxe plus concise afin de rassembler les constituants d'une proposition atomique permettrait une lecture plus simple d'une spécification.

CHAPITRE 5 CONCLUSION

5.1 Synthèse des travaux

Dans ce mémoire, nous avons examiné l'état de l'art des techniques de model-checking logiciel, en nous concentrant sur la vérification de programmes concurrents en C, suivant la norme POSIX (pThread). Nous décrivons les principales approches actuellement développées ainsi que les principaux projets actifs implémentant ces approches. Nous identifions deux principales limitations. L'explosion combinatoire reste, malgré les optimisations et les techniques de réduction, le principal obstacle à surmonter afin de permettre au model-checking logiciel de passer à l'échelle. De plus, les mécanismes de spécification dans le cas de programmes concurrents sont peu développés : les assertions sont peu adaptées à la concurrence, tandis que peu d'outils sont capables de vérifier des propriétés LTL, et toujours dans une forme limitée.

Nous nous sommes alors intéressés aux limitations des formalismes de spécifications. Nous avons mis en place un nouveau formalisme, basé sur LTL, et utilisant le concept de *zones de validité* afin de permettre de manipuler des positions du programme et des variables locales dans les propriétés LTL. Nous vérifions que ce formalisme permet de représenter à la fois la restriction de LTL classiquement utilisée ainsi que des assertions, tout en permettant de dépasser leurs principales limitations.

Enfin, nous présentons une transformation de source à source d'un programme en C permettant de vérifier un code spécifié par notre formalisme. Nous contournons le manque d'outils supportant le model-checking de propriétés LTL en reprenant l'approche présentée par [41] et nous l'adaptons à notre cas. Nous construisons ainsi un code instrumenté représentant le produit entre le code d'origine et un automate de Büchi, spécifié par des assertions. Ce code instrumenté peut ensuite être vérifié par un model-checker, sans nécessiter un support de LTL. Cette solution reste cependant limitée à l'exploration de traces finies. Nous avons testé cette instrumentation sur un jeu de tests, à l'aide des model-checkers CBMC et ESBMC, ce qui nous a permis de valider notre approche. Il est donc possible de vérifier des programmes spécifiés dans notre formalisme à l'aide de n'importe quel model-checker borné, d'autres techniques de model-checking étant probablement utilisables aussi au prix de modification de l'instrumentation ou d'une intégration de cette dernière directement dans le model-checker. Cependant, notre approche n'est pas encore capable de passer à l'échelle, car elle n'intègre pas de technique de réduction permettant de limiter l'explosion combinatoire due à la concurrence.

5.2 Améliorations futures

Le formalisme de spécification que nous proposons permet d'exprimer un ensemble de propriétés significativement plus large que celui exprimable par des assertions. Afin de mesurer l'impact de ce formalisme, des tests plus conséquents sur du code issus de systèmes réels seraient nécessaires. Une telle étude permettrait aussi de déterminer si des extensions plus larges sont encore nécessaires pour exprimer des propriétés fréquemment utilisées.

Une autre piste de travail intéressante serait d'intégrer la procédure de vérification de notre instrumentation directement dans un outil de model-checking. On perdrait alors les avantages d'une instrumentation, mais il serait possible de gagner en performances et de contourner un certain nombre de limitations actuelles de baProduct. Améliorer les performances pourrait aussi passer par l'intégration de techniques de réduction.

Enfin, il serait possible d'étudier comment tirer parti de l'instrumentation dans d'autres contextes de vérification. En particulier, l'instrumentation donne accès à l'automate de Büchi. En le modifiant, il est possible de restreindre les exécutions explorées par un model-checker et se concentrer sur des scénarios critiques. L'instrumentation pourrait aussi être modifiée afin d'être rendue exécutable, quitte à remplacer les choix non déterministes par des choix aléatoires. Cela ouvre la porte à des approches de vérification de notre spécification en utilisant des outils de test (stress testing ou fuzzing par exemple).

