Vérification distribuée à la volée de grands espaces d'états

Christophe Joubert

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Institut National de Recherche en Informatique et en Automatique
projet VASY

N° attribué par la bibliothèque

THÈSE

pour obtenir le grade de

Docteur de l'Institut National Polytechnique de Grenoble
spécialité Informatique : Systèmes et Logiciels

au titre de l'école doctorale de Mathématiques, Sciences et Technologies de l'Information, Informatique

présentée et soutenue publiquement le 12 décembre 2005 par

Christophe JOUBERT

Vérification distribuée à la volée de grands espaces d'états

Directeurs de thèse :	Hubert	GARAVEL	Directeur de Recherche à l'INRIA Rhône-Alpes
	Radu	MATEESCU	Chargé de Recherche à l'INRIA Rhône-Alpes
Après avis des rapporteurs :			
	Alessandro	FANTECHI	Professeur à l'Université de Florence
	François	VERNADAT	Professeur à l'INSA de Toulouse
Devant la commission d'examinateurs formée de :			
	Brigitte	PLATEAU	Professeur à l'INPG (Présidente du Jury)
	Yves	ROBERT	Professeur à l'ENS de Lyon
	Eric	MADELAINE	Chargé de Recherche à l'INRIA Sophia Antipolis

Résumé : La vérification des systèmes d'états finis, qu'ils soient distribués ou concurrents, est confrontée en pratique au problème d'explosion d'états (taille prohibitive de l'espace d'états sous-jacent) qui survient pour des systèmes de taille réaliste, contenant de nombreux processus en parallèle et des structures de données complexes. Différentes techniques ont été proposées pour combattre l'explosion d'états, telles que la vérification à la volée, la réduction d'ordre partiel, ou encore la vérification distribuée. Cependant, l'expérience pratique a montré qu'aucune de ces techniques, à elle seule, n'est toujours suffisante pour maîtriser des systèmes à grande échelle.

Cette thèse propose une combinaison de ces approches dans le but de faire passer à l'échelle leurs capacités de vérification. Notre approche est basée sur les Systèmes d'Equations Booléennes (SEBs), qui fournissent une représentation intermédiaire élégante des problèmes de vérification définis sur des Systèmes de Transitions Etiquetées, comme la comparaison par équivalence, la réduction par tau-confluence, l'évaluation de formules de mu-calcul modal sans alternance, ou encore la génération de cas de tests de conformité.

Nous proposons DSOLVE et MB-DSOLVE, deux nouveaux algorithmes pour la résolution distribuée à la volée de SEBs (contenant un, resp. plusieurs blocs d'équations), et nous les employons comme moteurs de calcul pour quatre outils de vérification à la volée développés au sein de la boîte à outils CADP en utilisant l'environnement OPEN/CÆSAR : le comparateur par équivalence BISIMULATOR, le réducteur TAU_CONFLUENCE, l'évaluateur de propriétés temporelles EVALUATOR, et le générateur de cas de tests de conformité EXTRACTOR. Les mesures expérimentales effectuées sur des grappes de machines montrent des accélérations quasi-linéaires et un bon passage à l'échelle des versions distribuées de ces outils par rapport à leurs équivalents séquentiels.

Mots-clés : système d'équations booléennes, algorithme à mémoire distribuée, vérification à la volée, comparaison par équivalence, réduction d'ordre partiel, évaluation de formules de logique temporelle, génération de cas de tests de conformité

Abstract : The verification of concurrent finite-state systems is confronted in practice with the state explosion problem (prohibitive size of the underlying state space), which occurs for realistic systems containing many parallel processes and complex data structures. Various techniques for fighting against state explosion have been proposed, such as on-the-fly verification, partial order reduction, and distributed verification. However, practical experience has shown that none of these techniques alone is always sufficient to handle large-scale systems.

In this thesis, we propose a combination of these approaches in order to scale up their capabilities. Our approach is based upon Boolean Equation Systems (BESs), which provide an elegant intermediate representation for verification problems defined on Labeled Transition Systems, such as equivalence checking, tau-confluence reduction, model checking of alternation-free mu-calculus, and test-case generation.

We propose DSOLVE and MB-DSOLVE, two new algorithms for distributed on-the-fly resolution of BESs (containing one, resp. several equation blocks), and employ them as computing engines for four on-the-fly verification tools developed within the CADP toolbox using the OPEN/CÆSAR environment : the BISIMULATOR equivalence checker, the TAU_CONFLUENCE reductor, the EVALUATOR model checker, and the EXTRACTOR test-case generator. Experimental measures performed on clusters of machines show quasi-linear speedups and a good scalability of the distributed versions of these tools w.r.t. their sequential counterparts.

Keywords : boolean equation systems, distributed memory algorithm, on-the-fly verification, equivalence checking, partial order reduction, model checking, test case generation

Remerciements

Je souhaite, par ces quelques lignes, exposer des remerciements sincères et toute ma reconnaissance aux personnes qui ont contribué directement ou non à ce travail de recherche, ou qui m'ont tout simplement accompagné durant ces trois années de doctorat.
En premier lieu, je remercie vivement tous les membres de mon jury de thèse :

- M^{me} Brigitte Plateau, Professeur à l'Inpg, pour m'avoir fait l'honneur de présider mon jury et pour l'examen de mon travail de recherche,
- M. Alessandro Fantechi, Professeur à l'Université de Florence (Italie), qui a accepté de juger cette thèse. Je tiens à le remercier pour son rapport, ses remarques pertinentes et sa synthèse de qualité concernant ce manuscrit,
- M. François Vernadat, Professeur à l'Insa de Toulouse, qui a également bien voulu évaluer mon travail. Je souhaite le remercier pour sa disponibilité et pour ses suggestions qui ont contribué à enrichir ce document,
- M. Yves Robert, Professeur à l'Ens de Lyon, qui a examiné ce travail. Je lui suis reconnaissant d'avoir donné son avis d'expert en parallélisme, et je suis très sensible aux encouragements qu'il m'a apporté,
- M. Eric Madelaine, Chargé de Recherche à l'Inria Sophia Antipolis, qui a également accepté de participer à ce jury. Je le remercie tout particulièrement pour le grand intérêt qu'il porte aux outils de vérification résultant de cette thèse, ainsi que pour ses précieux conseils,
- M. Hubert Garavel, Directeur de Recherche à l'Inria Rhône-Alpes, et responsable du projet Vasy, qui m'a accueilli avec la plus grande bienveillance au sein de son équipe. Je lui suis remerciant pour l'encadrement de ce travail de thèse, sa vision au long terme des problèmes de vérification et la qualité et pertinence de chacune de ses interventions.

Enfin, et non des moindres, je tiens à remercier très chaleureusement Radu Mateescu, Chargé de Recherche à l'Inria Rhône-Alpes, pour avoir assuré la direction de cette thèse, encadré parfaitement l'avancée de mes travaux, et intervenu avec une rapidité remarquable chaque fois que ces derniers le nécessitaient. Pour cela et bien plus encore, je te remercie Radu.

Une thèse est avant tout une expérience vécue. A ce titre, je remercie tout l'Inria Rhône-Alpes qui procure un environnement de travail dynamique, agréable et motivant. Mes journées de recherche au laboratoire ont également été dictées par le climat stable et productif de l'équipe Vasy. Ainsi, je souhaite remercier tous les membres présents et passés de l'équipe avec qui j'ai pu interagir, obtenir de l'aide, un conseil, ou tout simplement partager des moments sympathiques, notamment et sans ordre particulier Frédéric Lang, Nicolas Descoubes, David Champelovier, Gwen Salaün, Valérie Gardès, Damien Bergamini, Wendelin Serwe, Elodie Toihein, Jérôme Fereyre, Abdul-Malik Khan, Nathalie Lépy, Catherine Magnin, Solofo Ramangalahy, Bruno Ondet, Frédéric Tronel, Aurore Collomb, Gordon Pace, Alban Catry, Guillaume Schaeffer, Gilles Stragier.

Sans tous les énumérer, je souhaite aussi associer à ces remerciements mes amis avec qui j'ai vécu de grands moments ces trois dernières années.

Je tiens également à témoigner du soutien perpétuel de ma famille. Ils ont été le fer de lance de mes études puis travaux de recherche. Pour leur appui inébranlable et la confiance qu'ils m'ont apporté tout au long de ces années, je leur suis très reconnaissant.

Enfin, Amparo mérite une reconnaissance toute particulière pour avoir su être à mes côtés dans les moments les plus difficiles, pour m'avoir suivi malgré les obstacles, et conseillé en cas de doutes. Amparo, en me complétant, tu m'as rendu plus heureux chaque jour. Que ces quelques mots de grands remerciements te soient personnellement dédiés.

Table des matières

Introduction

Concevoir des systèmes informatiques (logiciels ou architectures matérielles) très larges, garantis pour fonctionner correctement, est encore un objectif non résolu malgré les dizaines d'années de recherche dans cette direction. Dus à l'omniprésence des systèmes informatiques dans notre société, les coûts engendrés par des systèmes défectueux deviennent alarmants.

Dans cette thèse, notre objectif est la détection des *fautes conceptuelles* (ou erreurs) dans la réalisation de systèmes. De telles fautes apparaissent souvent lors de circonstances imprévisibles, ou trop compliquées pour être découvertes et comprises par des êtres humains. Elles surviennent, par exemple, lorsqu'un programme est exécuté dans un environnement distribué. Tester exhaustivement de tels programmes est pratiquement impossible. La conception de ces systèmes nécessite des méthodes et outils de vérification, dont une catégorie largement reconnue est la *vérification formelle*.

Deux grandes approches existent dans le domaine de la vérification formelle : l'approche basée sur la *preuve de théorèmes* et l'approche basée sur les *modèles*. La première permet de traiter des systèmes ayant un nombre infini d'états, mais ne peut pas être complètement automatisée. En revanche, la méthode basée sur les modèles, bien que restreinte à des systèmes ayant un nombre fini d'états, permet un processus de vérification simple et plus rapide, qui ne nécessite pas un haut niveau d'expertise de la part de l'utilisateur.

L'approche que nous adoptons est celle orientée modèle, très connue dans la plupart des disciplines d'ingénierie. Au lieu de construire un objet directement, nous en concevons d'abord un modèle, lequel est analysé et vérifié entièrement. Une fois que le modèle obtenu et testé est satisfaisant, l'objet peut vraiment être construit. En ce qui concerne les systèmes informatiques très larges, le modèle est une abstraction (ou simplification) du comportement de parties critiques du système, et la vérification consiste à prouver que le modèle satisfait toutes les propriétés désirées. Les recherches en méthodes formelles ont permis de définir un ensemble complet de notations pour décrire approximativement tous les modèles que nous pouvons concevoir, et pour spécifier toutes les propriétés intéressantes, incluant les propriétés comportementales et temporelles. La quête essentielle que nous devons maintenant remplir est de prouver que le système satisfait les propriétés requises.

Vérification de programmes distribués et distribution des calculs

Le mécanisme de vérification. La *vérification* est un processus utilisé pour démontrer la correction fonctionnelle d'une conception logicielle, généralement décrite au moyen d'un langage de programmation ou mieux, d'un langage de haut niveau (ou formel) ayant une sémantique opérationnelle bien définie, comme les algèbres de processus ELOTOS[1] [ISO01], μ-CRL2[2] [Gro05], etc. Dans l'ap-

[1] Enhancements to the Language Of Temporal Ordering Specification
[2] *micro* Common Representation Language 2

proche basée sur les modèles, très largement utilisée car caractérisée par un bon compromis coût-performance, cette description est ensuite traduite automatiquement vers un modèle sous-jacent, qui est souvent un *système de transitions étiquetées* (STE), c'est-à-dire un graphe (ou automate) contenant, éventuellement avec certaines abstractions, tous les comportements possibles du système logiciel. La vérification consiste alors à comparer le modèle du système avec sa *spécification* ou les *propriétés* qui décrivent le fonctionnement attendu du système (ou encore les services que celui-ci doit fournir), comme par exemple, l'absence de blocage, l'exclusion mutuelle, l'équité, etc.

Les propriétés vérifiées. Selon la nature de la spécification ou le formalisme utilisé pour exprimer les propriétés, on peut distinguer deux types d'approches pour la vérification :

- *Propriétés comportementales* : elles décrivent le fonctionnement du système sous forme d'automates (ou bien en utilisant des descriptions de plus haut niveau que l'on traduit ensuite en automates). Etant donné que le système à vérifier et ses propriétés comportementales peuvent tous deux être représentés par des automates, la vérification consiste à les comparer au moyen de *relations d'équivalence* ou de *préordre*.
- *Propriétés logiques* : elles caractérisent des propriétés fonctionnelles du système, telles que l'absence de blocage, l'exclusion mutuelle, la transmission correcte des messages ou l'accès équitable aux ressources. Parmi les formalismes utilisés, les *logiques temporelles*, en particulier le *μ-calcul modal*, s'avèrent bien adaptés pour décrire l'évolution du système dans le temps. Dans ce cas, la vérification consiste à s'assurer que l'automate modélisant le système à vérifier satisfait un ensemble de propriétés logiques à l'aide d'outils appelés *évaluateurs*.

L'explosion d'états. Bien que les techniques de vérification basée sur les modèles soient efficaces et complètement automatisables, leur principale limitation réside dans le problème de l'*explosion d'états*, qui survient lorsque le nombre d'états du système à vérifier dépasse les capacités en mémoire de la machine. L'explosion d'états est une croissance combinatoire de l'espace d'états d'un système logiciel lorsqu'il est composé de plusieurs composants parallèles asynchrones (entrelacement des actions lors de l'exécution) et lorsqu'il manipule des données complexes (un état étant associé à chaque combinaison de valeurs possibles de ces données). L'explosion de l'espace d'états est la raison pour laquelle la taille des systèmes que les outils de vérification peuvent actuellement maîtriser est généralement petite, et pour laquelle de nombreuses applications intéressantes restent encore hors de portée. Plusieurs solutions partielles à ce problème existent, telles que la combinaison des techniques de vérification avec des techniques de manipulation des graphes. Parmi les plus représentatives, nous pouvons mentionner :

- La vérification *à la volée* est une technique de manipulation des graphes sous forme *implicite* (c'est-à-dire sous la forme d'un état initial et d'une fonction successeur), généralement pour les modèles de grande taille (supérieure à 10^8 états), en explorant dynamiquement les parties du graphe en fonction des besoins. Contrairement à la vérification *globale*, qui manipule les graphes sous forme *explicite* (c'est-à-dire sous la forme de l'ensemble des états, d'une fonction successeur et d'une fonction prédécesseur), elle ne nécessite par la sauvegarde en mémoire de l'ensemble des états et des transitions pour procéder à la vérification du système. Cette méthode, simple et efficace, s'avère particulièrement utile dans les premières phases du processus de conception, quand les erreurs sont susceptibles d'être plus fréquentes.
- La *réduction* d'espace d'états consiste à réduire la taille de l'espace d'états exploré, afin de transformer le problème de vérification du système global en un problème équivalent sur un plus petit espace d'états. Une de ces techniques, connue sous le nom de *réduction par ordre partiel*, élimine les entrelacements d'actions superflus produits par des transitions causalement indépendantes. Une autre technique, dite de *réduction par symétries*, exploite la répétition

de structures similaires souvent rencontrées dans les systèmes répartis, comme par exemple la présence de plusieurs processus parallèles identiques qui ne diffèrent que par les valeurs de leurs paramètres, pour réduire l'espace d'états du système global.

- La vérification *compositionnelle* permet de définir un système en terme de composants parallèles. La taille de l'espace d'états de la composition parallèle ainsi obtenue est alors bornée à seulement le produit des états des composants. Cette technique est généralement combinée avec une minimisation de chacun des composants modulo une relation d'équivalence qui préserve les propriétés à vérifier, avant de construire, parcourir ou analyser l'espace d'états global. Cette technique est aussi appelée *réduction par minimisation compositionnelle*.
- La vérification *distribuée* utilise un ensemble de machines ou processeurs (également appelés *nœuds*) connectés par un réseau, pour augmenter les ressources de calcul et d'espace mémoire disponibles par un ou deux ordres de grandeur, et permettre la vérification de systèmes trop larges pour des machines séquentielles.

Cependant, l'expérience pratique a montré qu'aucune de ces techniques prises séparément ne peuvent suffire pour éviter l'explosion d'états, et leur utilisation conjointe a de grandes chances d'être plus efficace.

La vérification distribuée. Les dernières années ont été riches en résultats concernant le calcul scientifique de haute performance, qui n'a cessé d'accroître son importance pour le support d'explorations scientifiques et d'ingénierie multi-disciplinaire. Etant donné que les communautés du calcul scientifique et des méthodes formelles n'ont traditionnellement pas travaillé ensemble, il existe très peu de travaux décrivant les interactions possibles entre les différents résultats théoriques et pratiques. Un exemple de telle interaction est l'utilisation de méthodes formelles pour vérifier des systèmes logiciels et matériels pour le calcul de haute performance [PBY+05]. Notre étude adresse le problème inverse, à savoir, l'exploitation de la puissance de calcul et du parallélisme potentiels d'une pile de PCs pour l'exécution d'applications de vérification.

L'importance scientifique de la vérification distribuée repose sur le *passage à l'échelle* (ou extensibilité) de la vérification basée sur les modèles comme un outil d'analyse de systèmes logiciels très larges. En termes d'états explicites, la barrière du 1 Giga (10^9) états a été dépassée récemment et l'objectif avec les nouvelles méthodes distribuées est de pouvoir manipuler des espaces contenant 1 (10^{12}) états. En rendant le processus de vérification plus performant, faisable et applicable dans un contexte réel, un objectif de ce travail de thèse est également de rendre l'application de la vérification basée sur les modèles plus populaire et largement diffusée. Dans ce but, nous cherchons à simplifier autant que possible le développement des outils de vérification — qui sont des composants complexes de logiciel — en promouvant des technologies génériques, applicables à plusieurs problèmes de vérification, et des architectures modulaires de logiciel permettant une réutilisation maximale des algorithmes existants. Notre méthodologie est basée sur la résolution de *systèmes d'équations booléennes* (SEBs) d'alternance 1 [Mad97], un cadre bien établi utilisé pour la représentation des problèmes de vérification, tels que l'évaluation de formules de μ-calcul modal sur des STEs [And94, MS03], la comparaison par équivalence entre STEs modulo des relations d'équivalence variées [AV95, Mat03a], et la réduction par τ-confluence, une forme de réduction d'ordre partiel sur les STEs qui préserve l'équivalence de branchement [GP00, PLM03].

L'application d'algorithmes distribués de résolution de SEBs comme moteur de vérification pour ces problèmes permet un gain immédiat, linéaire au nombre de machines interconnectées utilisées pour le calcul, tout en facilitant, par sa modularité, la combinaison possible avec des techniques classiques de vérification visant à lutter contre l'explosion d'états.