5.3 Remarques conclusives

À travers ce mémoire, nous nous sommes intéressés à étendre légèrement les possibilités de spécification pour le model-checking de programmes concurrents. Cependant, ces progrès viennent avec un coût de vérification plus important. Ces coûts de vérification, liés en grande partie à l'explosion combinatoire, constituent la principale limite du model-checking actuellement. Ce problème ne semble pas proche d'être résolu, à moins d'une percée spectaculaire dans ce domaine : on assiste plutôt à un ensemble d'optimisations qui permettent de vérifier des programmes de taille toujours plus grande. De manière concomitante, de nouveaux langages de programmation émergent, axés sur la sécurité et permettant de prouver certaines propriétés d'un programme à sa compilation. Un exemple est Rust[38], qui permet d'assurer l'absence de data-race et une certaine forme de consistance entre les threads. Le model-checking pourrait profiter des propriétés assurées par ces langages afin de réduire davantage la taille des modèles.

RÉFÉRENCES

[1] Property pattern mappings for LTL. En ligne : http://patterns.projects.cs.ksu.edu/documentation/patterns/ltl.shtml

[2] SARL : The symbolic algebra reasoning library. En ligne : http://vsl.cis.udel.edu/sarl/

[3] How SQLite is tested. En ligne : https://www.sqlite.org/testing.html

[4] JSON. En ligne : http://www.json.org/

[5] J. Barnat, L. Brim, et P. Rockai, "Towards LTL model checking of unmodified thread-based c c++ programs", dans *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7226 LNCS, pp. 252 – 266. En ligne : http://dx.doi.org/10.1007/978-3-642-28891-3_25

[6] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, et J. Weiser, "DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs", dans *Computer Aided Verification (CAV 2013)*, série LNCS, vol. 8044. Springer, 2013, pp. 863–868.

[7] G. Basler, A. Donaldson, A. Kaiser, D. Kroening, M. Tautschnig, et T. Wahl, "SatAbs : A bit-precise verifier for c programs (competition contribution)", dans *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7214 LNCS, pp. 552 – 555. En ligne : http://dx.doi.org/10.1007/978-3-642-28756-5_47

[8] I. Beer, S. Ben-David, et A. Landver, "On-the-fly model checking of RCTL formulas", dans *Proceedings of the 10th International Conference on Computer Aided Verification*, série CAV '98. Springer-Verlag, pp. 184–194. En ligne : http://dl.acm.org/citation.cfm?id=647767.733619

[9] D. Beyer, "Status report on software verification - (competition summary sv-comp 2014)", dans *In Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2014, pp. 373–388.

[10] ——, "Software verification and verifiable witnesses", dans *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of*

Systems-Volume 9035. Springer-Verlag New York, Inc., 2015, pp. 401–416.

[11] ——, "Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016)", dans *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2016, pp. 887–904.

[12] ——, "Software verification with validation of results", dans *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2017, pp. 331–349.

[13] D. Beyer et K. Friedberger, "A light-weight approach for verifying multi-threaded programs with cpachecker", *arXiv preprint arXiv :1612.04983*, 2016.

[14] D. Beyer et M. E. Keremoglu, "Cpachecker : A tool for configurable software verification", dans *International Conference on Computer Aided Verification*. Springer, 2011, pp. 184–190.

[15] D. Beyer, T. A. Henzinger, R. Jhala, et R. Majumdar, "The software model checker blast", *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5-6, pp. 505–525, 2007.

[16] R. E. Bryant, "Graph-based algorithms for boolean function manipulation", *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, Août 1986. DOI : 10.1109/TC.1986.1676819. En ligne : http://dx.doi.org/10.1109/TC.1986.1676819

[17] M. Carter, S. He, J. Whitaker, Z. Rakamarić, et M. Emmi, "Smack software verification toolchain", dans *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016, pp. 589–592.

[18] F. Cassez, A. M. Sloane, M. Roberts, M. Pigram, P. Suvanpong, et P. G. de Aledo, "Skink : Static analysis of programs in LLVM intermediate representation", dans *Tools and Algorithms for the Construction and Analysis of Systems*, A. Legay et T. Margaria, éds. Springer Berlin Heidelberg, vol. 10206, pp. 380–384, DOI : 10.1007/978-3-662-54580-5_27. En ligne : http://link.springer.com/10.1007/978-3-662-54580-5_27

[19] E. Clarke, D. Kroening, et F. Lerda, "A tool for checking ANSI-c programs", dans *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Berlin, Heidelberg, pp. 168–176, DOI : 10.1007/978-3-540-24730-2_15. En ligne : http://link.springer.com/chapter/10.1007/978-3-540-24730-2_15

[20] E. Clarke, D. Kroening, N. Sharygina, et K. Yorav, “Satabs : Sat-based predicate abstraction for ansi-c”.