Contexte de la thèse

Notre travail de thèse vise à concevoir et construire une infrastructure générique pour la vérification distribuée à la volée basée sur les modèles permettant la vérification fonctionnelle de systèmes concurrents complexes. Ce travail s'inscrit dans le contexte de la boîte à outils CADP (*Construction and Analysis of Distributed Processes*) [GLM02] pour la conception de protocoles de communication et de systèmes distribués. Dans la mesure du possible, nous essayons de valider nos propositions par le développement d'outils, en utilisant l'environnement générique OPEN/CÆSAR [Gar98] pour l'exploration à la volée de STEs, afin de les combiner avec d'autres outils existants de CADP et de permettre un interfaçage direct avec divers langages de spécification décrivant le programme source à vérifier. L'application de ces outils à des études de cas complexes, souvent industrielles, fait entièrement partie du processus de validation des méthodes proposées. Cette confrontation systématique avec les problèmes d'implémentation et d'expérimentation est un aspect essentiel de notre démarche.

L'approche que nous avons adoptée vise à remédier aux limitations techniques classiques de vérification concernant le passage à l'échelle sur la taille des programmes à vérifier. Plus précisément, nous avons poursuivi un double objectif :

- Concevoir des algorithmes distribués de résolution de SEBs contenant un ou plusieurs blocs d'équations booléennes (sans dépendances interblocs mutuellement récursives), afin d'augmenter les capacités de la résolution de SEBs (actuellement limitée à environ 10^8 variables sur les machines séquentielles) en utilisant les ressources de calcul et de mémoire d'un ensemble de machines interconnectées par un réseau. L'implémentation de ces algorithmes doit disposer d'interfaces de programmation génériques et utilisables par les techniques classiques de vérification et par l'ensemble des outils disposant d'une connexion à l'environnement OPEN/CÆSAR.
- Concevoir et développer des applications de vérification basée sur les modèles, traduisibles en termes de SEBs et bénéficiant directement des mécanismes de résolution distribuée proposés dans le premier objectif. Ces outils de vérification doivent être, autant que possible, indépendants du langage de description du programme à vérifier, tout en ayant des performances comparables aux meilleures outils existants.

Pour résoudre à la volée de manière distribuée des SEBs contenant un ou plusieurs blocs, nous avons conçu deux algorithmes distribués, appelés DSOLVE (*Distributed SOLVEr*) et MB-DSOLVE (*Multi-Block Distributed SOLVEr*), constitués principalement des éléments suivants.

Un modèle d'architecture à mémoire distribuée. Les performances des algorithmes distribués dépendent étroitement de l'architecture sur laquelle ils sont exécutés. Les architectures distribuées visées ici ne disposent pas d'une mémoire partagée et sont très génériques, car composées essentiellement de machines interconnectées par un réseau standard tel que ceux utilisés habituellement dans les entreprises et laboratoires académiques. Ce type d'architecture est plus couramment appelée une *grappe* de stations de travail (par exemple une grappe de PCs), dont la taille peut varier d'une échelle locale à une échelle nationale voire internationale avec les interconnexions de grappes, appelées *grilles*. Cette architecture implique généralement un mécanisme de distribution des tâches et des données par *passage de messages* qui nécessite un système de gestion efficace des communications engendrées par le calcul entre les nœuds.

Un modèle de programmation de type programme unique et données multiples (SPMD). Chaque nœud exécute une instance de l'algorithme distribué qui décrit l'ensemble des tâches à exécuter (principe du modèle SPMD). Les données sont fournies par la construction locale et progressive du

SEB et par la réception de variables booléennes envoyées par les nœuds distants. Les tâches sont principalement composées de deux parcours entrelacés, l'un d'exploration en avant (en profondeur d'abord (DFS) au niveau des blocs, et en largeur d'abord (BFS) au niveau des variables booléennes) du SEB à partir d'une variable d'intérêt, et l'autre, de propagation des valeurs booléennes stables le long des dépendances entre variables (principe des substitutions dans les équations booléennes). De plus, un processus *superviseur*, dont le comportement associé est également intégré aux algorithmes distribués, est ajouté aux processus de calcul distribué pour augmenter l'applicabilité de la solution et l'interaction avec l'utilisateur final, en permettant la gestion d'arrêts d'urgence du calcul, la génération de statistiques graphiques et textuelles, la phase d'initialisation du calcul distribué avec les copies et lancement des processus sur les nœuds distants, et la terminaison du calcul.

Une communication non-bloquante asynchrone. Afin de permettre un recouvrement maximal de la communication par les calculs, les opérations de communication sont rendues *non-bloquantes*, c'est-à-dire ne bloquant pas l'opération d'émission ou de réception jusqu'à sa terminaison, et *asynchrones*, c'est-à-dire ne dépendant pas de la synchronisation avec une opération de communication distante, comme une réception lors d'une tentative d'émission. La communication est ainsi rendue concurrente à d'autres opérations du calcul distribué global et local. Le point critique de la vérification basée sur les modèles étant la consommation mémoire, la couche de communication sur laquelle reposent les algorithmes distribués est basée sur la notion de tampons de communication bornés, qui sont gérés explicitement au niveau algorithmique, et permettent ainsi un contrôle fin de l'utilisation mémoire.

Une complexité linéaire en taille du problème. Les modèles SEBs traités étant très grands, et afin d'assurer la faisabilité de leur résolution, les algorithmes distribués ont une complexité linéaire en temps de calcul et en mémoire par rapport à la taille du problème à résoudre. De plus, et afin de limiter la proportion de messages échangés lors du calcul distribué sur le réseau et limiter ainsi le surcoût en temps induit par le passage de messages, la complexité en nombre de messages transmis est également linéaire en la taille du problème à résoudre.

Une génération de diagnostic (exemple ou contre-exemple) distribué. Lors du processus de vérification d'un système logiciel, la non-satisfaisabilité d'une propriété sur ce système conduit au besoin de décrire un contre-exemple illustrant cette situation afin de comprendre la source de l'erreur et de corriger le logiciel en conséquence. Ce mécanisme reste valable dans le cas symétrique où il est désirable d'avoir un exemple justifiant que la propriété a été vérifiée. Les algorithmes distribués proposés permettent de fournir un tel diagnostic sous forme d'une représentation implicite distribuée (c'est-à-dire une fonction successeur définie localement au nœud responsable de la variable courante du diagnostic).

Une détection distribuée de terminaison fine par bloc. Les algorithmes distribués proposés contiennent des mécanismes de détection de terminaison partielle de la résolution pour un bloc donné appartenant au SEB en cours de résolution. Cette approche permet d'obtenir un parallélisme à grain fin de la résolution en autorisant les parcours distribués de portions minimales et suffisantes d'un bloc parallèlement à la résolution d'autres blocs. La détection sur l'ensemble des nœuds de l'inactivité de la résolution locale à un bloc permet ainsi d'accélérer la résolution globale du SEB en augmentant le nombre de blocs explorés, et par suite la probabilité de trouver plus rapidement un bloc *terminal*, ne dépendant pas d'autres blocs, à partir duquel les valeurs stables seront propagées.

Après avoir défini formellement l'algorithme distribué DSOLVE (et par suite MB-DSOLVE), nous avons traité plusieurs problèmes de vérification, basée sur les modèles, traduisibles en termes de SEBs. Il

existe à l'heure actuelle très peu d'algorithmes pour la vérification distribuée de modèles, et encore moins d'approches combinant la manipulation implicite des graphes et des techniques de réduction. Du fait de l'absence de modularité des solutions déjà proposées et de leurs dépendances à des formats précis de description de propriétés ou de modèles, aucun de ces algorithmes ne peut être directement intégré dans notre contexte de résolution de SEBs. C'est pourquoi, nous avons adopté l'architecture modulaire proposé en [Mat03a], qui repose sur la bibliothèque générique CÆSAR_SOLVE intégrée à OPEN/CÆSAR, dédiée à la résolution séquentielle de SEBs. Chaque outil de vérification développé selon cette architecture comporte une *partie avant*, responsable de la traduction du problème correspondant en une résolution de SEB (et éventuellement de l'interprétation du diagnostic de cette résolution) et une *partie arrière*, responsable de la résolution à la volée du SEB et de la génération du diagnostic, assurée par la bibliothèque CÆSAR_SOLVE. En développant les algorithmes DSOLVE et MB-DSOLVE de manière compatible avec l'interface définie par CÆSAR_SOLVE, nous avons obtenu leur connexion directe avec les parties avant des outils de vérification séquentiels de CADP qui utilisent CÆSAR_SOLVE. Nous avons poursuivi nos travaux en définissant la traduction d'autres problèmes, tels que la génération de tests, en termes de SEBs et en développant les outils associés selon la même architecture modulaire.

Le fonctionnement de notre infrastructure générique pour la vérification distribuée à la volée basée sur les modèles est illustré dans la figure 1.

L'architecture modulaire de quatre outils de vérification y est détaillée ainsi que leur connexion aux algorithmes DSOLVE et MB-DSOLVE. Ces outils effectuent : la comparaison par équivalence (BISIMULATOR), la réduction par τ-confluence (TAU_CONFLUENCE), l'évaluation de formules de logiques temporelles (EVALUATOR) et la génération de cas de tests de conformité (EXTRACTOR utilisé conjointement avec DETERMINATOR). Tous prennent en entrée un programme à vérifier décrit implicitement par une fonction successeur donnée par l'environnement OPEN/CÆSAR, et une description de l'architecture distribuée (contenant notamment, la configuration des P nœuds de calcul) sous format GCF (*Grid Configuration File*). Certains nécessitent une entrée supplémentaire, telles qu'une propriété logique en μ-calcul modal (MCL), une spécification ou encore un objectif de tests sous format BCG (*Binary Coded Graph*). Les résultats de la vérification sont finalement produits en sortie de l'instance exécutée sur la machine locale.

Plan du document

Le chapitre 1 contient une réflexion critique sur l'état de l'art dans le domaine de la vérification distribuée basée sur les modèles. Différentes techniques de vérification (analyse d'accessibilité, comparaison par équivalence, réduction par équivalence, évaluation de formules logiques), et les versions distribuées qui leur sont associées, sont comparées selon leurs modèles de calcul distribué. Cette discussion nous permet de discerner à la fois les limites actuelles des méthodes de vérification distribuée mais aussi les principes de conception d'algorithmes distribués efficaces pour les problèmes de vérification, et leur implémentation concrète.

Le chapitre 2 introduit la formalisation du système d'équations booléennes (SEB), sa représentation sous forme de graphe booléen, ainsi que le mécanisme séquentiel de résolution de SEB. Il présente ensuite notre algorithme de résolution distribuée à la volée de SEBs monoblocs, appelé DSOLVE. Son implémentation et quelques résultats de performance concernant la résolution de SEBs monoblocs générés aléatoirement en utilisant une grappe de machines sont présentés également.

Figure 1: Les algorithmes DSOLVE et MB-DSOLVE et les outils de vérification distribuée à la volée basée sur les modèles

Le chapitre 3 est consacré à la première application de l'algorithme DSOLVE dans le domaine de la vérification : la comparaison à la volée par équivalence de deux systèmes de transitions étiquetées (STEs) sur un ensemble de machines homogènes de type grappes de PCs. Le potentiel de notre méthode sur des exemples concrets de taille réaliste extraits d'études de cas industrielles est confirmé pour cinq relations d'équivalence, et démontre la généricité de notre approche pour différents types de problèmes.

Le chapitre 4 définit une méthode pour la génération à la volée distribuée d'un espace d'états réduit qui est branchement bisimilaire avec l'original. Cette méthode est basée sur une application de l'algorithme DSOLVE à la détection de transitions τ-confluentes lors de la réduction d'ordre partiel par τ-confluence d'un STE. Cette application à un problème de vérification basée sur les modèles totalement différent de celui de la comparaison par équivalence confirme la généricité de notre approche, à la fois en architecture logicielle mais également en distribution des calculs de vérification qui, indépendamment du problème, exploitent efficacement plusieurs machines connectées par un réseau.

Le chapitre 5 contient une extension conservatrice de l'algorithme monobloc DSOLVE, appelée MB-DSOLVE. Les couches de complexité ajoutées à DSOLVE permettent à l'algorithme final d'être plus général et de résoudre à la volée de manière distribuée des SEBs d'alternance 1 contenant un ou plusieurs blocs. L'approche est validée dans un premier temps par l'implémentation de l'algorithme correspondant et ensuite par son application aux problèmes de comparaison par équivalences et de réduction d'ordre partiel sur lesquels la conservation des bonnes performances de DSOLVE est confirmée.

Le chapitre 6 se propose d'appliquer l'algorithme MB-DSOLVE au problème d'évaluation de formules temporelles du μ-calcul modal d'alternance 1 sur des STEs afin de distribuer ce problème de vérification locale sur un ensemble de machines interconnectées. Une comparaison fine avec des outils séquentiels existants optimisés ainsi que distribués mais utilisant une technologie différente, a permis de confirmer le bon fonctionnement de notre algorithme MB-DSOLVE, et a mis en évidence les avantages de notre approche basée sur les SEBs pour la détection très rapide de contre-exemples, obtenus lorsqu'une propriété n'est pas satisfaite par le système en cours de vérification.

Le chapitre 7 décrit une deuxième application de l'algorithme MB-DSOLVE à un problème de vérification basée sur les modèles : la génération à la volée de cas de tests de conformité pour une implémentation sous test (IUT) et un objectif de test donnés sous forme de STEs. En suivant l'approche modulaire structurant les diverses applications développées précédemment, un générateur (séquentiel et distribué) de tests a été conçu et développé, et se compose d'une partie avant traduisant la détection à la volée du graphe complet de tests (CTG) en termes de SEBs multiblocs (2 blocs dans le cas d'algorithmes de résolution séquentielle, et 3 blocs dans le cas de MB-DSOLVE) et générant le CTG à partir du diagnostic donné par les algorithmes de résolution, et d'une partie arrière utilisant un moteur de résolution de SEBs soit séquentiel (CÆSAR_SOLVE), soit distribué (MB-DSOLVE). La validation de l'approche est établie par la comparaison détaillée avec l'outil séquentiel TGV, qui effectue la génération de cas de tests de conformité.

Chapitre 1

Vérification distribuée

Sans aucun doute, la vérification distribuée, qu'elle soit basée sur les modèles explicites ou symboliques, est un champ de recherche et d'application en pleine expansion et promis à un avenir riche en avancées scientifiques. Un indicateur clair en est la création, depuis 2002, du colloque international sur les *Méthodes Parallèles et Distribuées pour la vérifiCation* (PDMC)[1]. Des numéros spéciaux de journaux internationaux comme *Outils Logiciels pour le Transfert Technologique* (STTT) en 2005 et *Méthodes Formelles dans la Conception des Systèmes* (FMSD) en 2006 ont été (ou le seront prochainement) publiés. Une dizaine de groupes internationaux de recherche experts en méthodes formelles, et plus particulièrement en vérification basée sur les modèles, se sont récemment intéressés à cette nouvelle thématique. Une multitude de projets très prometteurs sont en cours de réalisation, et vont de la vérification distribuée sur grilles de calcul, en passant par les problèmes d'équilibrage de charge dynamique ou d'utilisation efficace d'environnements hétérogènes, à la vérification de systèmes temps-réel, et de propriétés complexes portant sur les comportements, le temps, les valeurs et la stochasticité.

—— ⋆ —— ⋆ —— ⋆ ——

Ce chapitre contient une synthèse des différentes approches de vérification parallèle et distribuée rencontrées durant ces dix dernières années et positionne notre travail de thèse par rapport à l'ensemble des travaux réalisés en introduisant les concepts indispensables pour la suite de ce document.

Nous commençons, dans la section 1.1, par présenter les différents problèmes de vérification formelle qui ont été parallélisés ou distribués ainsi que les techniques sur lesquelles ils sont basés. La section 1.2 introduit ensuite, les concepts de base en architecture, programmation et algorithmes distribués que nous avons adopté pour la répartition des problèmes de vérification sur un ensemble de machines interconnectées, ainsi que divers critères d'évaluation (complexité, performance, généricité) utiles lors de la conception de solutions distribuées. Enfin, la section 1.3 définit le contexte de validation de notre approche basé sur des systèmes à vérifier, eux-mêmes distribués (tels que des programmes parallèles asynchrones), et représentés formellement par des modèles finis, qui serviront comme donnée d'entrée à notre processus de vérification distribuée.

[1] `http ://pdmc.informatik.tu-muenchen.de/`

1.1 Approches pour la vérification parallèle et distribuée

Lors de ces dernières années, la vérification distribuée de programmes a reçu beaucoup d'intérêt. De nombreux outils de vérification ont été développés, allant de la génération distribuée d'*espace d'états* [GMS01, Jou02, GMB+06, GMB+05, Cia01, HBB99] à la *vérification symbolique* distribuée [BDHGS00]. Des outils comme CADP [GLM02], SPIN [LS99] ou encore MURϕ [SD97] ont été adaptés pour être exécutés sur des machines distribuées. Différentes approches ont été considérées : à base de disques distribués [KH99], avec plusieurs flots d'exécution (multithreadée) [AKH97], avec compactage des états [MCC97], avec équilibrage dynamique de charge [ADK97], et à base de mémoire distribuée ou partagée [CGN98].

Cependant, la vérification distribuée est intrinsèquement un problème difficile (voir à ce sujet [IB02]). Elle requiert une double expertise à la fois en vérification formelle et en calcul distribué. La vérification, de son côté, peut être divisée en plusieurs étapes, parmi lesquelles, la création d'un modèle abstrait du système à vérifier, la vérification de propriétés sur le modèle, et la génération d'un contre-exemple, si les propriétés ne sont pas satisfaites. La distribution de la vérification consiste à traiter essentiellement trois points supplémentaires : le partage des données et des tâches, la minimisation du temps global d'inactivité, et la détection de terminaison. Les algorithmes distribués résultant de cet entrelacement d'étapes sont complexes et difficiles à concevoir.

De plus, aucun effort n'a été fourni jusqu'à présent pour permettre la réutilisation de techniques distribuées dans le cadre de différents problèmes de vérification. Ce n'est que très récemment (voir par exemple [HLL04]), qu'un souci d'architecture logicielle de vérification bien définie est apparu, afin de faciliter le développement d'outils de vérification distribuée futurs. Nous proposons dans cette étude d'aller encore plus loin dans la généricité des méthodes et outils pour la vérification distribuée en utilisant une représentation théorique unifiante des problèmes de vérification. L'approche que nous adoptons, et qui a fait ses preuves durant ces dernières années, est de résoudre le problème de vérification en le réduisant à la résolution de *systèmes d'équations booléennes* (SEBs). Dans le cas, par exemple, de l'évaluation de formules de μ-calcul, la formule à vérifier est dépliée sur le *système de transitions étiquetées* (STE) correspondant au système à vérifier. Cela résulte en un système ordonné d'équations booléennes, chacune marquée par un plus petit ou plus grand point fixe. Plusieurs algorithmes pour résoudre de tels SEBs existent [Mad97, MS03, Mat03a, GK04].