[21] S. Conchon, J.-C. Filliâtre, et J. Signoles, “Designing a generic graph library using ml functors.” *Trends in functional programming*, vol. 8, pp. 124–140, 2007.

[22] B. Cook, D. Kroening, et N. Sharygina, “Over-approximating boolean programs with unbounded thread creation”, dans *2006 Formal Methods in Computer Aided Design*, pp. 53–59. DOI : 10.1109/FMCAD.2006.24

[23] L. Cordeiro et B. Fischer, “Verifying multi-threaded software using SMT-based context-bounded model checking”, dans *Proceedings - International Conference on Software Engineering*, pp. 331 – 340. En ligne : http://dx.doi.org/10.1145/1985793.1985839

[24] L. Cordeiro, J. Morse, D. Nicole, et B. Fischer, “Context-bounded model checking with ESBMC 1.17 (competition contribution)”, dans *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7214 LNCS, pp. 534 – 537. En ligne : http://dx.doi.org/10.1007/978-3-642-28756-5_42

[25] N. Dershowitz. Software horror stories. En ligne : https://www.cs.tau.ac.il/~nachumd/horror.html

[26] V. D’Silva, D. Kroening, et G. Weissenbacher, “A survey of automated techniques for formal software verification”, vol. 27, no. 7, pp. 1165–1178. DOI : 10.1109/TCAD.2008.923410

[27] B. Fischer, O. Inverso, et G. Parlato, “CSeq : A concurrency pre-processor for sequential c verification tools”, dans *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, pp. 710 – 713. En ligne : http://dx.doi.org/10.1109/ASE.2013.6693139

[28] P. Gastin et D. Oddoux, “Fast LTL to büchi automata translation”, dans *Computer Aided Verification*. Springer, Berlin, Heidelberg, pp. 53–65. DOI : 10.1007/3-540-44585-4_6. En ligne : https://link.springer.com/chapter/10.1007/3-540-44585-4_6

[29] S. Graf et H. Saidi, “Construction of abstract state graphs with PVS”, dans *Computer Aided Verification*, série Lecture Notes in Computer Science. Springer,

Berlin, Heidelberg, pp. 72–83. DOI : 10.1007/3-540-63166-6_10. En ligne : https://link.springer.com/chapter/10.1007/3-540-63166-6_10

[30] O. Grumberg et H. Veith, *25 years of model checking : history, achievements, perspectives*. Springer, 2008, vol. 5000.

[31] A. Gupta, C. Popeea, et A. Rybalchenko, "Predicate abstraction and refinement for verifying multi-threaded programs", dans *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, série POPL '11. ACM, pp. 331–344. DOI : 10.1145/1926385.1926424. En ligne : http://doi.acm.org/10.1145/1926385.1926424

[32] T. A. Henzinger, R. Jhala, R. Majumdar, et S. Qadeer, "Thread-modular abstraction refinement", dans *Computer Aided Verification*. Springer, Berlin, Heidelberg, pp. 262–274. DOI : 10.1007/978-3-540-45069-6_27. En ligne : https://link.springer.com/chapter/10.1007/978-3-540-45069-6_27

[33] G. J. Holzmann, "The model checker spin", *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[34] O. Inverso, E. Tomasco, B. Fischer, S. L. Torre, et G. Parlato, "Bounded model checking of multi-threaded c programs via lazy sequentialization", dans *Computer Aided Verification*, série Lecture Notes in Computer Science, A. Biere et R. Bloem, éds. Springer International Publishing, pp. 585–602, DOI : 10.1007/978-3-319-08867-9_39. En ligne : http://link.springer.com/chapter/10.1007/978-3-319-08867-9_39

[35] K. Jiang, *Model Checking C Programs by Translating C to Promela*. En ligne : http://www.diva-portal.org/smash/record.jsf?pid=diva2:235718

[36] R. Koymans, "Specifying real-time properties with metric temporal logic", *Real-time systems*, vol. 2, no. 4, pp. 255–299, 1990.

[37] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, et J. Vouillon, *The OCaml system (release 4.04) : Documentation and user's manual*, Institut National de Recherche en Informatique et en Automatique, Nov. 2016. En ligne : http://caml.inria.fr/distrib/ocaml-3.12/ocaml-3.12-refman.pdf

[38] N. D. Matsakis et F. S. Klock, II, "The rust language", *Ada Lett.*, vol. 34, no. 3, pp. 103–104, Oct. 2014. DOI : 10.1145/2692956.2663188. En ligne : http://doi.acm.org/10.1145/2692956.2663188

[39] K. L. McMillan, “Lazy abstraction with interpolants”, dans *Computer Aided Verification*. Springer, Berlin, Heidelberg, pp. 123–136, DOI : 10.1007/11817963_14. En ligne : http://link.springer.com/chapter/10.1007/11817963_14

[40] J. Morse, L. Cordeiro, D. Nicole, et B. Fischer, “Handling unbounded loops with ESBMC 1.20 : (competition contribution)”, dans *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7795 LNCS, pp. 619 – 622. En ligne : http://dx.doi.org/10.1007/978-3-642-36742-7_47

[41] ——, “Model checking ltl properties over ansi-c programs with bounded traces”, *Software & Systems Modeling*, vol. 14, no. 1, pp. 65–81, 02 2015. DOI : 10.1007/s10270-013-0366-0. En ligne : https://doi.org/10.1007/s10270-013-0366-0

[42] J. Mrazek, P. Bauch, H. Lauko, et J. Barnat, “SymDIVINE : Tool for control-explicit data-symbolic state space exploration”, dans *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9641, pp. 208 – 213. En ligne : http://dx.doi.org/10.1007/978-3-319-32582-8_14

[43] M. Musuvathi, S. Qadeer, et T. Ball, “CHESS : A systematic testing tool for concurrent software”. En ligne : https://www.microsoft.com/en-us/research/publication/chess-a-systematic-testing-tool-for-concurrent-software/

[44] G. C. Necula, S. McPeak, S. P. Rahul, et W. Weimer, *CIL : Intermediate Language and Tools for Analysis and Transformation of C Programs*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2002, pp. 213–228. DOI : 10.1007/3-540-45937-5_16. En ligne : https://doi.org/10.1007/3-540-45937-5_16

[45] T. L. Nguyen, B. Fischer, S. La Torre, et G. Parlato, “Lazy sequentialization for the safety verification of unbounded concurrent programs”, dans *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9938 LNCS, pp. 174 – 191. En ligne : http://dx.doi.org/10.1007/978-3-319-46520-3_12

[46] A. Pnueli, “The temporal logic of programs”, dans *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pp. 46–57. DOI : 10.1109/SFCS.1977.32

[47] C. Popeea et A. Rybalchenko, “Threader : A verifier for multi-threaded programs : (competition contribution)”, dans *Lecture Notes in Computer Science (including*

subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 7795 LNCS, pp. 633 – 636. En ligne : http://dx.doi.org/10.1007/978-3-642-36742-7_51

[48] J. Regehr. Use of assertions – embedded in academia. En ligne : https://blog.regehr.org/archives/1091

[49] E. Tomasco, O. Inverso, B. Fischer, S. L. Torre, et G. Parlato, "Verifying concurrent programs by memory unwinding", dans *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Berlin, Heidelberg, pp. 551–565. DOI : 10.1007/978-3-662-46681-0_52. En ligne : https://link.springer.com/chapter/10.1007/978-3-662-46681-0_52

[50] B. Wachter, D. Kroening, et J. Ouaknine, "Verifying multi-threaded software with impact", dans *2013 Formal Methods in Computer-Aided Design*, pp. 210–217. DOI : 10.1109/FMCAD.2013.6679412

[51] Y. Yang, X. Chen, G. Gopalakrishnan, et R. M. Kirby, "Runtime model checking of multithreaded c/c++ programs", 2007.