Nous souhaitons définir dans ce travail de thèse des algorithmes distribués de résolution de SEB, permettant ainsi la distribution directe de nombreux problèmes de vérification.

Dans la suite de cette section, nous présentons les diverses applications de vérification qui ont déjà été distribuées ainsi que les directions prises par les travaux actuels et les applications que nous souhaitons distribuer avec notre approche.

1.1.1 Analyse d'accessibilité

Traditionnellement, la vérification distribuée s'est d'abord focalisée sur le problème de la génération distribuée d'espace d'états, du fait de sa simplicité. Un état de l'art concernant la distribution du problème de génération d'espaces d'états est présent dans [Cia01]. Plusieurs méthodes de génération distribuée d'espace d'états ont été proposées et utilisent de la mémoire distribuée ou partagée pour obtenir des gains en performance. Les premiers algorithmes remontent à Caselli et al. [CCBF94, CCM95] et Ciardo [NC97] qui les ont expérimentés sur des architectures à mémoire partagée de type supercalculateurs. Depuis, les architectures parallèles évoluant, de nombreuses améliorations et optimisations ont été proposées.

Dans [GMS01, Jou02, GMB+06, GMB+05], de larges espaces d'états sont générés sur une grappe de

stations de travail. Les performances en temps d'exécution sont grandement améliorées par rapport aux outils séquentiels similaires, et la charge est répartie de manière équilibrée. La taille des espaces d'états qui peuvent être générés peut augmenter d'un facteur linéaire en nombre de nœuds utilisés. Par contre, des outils restent à être développés pour permettre l'utilisation directe de l'espace d'états distribué par les outils de vérification existants, évitant ainsi la fusion des fragments de l'espace d'états distribués en un espace d'états explicite stocké sur une seule machine. Une telle amélioration a été apportée par [BvLL03], où l'espace d'états est généré dans un format distribué directement utilisable par des outils de réduction [Orz04].

D'autres travaux ont porté sur des problèmes particuliers de la génération distribuée d'espace d'états, comme la représentation d'état [HBB99, MCC97, CCM01], la distribution des états [AH97], les politiques d'équilibrage de charge [CGN98, KMHK98], les systèmes de fichiers [BvLL03], les schémas de distribution optimale [OvdPE04], les systèmes temps-réel [Beh05] et l'implémentation de ces algorithmes dans les outils [SD97, Jou03].

Les travaux actuels visent essentiellement à étendre les différents résultats de la littérature pour porter les méthodes et outils de génération distribuée de grands espaces d'états sur des architectures massivement distribuées comme les grilles de calcul.

1.1.2 Réduction par équivalence

Etant donné que l'explosion d'états est le goulot d'étranglement pour la vérification basée sur les modèles, beaucoup de travaux de recherche sont dédiés aux techniques de réduction de l'espace d'états, dont les algorithmes les plus connus sont ceux donnés par Kanellakis-Smolka [KS90] et Paige-Tarjan [PT87]. La réduction d'espace d'états par équivalence permet de préserver les propriétés que l'on souhaite vérifier sur un système. Par exemple, l'équivalence de branchement préserve toutes les propriétés exprimables en logique temporelle arborescente CTL-X (c'est-à-dire la logique *Computational Tree Logic* sans l'opérateur *next*).

Deux grandes techniques de réduction existent : les techniques globales, qui visent à réduire un espace d'états explicite déjà généré, et les techniques locales ou à la volée, qui sont appliquées durant la génération de l'espace d'états.

Réductions globales. Il existe plusieurs raisons pour considérer la réduction d'espace d'états explicites déjà générés. Premièrement, si le modèle est (pratiquement) sans erreur et que nous voulons vérifier plusieurs propriétés sur lui, alors il sera plus efficace de réaliser la génération de l'espace d'états une fois pour toutes, de réduire le modèle une seule fois, et d'effectuer toutes les vérifications sur le modèle réduit. Une deuxième raison de considérer la réduction distribuée est qu'après la génération distribuée et la réduction distribuée, dans de nombreux cas il ne reste plus qu'un espace d'états qui rentre dans la mémoire d'une seule machine. Dans ces cas là, nous pouvons appliquer des algorithmes séquentiels existants d'analyse sur l'espace d'états réduit.

Dans les travaux réalisés jusqu'à présent, des algorithmes distribués globaux pour la réduction modulo l'équivalence forte et de branchement [BO02, BO03, BO05, Orz04] ont été conçus. Les deux types de réduction ont également été implémentées, et montrent des accélérations intéressantes sur des grappes de machines de tailles moyennes, et permettent d'analyser des systèmes plus larges qu'avec des solutions séquentielles.

Des travaux actuels visent à améliorer l'algorithme pour l'équivalence de branchement. En particulier, l'algorithme séquentiel très connu [GV90] prend pour hypothèse que dans une étape de précalcul tous les cycles de transitions τ sont fusionnés par un algorithme calculant les composantes fortement connexes (SCCs). Parce qu'il était largement admis que le problème de détection de SCCs était difficile

à distribuer, les algorithmes distribués actuels [BO05, BO04] fonctionnent encore avec la présence de cycles de τ. Cependant, des travaux récents [OvdP05] ont montré que contrairement à ce qui était cru, les cycles de τ peuvent être éliminés par un algorithme distribué assez efficacement sur des espaces d'états obtenus à partir d'études de cas. Cette extension devrait fournir un meilleur algorithme pour la réduction distribuée par équivalence de branchement, à la fois en temps et en mémoire.

Une autre direction actuelle de travail est la généralisation des algorithmes pour différentes équivalences. En particulier, les algorithmes distribués de réduction présentés ci-dessus, peuvent être appliqués aux équivalences qui ont une définition en termes de *signatures* (c'est-à-dire des structures de données qui lient les classes d'équivalences des états avec le comportement observable). Des travaux considèrent actuellement les équivalences stochastiques [HS99, DHS03, SD04], parce qu'elles peuvent être utilisées comme étape de précalcul pour la vérification de modèles stochastiques [HJ03, GH02]. Ces travaux étudient notamment si les équivalences stochastiques peuvent être exprimées en termes de signatures, ainsi que des algorithmes distribués pour le calcul de minimisation par équivalence stochastique d'espaces d'états distribués.

Réductions locales. Quelques exemples de techniques locales de réduction d'espace d'états sont la réduction d'ordre partiel, la réduction par symétrie, et l'interprétation abstraite.

Dus à des limitations essentiellement pratiques d'architecture logicielle, il existe très peu d'outils de réduction locale malgré l'existence de plusieurs algorithmes de réduction à la volée [BvdP02, PLM03].

Nous proposons d'appliquer notre approche de résolution distribuée à la volée de Sebs au problème de réduction locale d'ordre partiel par τ-confluence pour lequel, à notre connaissance, aucun algorithme distribué n'a été développé (voir le chapitre 4).

1.1.3 Comparaison par équivalence

Etant donné que les algorithmes de réduction calculent en fait les classes d'équivalences, ils peuvent donc être utilisés pour vérifier l'équivalence entre deux espaces d'états. Plus précisément, si les algorithmes de réduction sont appliqués sur l'union des deux espaces d'états, les états sources de chacun d'entre eux se retrouvent fusionnés dans le même état réduit si et seulement si les espaces d'états sont équivalents. Cette constatation est vraie pour n'importe quelle équivalence.

En conséquence, la discussion portant sur la distribution du problème de réduction d'espace d'états de la section 1.1.2 précédente, garde ici toute son importance.

Cependant, et contrairement à la réduction par équivalence, la comparaison par équivalence peut être facilement et naturellement effectuée à la volée, et de nombreux algorithmes séquentiels existent [Mat03a, Mat06, FM90].

Pourtant aucun algorithme distribué ou parallèle n'a été réalisé jusqu'à présent. Cette étude se propose de prendre le problème de comparaison par équivalence comme une autre application de notre approche distribuée générique (voir le chapitre 3).

1.1.4 Evaluation de formules logiques

Là où initialement la génération d'espace d'états depuis une spécification haut niveau était un objectif principal (et par conséquent développée de manière parallèle et distribuée [GMS01, Jou02, GMB+06, GMB+05, SD97, CGN98, MCC97, HBB99]), l'utilisation de la distribution a maintenant rendu la tâche de génération un problème mineur. A la place, le travail est maintenant concentré sur des

algorithmes distribués de vérification pour les logiques temporelles linéaires, comme LTL [LS99], et arborescentes, comme CTL ou le μ-calcul modal.

Depuis l'article originel sur l'évaluation de propriétés CTL [CES86], cette logique a été appliquée à de nombreuses études de cas industrielles et plusieurs outils de vérification associés ont été développés (par exemple, SMV [McM93] et PRISM [KNP02]). La distribution et parallélisation de l'évaluation de formules logiques a commencé avec la version parallèle du vérificateur MURϕ [SD97], et a été continuée avec la version distribuée de l'algorithme utilisé par SPIN pour vérifier des propriétés de sûreté [LS99], ainsi que celui du vérificateur symbolique de modèles temporisés UPPAAL [BHV00].

Une extension de l'algorithme de [LS99] a ensuite été réalisée [BBS01] pour les propriétés de vivacité, ainsi qu'une proposition élégante basée sur l'évaluation modulaire [BCY02]. D'autre travaux ont récemment amélioré cette approche par une méthode directe et efficace [BH05], où un système de 52 processeurs a pu être utilisé à 75%. En outre, des recherches en cours visent à rendre ces algorithmes plus optimisés au niveau des structures de données, et plus génériques de façon à pouvoir être appliqués de manière automatique, car à l'instant présent des fonctions codant les états doivent être développées manuellement pour réaliser de bonnes performances au niveau du stockage. De plus, les outils accessibles associés au travail de [BH05] ne sont optimisés que pour les réseaux de Pétri seulement, ce qui restreint son application et la possibilité de le comparer à d'autres travaux.

Une attitude générale est de rendre les techniques de vérification aussi souvent que possible indépendantes des méthodes de spécification. Pour cela, nous pouvons utiliser le μ-calcul modal [Koz83, BS01] qui est une logique très expressive, dans laquelle d'autres logiques, comme LTL et CTL*, peuvent être traduites. Plusieurs algorithmes distribués pour la résolution de fragments de faible alternance du μ-calcul modal ont aussi été développés, par exemple [LSW03, BLW01b, BLW01a] (basé sur la théorie des jeux à deux joueurs) et [BZ03] (une approche modulaire basée sur les assomptions).

Plusieurs travaux actuels visent à améliorer l'état de l'art en développant des algorithmes distribués pour vérifier le μ-calcul complet. Deux approches sont essentiellement suivies : la première maintient les vecteurs distribués d'ensemble de bits d'états satisfaisants (une approximation) des sous-formules. La distribution survient du fait que tous les états échangent de manière distribuée de l'information avec leurs successeurs à chaque cycle. La deuxième approche est basée sur la résolution du SEB associé au problème d'évaluation de formules logiques. Pour cela, il existe plusieurs points de départ dont l'un est l'algorithme basé sur l'élimination de Gauss [Mad97]. Cette procédure d'élimination de Gauss peut être très naturellement implémentée de manière distribuée. Le vrai objectif serait d'inclure des optimisations séquentielles comme celles décrites dans [MS03, GK04].

En développant nos algorithmes distribués de résolution locale de SEBs, nous répondons en partie à cet objectif, en permettant l'évaluation distribuée à la volée de propriétés logiques en μ-calcul modal d'alternance 1 à partir de SEBs d'alternance 1 correspondants (voir le chapitre 6).

1.1.5 Discussion

L'observation la plus commune et souvent avancée par des personnes expertes en vérification basée sur les modèles, est que la distribution de la vérification n'apporte qu'un gain de facteur constant par rapport à la vérification séquentielle, linéaire en nombre de nœuds utilisés pour le calcul.

Cette remarque est tout à fait fondée et l'attitude à avoir lors de la vérification de systèmes logiciels est bien d'utiliser des algorithmes aussi fins et intelligents que possible. De tels exemples d'algorithmes ont déjà été cités dans cette section, comme la réduction d'espace d'états par ordre partiel ou par symétries, la vérification compositionnelle, l'approche symbolique (bien que sujette au problème d'explosion des BDDs) et autres solutions évitant le problème d'explosion d'états par des techniques

toujours plus sophistiquées.

Seulement, le gain apporté par une solution distribuée, même d'un facteur constant par rapport à des solutions séquentielles, peut être significatif à la fois en temps d'exécution et en espace mémoire. Les grappes de stations de travail constituent une architecture distribuée largement utilisée dans le milieu industriel et académique, et certaines contiennent déjà des centaines de machines. L'ordre de grandeur gagné par l'utilisation de telles architectures permet déjà de savoir si une vérification est faisable ou non. De plus, les algorithmes distribués de vérification développés ces dernières années passent bien à l'échelle, contrairement aux solutions séquentielles.

D'autre part, avec l'avènement des architectures parallèles telles que les grilles de calcul ou les processeurs multicœur, et la stagnation des cadences (due à la loi de Moore et à la production de chaleur par les composants), il ne sera plus possible, comme c'est le cas aujourd'hui, d'augmenter les capacités d'un programme de vérification en partant du principe que les fréquences de cadence des processeurs augmentent. Pour exploiter la puissance de calcul des architectures de demain, il faudra paralléliser ces programmes.

En conséquence, la solution de la vérification distribuée, combinée avec des techniques séquentielles performantes, répond à l'évolution naturelle des machines en anticipant le futur de l'industrie logicielle.

1.2 Distribution des calculs

Nous introduisons dans cette section un certain nombre de notions utiles pour la distribution des calculs de vérification, ce qui nous permettra de préciser le contexte et les hypothèses sous lesquelles nous développerons, par la suite, les algorithmes distribués et outils de vérification.

1.2.1 Grappes de machines et grilles de calcul

Les premiers travaux de vérification basée sur les modèles ont concerné essentiellement des algorithmes séquentiels. Ce n'est que récemment, que des algorithmes parallèles et distribués de vérification ont été développés, utilisant d'abord des architectures à mémoire partagée comme des supercalculateurs [CCBF94, CCM95, NC97], et puis progressivement des architectures à processeurs symétriques multiples (SMPs) partageant également le même espace d'adressage [ADK97].

La tendance a ensuite évolué vers le développement d'algorithmes distribués efficaces pour grappes de PCs ou stations de travail (peu chères et très largement accessibles), avec un contrôle plutôt centralisé, et typiquement, une structure homogène [MCC97, NC97, SD97, KMHK98, KH99, HBB99, LS99, GMS01, Jou02, GMB+06, GMB+05]. Les travaux actuels et futurs s'orientent de plus en plus vers des architectures parallèles composées de grappes de machines interconnectées entre elles par des réseaux multiGigabit, plus couramment appelées grilles de calcul.

Les premiers efforts envisagés par ces travaux visent des grilles nationales comme Grid'5000 (France) ou Surfnet6 (Pays-Bas) avant d'être étendues, plus tard, à des grilles européennes et mondiales (comme Nordugrid). L'étape de passage d'une grappe de machines pour la vérification de modèles à la grille de vérification est cruciale, car elle repose en grande partie sur la robustesse des algorithmes employés. Non seulement le pôle de ressources disponibles dans une grille est dynamique dans le temps, mais aussi les capacités de chaque ressource disponible (c'est-à-dire les nœuds) ne sont pas connues à l'avance pour tout instant dans le temps. De plus, les délais de communication entre deux nœuds d'une grappe sont différents de ceux entre plusieurs grappes. La même remarque peut être faite pour les accès à la mémoire et aux disques. Ces hétérogénéités font que les techniques utilisées pour l'équilibrage

de charge dans les algorithmes de vérification développés pour les grappes (et réalisant de bonnes accélérations parallèles) et basés sur différentes formes de hachage, ne fonctionnent pas nécessairement bien pour les grilles. Ainsi, le développement de nouveaux algorithmes s'avérera indispensable.

Dans cette étude, nous ne cherchons pas à traiter les types de problèmes qui surviennent dans un contexte hétérogène de grille de calculs, comme la tolérance aux pannes, l'ordonnancement des tâches et la communication entre architectures différentes, qui sont autant de pistes de travaux futurs. Nous préférons utiliser dans un premier temps une architecture distribuée homogène, comme les grappes de PCs, afin de pouvoir expérimenter nos méthodes et outils sur des cas concrets en environnement réel. Cette mise en situation pratique implique déjà de nombreux aspects du calcul distribué, comme les mécanismes de base de communication par passage de messages, l'équilibrage de charge, le déploiement des processus sur les machines distantes et la détection distribuée de terminaison du calcul.

1.2.2 Modèle d'architecture distribuée

Le modèle de machine distribuée que nous visons est composé de P ordinateurs distribués faiblement couplés, également appelés *nœuds*. Chaque nœud, numéroté de 1 à P, a son propre processeur et sa propre mémoire. L'interconnexion entre plusieurs nœuds est réalisée par un réseau haut-débit standard. Nous nous concentrons sur de telles architectures à mémoire distribuée, comme les grappes de PCs (*cluster*) et les réseaux de stations de travail (NOWs), car elles sont accessibles dans les réseaux locaux (LANs) de la plupart des laboratoires de recherche et des entreprises. Un autre argument est qu'elles sont aussi très bon marché, qu'elles peuvent évoluer en puissance de calcul et en espace mémoire très facilement, contrairement aux supercalculateurs à processeurs multiples, très performants mais très chers et peu évolutifs. La figure 1.1 illustre un exemple de telle architecture matérielle distribuée.

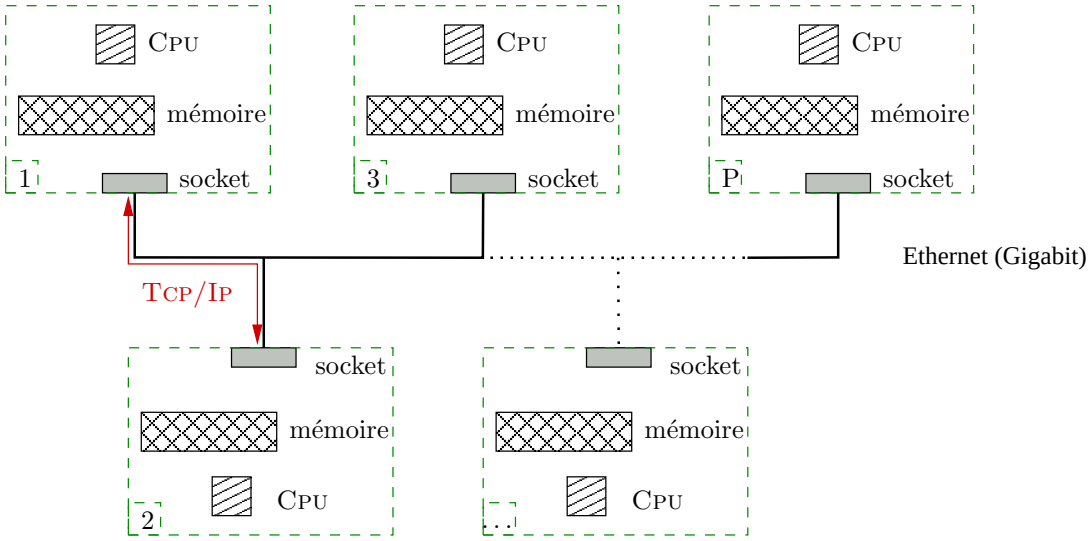


Figure 1.1: Architecture distribuée composée de machines homogènes interconnectées par un réseau standard

La seule hypothèse que nous faisons sur le réseau est que toutes les machines puissent communiquer deux à deux par échange de messages avec le protocole TCP/IP via des sockets. Concernant les

nœuds, nous supposons qu'ils sont homogènes en CPU, mémoire et système d'exploitation. Cette caractéristique facilite à la fois la conception des algorithmes distribués, comme par exemple au niveau de la répartition des données et de l'ordonnancement des tâches, qui sont des problèmes difficiles sur des architectures hétérogènes, mais également la programmation de ces algorithmes et leur expérimentation sur des architectures réelles à mémoire distribuée.