[52] A. Zaks et R. Joshi, "Verifying multi-threaded c programs with SPIN", dans *Model Checking Software*. Springer, Berlin, Heidelberg, pp. 325–342, DOI : 10.1007/978-3-540-85114-1_22. En ligne : http://link.springer.com/chapter/10.1007/978-3-540-85114-1_22

[53] M. Zheng, M. S. Rogers, Z. Luo, M. B. Dwyer, et S. F. Siegel, "CIVL : Formal verification of parallel programs", dans *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 830–835. DOI : 10.1109/ASE.2015.99

ANNEXE A Types de propriétés supportés par différents model-checkers

Tableau A.1 Types de propriétés prises en charge par les outils présentés.

Outils	Types de propriétés vérifiées								
	Assertions	LTL	Arithmétique	Tableaux	Pointeurs	Data races	Deadlock	Cast	Init
SPIN		✓							
Pancam	✓								
Inspect	✓								
Divine	✓	✓							
CIVL	✓		✓	✓	✓	✓	✓	✓	✓
Impara	✓								
Satabs	✓		✓	✓	✓				
Threader	✓								
CPAChecker	✓								
CBMC	✓		✓	✓	✓				
ESBMC	✓		✓	✓	✓	✓	✓		
CSeq	✓								
Lazy-CSeq	✓						✓		
Mu-CSeq	✓								
UL-CSeq	✓								

ANNEXE B Exemple de spécification

Dans cette annexe, nous présentons un exemple détaillé de spécification et de vérification d'un problème simple à l'aide de baProduct. Nous détaillons les étapes intermédiaires.

Considérons le problème suivant : un système contient deux batteries. Les batteries se vident simultanément au fil de l'utilisation. Lorsqu'une batterie est épuisée, elle est remplacée. Une batterie est suffisante pour que le système fonctionne, cependant, si les deux batteries sont vides en même temps, le système n'a plus d'énergie et cesse de fonctionner.

Un tel système peut être modélisé par le code ci-dessous. Deux threads, `battery1` et `battery2` contrôlent le rythme auquel les batteries se vident. Le pourcentage d'énergie dans la batterie est représenté par la variable `energy`. Celle-ci est décrémentée dans une boucle, ce qui représente la consommation de l'énergie, avant d'être remise à une valeur de 10, ce qui simule le remplacement. Dans ce système simpliste, aucune synchronisation n'existe entre les threads. On peut aussi noter que la boucle principale des threads ne s'exécute que trois fois, afin que la trace soit finie. Dans un cas d'utilisation réel, on utiliserait une boucle infinie et les options du model-checker permettraient de limiter le nombre de dépliages, mais pour des raisons de simplicité, nous codons directement ce comportement.

Listing B.1 Thread 1

```
void* battery1(void* d) {
  int energy;
  int i;
  // Initialization
  energy = 10;
  for(i=0; i<3; i++) {
    // Empty the battery
    while (energy > 0)
        energy--;
    // Replace the battery
    energy = 10;
  }
  pthread_exit(NULL);
}
```

Listing B.2 Thread 2

```
void* battery2(void* d) {
  int energy;
  int i;
  // Initialization
  energy = 10;
  for(i=0; i<3; i++) {
    // Empty the battery
    while (energy > 0)
        energy--;
    // Replace the battery
    energy = 10;
  }
  pthread_exit(NULL);
}
```

Pour s'assurer que le système fonctionne correctement, on souhaite s'assurer qu'une des deux

batteries a toujours un niveau d'énergie supérieur à 20%, une fois le système initialisé.

Nous allons tout d'abord spécifier cette propriété sur le système. Pour cela, on délimite tout d'abord les zones de validités. L'initialisation des batteries ne doit pas en faire partie, mais la boucle principale du programme y est incluse.