1.2.3 Modèle de programmation distribuée

Une distinction claire devrait être faite à ce niveau entre le parallélisme sur les machines à mémoire partagée (SMMs) et le parallélisme sur machines à mémoire distribuée (DMMs). Les algorithmes dans la première catégorie prennent généralement pour hypothèse la présence d'une seule mémoire et de plusieurs processeurs. Le nombre de processeurs disponibles est de l'ordre de la taille du problème, alors que pour les machines à mémoire distribuée, le nombre de processeurs est beaucoup plus petit et insignifiant par rapport à la taille du problème. Aussi, pour la mémoire partagée, la communication entre processeurs n'est pas un problème, alors que pour la mémoire distribuée, la *latence* (c'est-à-dire le temps de parcours sur le réseau) de la communication joue un rôle important.

Aussi, les programmes conçus pour la mémoire partagée ne peuvent pas facilement être exécutés sur une DMM, à cause, la plupart du temps, de la latence trop haute d'un système de mémoire partagée virtuelle. Par contre, dans l'autre direction (des DMMs vers les SMMs), la migration des programmes produit des résultats acceptables, comme il peut être montré par des exécutions sur des grappes de SMPs. Ainsi, deux écoles s'affrontent actuellement pour la thématique de la vérification distribuée et parallèle : l'une utilise les SMMs, et l'autre les DMMs.

En choisissant une architecture à mémoire distribuée, nous avons opté naturellement pour des modèles spécifiques de programmation et d'algorithmes distribués basés sur le paradigme du *passage de messages*, pour la transmission de tâches et de données à travers le réseau. Les programmes en résultant sont généralement constitués d'un ensemble de tâches identiques sur chacun des nœuds, et d'un ensemble de données, disjoint deux à deux sur chacun des nœuds. Ce type de programmation distribuée fait partie de la classe MIMD (*instruction multiple, donnée multiple*) établie par Flynn [Fly66], et dans notre cas, plus particulièrement de la classe SPMD (*programme unique, donnée multiple*) car un seul et même programme est exécuté sur chacun des nœuds.

Toutefois, ce même programme discerne un comportement asymétrique pour un des processus exécutés. En effet, la première instance (également appelée *superviseur*) de notre programme distribué, a un rôle totalement orthogonal (et donc un algorithme différent) par rapport à la finalité du calcul distribué (et donc par rapport aux autres instances, appelées *travailleurs*). Son rôle est d'initier chaque instance du programme sur les nœuds distants, de mettre en place les interconnexions entre les nœuds, de relever éventuellement des statistiques sur l'avancement de l'exécution globale du programme distribué, et enfin de permettre l'interaction avec l'utilisateur final du programme, au moyen d'une interface, en accédant par exemple à ses requêtes d'interruption du calcul et en lui indiquant l'état de terminaison du programme. Généralement, un tel processus (également appelé *maître* dans un schéma maître/esclave, ou encore *agent central* dans d'autres schémas) est exécuté sur la machine de l'utilisateur qui n'a, à ce titre, pas besoin de faire partie de l'ensemble des machines homogènes participant au calcul distribué.

Une représentation abstraite de cette topologie réseau et des P processus distribués peut être décrite par la partie gauche de la figure 1.2 et un exemple de mise en pratique réelle correspondante par la partie droite de la même figure.

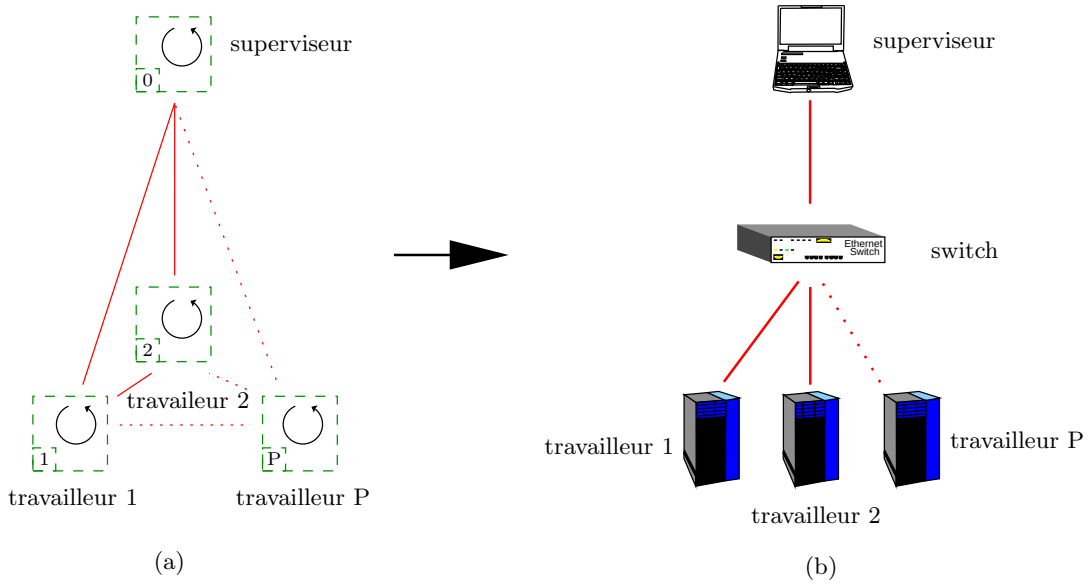


Figure 1.2: Modèle de programmation superviseur-travailleurs abstrait (a) et concret (b)

1.2.4 Modèle d'algorithme distribué

Nous considérons essentiellement un parallélisme de données, en exploitant la structure régulière des données de notre problème (c'est-à-dire les variables booléennes du SEB) et en appliquant en distribué un même calcul à des données différentes. Mais nous effectuons également un parallélisme de contrôle en décrivant notre application en termes de tâches indépendantes pouvant être réalisées en distribué. Notre parallélisme de données est obtenu en subdivisant les structures de données du programme et en distribuant les sous-structures sur les nœuds de calcul. En distribuant les données, nous distribuons ainsi également les calculs (*règle du propriétaire qui calcule*).

Nous assurons la distribution des données (c'est-à-dire le partitionnement du graphe booléen) par une fonction de hachage statique $h : V \rightarrow [1, P]$ connue de tous les nœuds. L'utilisation d'une fonction de hachage statique h garantit un bon équilibrage de charge, mais ne préserve pas nécessairement la localité des données, c'est-à-dire qu'elle ne minimise pas les dépendances entre les variables booléennes affectées à différents nœuds. Un schéma d'équilibrage de charge dynamique améliorerait la localité ainsi que l'équilibrage global de charge [KM04] dans le cas d'architectures hétérogènes, mais au prix de messages additionnels et de points de synchronisation.

En ce qui concerne les structures de données, nous avons décidé de mémoriser les données dans des tables de hachage et non dans d'autres types de structures de données telles que les arbres de recherche équilibrés, et cela pour deux raisons : d'une part, les tables de hachage sont des structures qui donnent de bons résultats en pratique en temps de recherche bien qu'elles aient théoriquement une complexité de recherche moins bonne que les arbres de recherche ; d'autre part, elles permettent une économie de mémoire non négligeable dans le cadre de la vérification de très larges SEBs car, par exemple, comparé aux arbres de recherche, elles ne nécessitent pas le stockage de pointeurs gauches et droits.

Nous avons choisi de développer dans un premier temps des algorithmes monothreadés (c'est-à-dire à un seul flot d'exécution), car ils sont plus faciles à implémenter et à expérimenter rapidement. Des variantes multithreadées de nos algorithmes seront proposées au cours de ce manuscrit, mais non

implémentées. Dans le cadre des expérimentations, nous nous limiterons à seulement une instance d'algorithme exécutée par nœud, bien que nos algorithmes aient été testés avec succès sur des grappes d'architectures symétriques à processeurs multiples (SMP) bi-XEON, avec deux travailleurs exécutés par machine, et une utilisation combinée de mémoire distribuée (globalement sur l'ensemble des nœuds) et partagée (localement sur chacun des nœuds).

1.2.5 Mesures de complexité

Pour évaluer les performances de nos conceptions distribuées, nous utilisons les mesures de complexité en temps, mémoire, et message pour les algorithmes distribués définis comme suit :

Définition 1.1 (Complexité en temps d'exécution)
La complexité en temps d'exécution dans le pire des cas est le temps maximal, parmi toutes les exécutions possibles pour toutes les entrées possibles, écoulé entre le moment où le premier nœud commence son exécution et le moment où le dernier nœud s'arrête.