Listing B.3 Thread 1 avec labels

```
void* battery1(void* d) {
  int energy;
  int i;
  energy = 10;
b1: for(i=0; i<3; i++) {
    // Empty the battery
    while (energy > 0)
        energy--;
    // Replace the battery
    energy = 10;
  }
e1: pthread_exit(NULL);
}
```

Listing B.4 Thread 2 avec labels

```
void* battery2(void* d) {
  int energy;
  int i;
  energy = 10;
b2: for(i=0; i<3; i++) {
    // Empty the battery
    while (energy > 0)
        energy--;
    // Replace the battery
    energy = 10;
  }
e2: pthread_exit(NULL);
}
```

On définit ensuite les fonctions d'évaluation. Ici, nous allons utiliser deux propositions atomiques, chacune indiquant si une batterie a un niveau d'énergie faible. Cependant, une seule fonction d'évaluation est nécessaire puisque les deux propositions atomiques s'évaluent selon la même formule.

Listing B.5 Fonction d'évaluation

```
int low_power(int energy) {
  return energy <= 2;
}
```

Il ne reste alors qu'à exprimer la spécification en elle même.

Listing B.6 Spécification

```
{ "ltl": "G(! (p1 && p2))",
  "pa": [
    {"name": "p1",
      "default": false,
      "expr": "low_power",
      "span": ["b1", "e1"],
      "params": ["battery1::energy"]
    },
    { "name": "p2",
      "default": false,
      "expr": "low_power",
      "span": ["b2", "e2"],
      "params": ["battery2::energy"]
    }
  ]
}
```

On utilise alors baProduct pour instrumenter ce code. On utilise pour cela la commande `baProduct.py -i battery.c -s battery.spec`. Cette commande réalise tout d'abord un appel à LTL2BA afin de générer l'automate correspondant à la négation de la formule LTL (étape 1 dans la Figure 4.1). On obtient l'automate en Figure B.1.

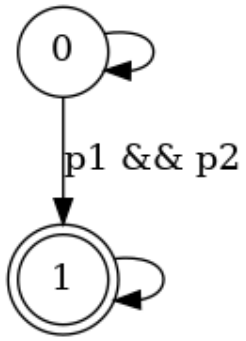

Figure B.1 Automate de Büchi généré par LTL2BA pour la spécification des batteries.

Cet automate est ensuite représenté en C. On obtient le code suivant (on suppose ici que CBMC va être utilisé comme backend).

Listing B.7 Automate encodé en C

```
// Current values of atomic propositions
int _ltl2ba_atomic_p1 = 0;
int _ltl2ba_atomic_p2 = 0;
// Is a proposition atomic in its validity area ?
int _ltl2ba_active_p1 = 0;
int _ltl2ba_active_p2 = 0;
// In a validity area, point to local variables
// used in evaluation functions
int *_ltl2ba_pointer_thread2_energy = 0;
int *_ltl2ba_pointer_thread1_energy = 0;
// Current state of the automaton
int _ltl2ba_state_var = 0;

// Transition function of the automaton
void _ltl2ba_transition() {
  int choice = nondet_uint(); // Choose a transition
  switch (_ltl2ba_state_var) {
  case 0: // Transitions from state 0
    if (choice == 0) {
      __CPROVER_assume(_ltl2ba_atomic_p1 && _ltl2ba_atomic_p2);
      _ltl2ba_state_var = 1;
    } else if (choice == 1) {
       __CPROVER_assume(1);
       _ltl2ba_state_var = 0;
    } else {
       __CPROVER_assume(0);
    }
    break;
  case 1: // State one is an accepting pit
    __CPROVER_assert(0, "ERROR_SURE");
    break;
  }
}

// Pre-calculated results for stutter acceptance
```

```
int _ltl2ba_surely_reject[2] = {0, 0};
int _ltl2ba_surely_accept[2] = {0, 1};
int _ltl2ba_stutter_accept[8] = {0, 1, 0, 1, 0, 1, 1, 1,};

int _ltl2ba_sym_to_id() {
  int id = 0;
  id |= (_ltl2ba_atomic_p2 << 1);
  id |= (_ltl2ba_atomic_p1 << 0);
  return id;
}
// Raise an assertion according to the final state
void _ltl2ba_result() {
  // The automaton accept the word, whatever the suffix
  int reject_sure = _ltl2ba_surely_accept[_ltl2ba_state_var];
  __CPROVER_assert(!reject_sure, "ERROR SURE");
  // The automaton accept the word in stutter-extension semantic
  int id = _ltl2ba_sym_to_id();
  int accept_stutter =_ltl2ba_stutter_accept[id * 2
                                             + _ltl2ba_state_var];
  __CPROVER_assert(!accept_stutter, "ERROR MAYBE");
  // The automaton does not surely reject the word
  int valid_sure = _ltl2ba_surely_reject[_ltl2ba_state_var];
  __CPROVER_assert(valid_sure, "VALID MAYBE");
}
```

Le code est ensuite instrumenté pour construire le produit avec l'automate (étape 2 dans la Figure 4.1).

Listing B.8 Code instrumenté

```
void *battery1(void *d )
{
  int energy ;
  int i;
  energy = 10;
  // Beginning of the validity area
  // Open an atomic bloc
  __CPROVER_atomic_begin();
```

```
  _ltl2ba_pointer_thread1_energy = &energy;
  _ltl2ba_atomic_p1 =
      low_power(*_ltl2ba_pointer_thread1_energy);
  _ltl2ba_active_p1 = 1;
  _ltl2ba_transition();
  __CPROVER_atomic_end();
  b1: for(i=0; i<3; i++) {
    while (energy > 0) {
      // Variable modification
      __CPROVER_atomic_begin();
      energy--;
      // Compute the new value of the atomic proposition
      _ltl2ba_atomic_p1 =
          low_power(*_ltl2ba_pointer_thread1_energy);
      // Take a transition in the automaton
      _ltl2ba_transition();
      __CPROVER_atomic_end();
    }
    __CPROVER_atomic_begin();
    energy = 10;
    _ltl2ba_atomic_p1 =
        low_power(*_ltl2ba_pointer_thread1_energy);
    _ltl2ba_transition();
    __CPROVER_atomic_end();
  }
  // End of the validity area
  __CPROVER_atomic_begin();
  // Set the atomic proposition to the default value
  _ltl2ba_atomic_p1 = 0;
  // Deactivate the atomic proposition
  _ltl2ba_active_p1 = 0;
  // Take a transition in the automaton
  _ltl2ba_transition();
  __CPROVER_atomic_end();
e1: pthread_exit(0);
}
```

```
void *battery2(void *d )
{
  int energy ;
  int i;
  energy = 10;
  __CPROVER_atomic_begin();
  _ltl2ba_pointer_thread2_energy = & energy;
  _ltl2ba_atomic_p2 = low_power(*_ltl2ba_pointer_thread2_energy);
  _ltl2ba_active_p2 = 1;
  _ltl2ba_transition();
  __CPROVER_atomic_end();
  b2: for(i=0; i<3; i++) {
    while (energy > 0) {
      __CPROVER_atomic_begin();
      energy--;
      _ltl2ba_atomic_p2 =
          low_power(*_ltl2ba_pointer_thread2_energy);
      _ltl2ba_transition();
      __CPROVER_atomic_end();
    }
    __CPROVER_atomic_begin();
    energy = 10;
    _ltl2ba_atomic_p2 =
        low_power(*_ltl2ba_pointer_thread2_energy);
    _ltl2ba_transition();
    __CPROVER_atomic_end();
  }
  __CPROVER_atomic_begin();
  _ltl2ba_atomic_p2 = 0;
  _ltl2ba_active_p2 = 0;
  _ltl2ba_transition();
  __CPROVER_atomic_end();
e2: pthread_exit(0);
}
int main(int argc , char **argv )
{
```

```
  pthread_t t1 ;
  pthread_t t2 ;
  pthread_create(&t1, 0, &battery1, 0);
  pthread_create(&t2, 0, &battery2, 0);
  pthread_join(t1, 0);
  pthread_join(t2, 0);
  __CPROVER_atomic_begin();
  _ltl2ba_result();
  __CPROVER_atomic_end();
  return 0;
}
```

Il ne reste plus qu'à vérifier ce code instrumenté à l'aide d'un model-checker. Ici, nous avons utilisé CBMC, à l'aide de la commande `cbmc battery_instr.c`. La sortie indique que l'assertion `ERROR_SURE` a été violée : une erreur est donc détectée de manière certaine dans le code.