■

Définition 1.2 (Complexité en espace mémoire)
La complexité en espace mémoire dans le pire des cas est l'espace mémoire maximal utilisé par tous les nœuds lors d'une exécution, parmi toutes les exécutions possibles pour toutes les entrées possibles.
■

Définition 1.3 (Complexité en nombre de messages)
La complexité en nombre de messages dans le pire des cas est le nombre maximal de messages envoyés par tous les nœuds lors d'une exécution, parmi toutes les exécutions possibles pour toutes les entrées possibles. ■

1.2.6 Mesures de performance

Dans de nombreux cas, la complexité théorique dans le pire des cas d'un algorithme distribué n'est pas un bon indicateur sur son utilité en pratique. Pour montrer les qualités d'un outil distribué, des mesures de performance sont aussi réalisées, d'habitude sur les types de données pour lesquelles l'outil est conçu (dans notre cas, il s'agit de l'espace d'états correspondant au système à vérifier).

Deux mesures importantes pour la performance pratique d'un algorithme (parallèle ou) distribué sont l'accélération et l'efficacité. Etant donné un problème de vérification, si T_s est le temps pris par le meilleur algorithme séquentiel pour résoudre le problème, et T_P le temps pris par l'algorithme distribué pour résoudre le même problème sur une machine distribuée avec P nœuds, l'accélération S_P et l'efficacité E_P sont définis comme suit :

$$S_P = \frac{T_s}{T_P}$$

$$E_P = \frac{S_P}{P}$$

Remarque 1.1
La valeur optimale pour l'accélération S_P est donc P. Un graphe montrant les accélérations d'une solution distribuée sur un même problème mais avec un nombre de nœuds variable, sera composé d'une courbe linéaire proche de la diagonale $y = x$ si l'accélération est bonne. Dans le cas contraire, elle sera proche de l'axe des abscisses. ■

Remarque 1.2
L'efficacité optimale, quant à elle, est de 1. Un algorithme distribué avec une bonne efficacité sur un même problème quelque soit le nombre de nœuds utilisés, se traduira par une courbe légèrement décroissante et proche de l'optimum $y = 1$. Dans le cas contraire, la courbe convergera très rapidement vers une asymptote proche de l'axe des abscisses. ■

Une autre mesure intéressante est celle du surcoût en mémoire qu'introduit un algorithme (parallèle ou) distribué par rapport à la consommation mémoire de la solution séquentielle optimale. Etant donné un problème de vérification, si M_s est l'espace mémoire pris par le meilleur algorithme séquentiel pour résoudre le problème, et M_i la mémoire prise par l'algorithme distribué sur le nœud i pour résoudre le même problème sur une machine distribuée avec P nœuds, le surcoût mémoire M_P est défini comme suit :

$$M_P = \frac{\sum_{i=1}^{P} M_i}{M_s}$$

Remarque 1.3
La valeur optimale pour le surcoût en mémoire M_P est de 1. Un algorithme distribué montrant un faible surcoût en mémoire sur un même problème pour un nombre de nœuds variable, sera décrit par une courbe parallèle à la courbe optimale $y = 1$, en étant aussi proche que possible de cette dernière. Dans le cas contraire, la courbe sera linéaire en nombre de nœuds utilisés et s'éloignera très rapidement de l'optimum. ■

1.2.7 Généricité et ouverture aux technologies existantes

Tout au long de notre étude, nous nous efforcerons d'avoir un maximum de généricité vis-à-vis des langages de modélisation employés et langages de spécification de propriétés, ainsi que sur les structures de données employées et formats d'échange de modèle. Nous ne fixerons donc pas de format de spécification des programmes distribués que l'on souhaite vérifier avec chacune des applications de vérification distribuée, car cela amène souvent à des sources d'inefficacité non désirées.

Les méthodes développées viseront à être particulièrement indépendantes du langage de modélisation utilisé (algèbre de processus, réseau de Pétri, ...), du langage de spécification de propriétés (μ-calcul, CTL, CSL, ...) et de la représentation de l'espace d'états employée (explicite, symbolique, à la volée, hybride).

Formats d'entrées/sorties. A cause du nombre grandissant d'outils de vérification par modèle, et les très différentes directions dans lesquelles ils évoluent, des efforts sont maintenant faits pour le développement de formats de stockage d'espace d'états standard et d'interfaces standard. Quelques exemples sont le format de fichier FC2 [MS93], le format textuel de l'outil ALDÉBARAN, AUT [FGK+96], le format BCG (*Binary Coded Graphs*) [Gar95], le format SVC [BvLL03], et l'environnement générique OPEN/CÆSAR [Gar98].

Le principal format d'entrées/sorties utilisé dans ce travail de thèse est l'environnement générique OPEN/CÆSAR [Gar98], qui, bien que non spécialisé pour des calculs distribués comme l'est par exemple le format SVC2 [BvLL03], permet d'être indépendant du format de description des modèles donnés en entrée aux différents outils de vérification connectés à cet environnement, pour peu qu'il existe une connexion entre le format décrivant le modèle et l'environnement OPEN/CÆSAR.

Communication par passage de messages. Nous prenons pour hypothèse que la topologie réseau est fortement connectée et fiable (aucun message n'est perdu ou corrompu), car chacun des nœuds

nécessite de travailler et communiquer symétriquement avec chacun des autres nœuds. Les canaux de communication entre deux nœuds sont bidirectionnels, et préservent l'ordre FIFO entre les messages ayant les mêmes nœuds source et destinataire.

En plus des choix architecturaux initiaux, nous souhaitons améliorer plus encore les performances de résolution de SEB, en utilisant une couche de communication qui permet :

1. la réduction de consommation mémoire,
2. la maximisation du recouvrement des communications avec des calculs,
3. la diminution d'attente active suite à des échecs d'émission et de réception,
4. la prévention de blocage durant la communication entre nœuds.

Le 1^{er} point peut être résolu en limitant la taille des tampons d'émission et de réception. Cependant, cela requiert de traiter les échecs d'émission et de réception ($3^{\text{ème}}$ point), à cause de tampons d'émission pleins ou de réception vides. Le $2^{\text{ème}}$ point requiert des opérations de communication asynchrones et non bloquantes à la fois en émission et en réception dans le but d'éviter des points de synchronisation. Etant donné que nous nous concentrons sur des algorithmes monothreadés, une priorité à grain fin (c'est-à-dire concernant de petites unités de traitement de quelques instructions seulement) entre les activités de communication et celles de calcul doit également être statiquement définie afin de mener à bien le $2^{\text{ème}}$ point. Le $3^{\text{ème}}$ point suggère l'utilisation alternée de communications non-bloquantes avec des communications bloquantes. Nous devons donc établir un compromis entre l'utilisation de communications bloquantes pour éviter les attentes actives, et le recouvrement des communications par du calcul, en utilisant des communications non-bloquantes. Finalement, le $4^{\text{ème}}$ point nous donne une solution partielle à ce problème en permettant les communications bloquantes seulement lorsque les nœuds sont inactifs (c'est-à-dire, lorsqu'il n'y a plus d'activité locale au nœud en attente de travail).

La figure 1.3 illustre une classification possible des programmes distribués, la prise en compte de la communication lors des choix de conception, et souligne notre choix d'utiliser principalement le passage de message asynchrone non bloquant avec des tampons de communication bornés.

En respectant les choix (cases *bleues* de la figure 1.3) de couche de communication cités précédemment, nous avons défini un ensemble de primitives de communication qui sont utilisées par nos algorithmes dans la suite de ce document :

- RECEIVE (**out** : $msg \in \{Evl(x : V, y : V), Exp(x : V, y : V), Trm, Idl(nb\_msg : \mathbb{N}), Act, Ack(stamp : \mathbb{N})\}$, $sender : [0..P]$) permet la réception bloquante d'un message msg envoyé par un nœud $sender$,
- IRECEIVE (**out** : $msg \in \{Evl(x : V, y : V), Exp(x : V, y : V), Trm, Idl(nb\_msg : \mathbb{N}), Act, Ack(stamp : \mathbb{N})\}$, $sender : [0..P]$) $\rightarrow$ **Bool** permet la réception immédiate (c'est-à-dire, non bloquante) d'un message msg venant d'un nœud $sender$. Elle retourne T si le message a été reçu avec succès, ou F si tous les tampons de réception correspondants sont vides,
- SEND (**in** : $msg \in \{Evl(x : V, y : V), Exp(x : V, y : V), Trm, Idl(nb\_msg : \mathbb{N}), Act, Ack(stamp : \mathbb{N})\}$, $receiver : [0..P]$) permet l'émission bloquante du message msg vers le nœud $receiver$,
- ISEND (**in** : $msg \in \{Evl(x : V, y : V), Exp(x : V, y : V), Trm, Idl(nb\_msg : \mathbb{N}), Act, Ack(stamp : \mathbb{N})\}$, $receiver : [0..P]$) $\rightarrow$ **Bool** permet l'émission immédiate (c'est-à-dire, non bloquante) du message msg vers le nœud $receiver$. Elle retourne T si le message a été émis avec succès, ou F si le tampon d'émission est plein, et
- WAITEVENT (**in/out** : $receiver\_set : 2^{[0..P]}$, $sender\_set : 2^{[0..P]}$) permet l'attente bloquante sur la détection d'événements de communication au niveau des tampons locaux de réception $receiver\_set$ et d'émission $sender\_set$ associés aux nœuds dans $[0..P]$.

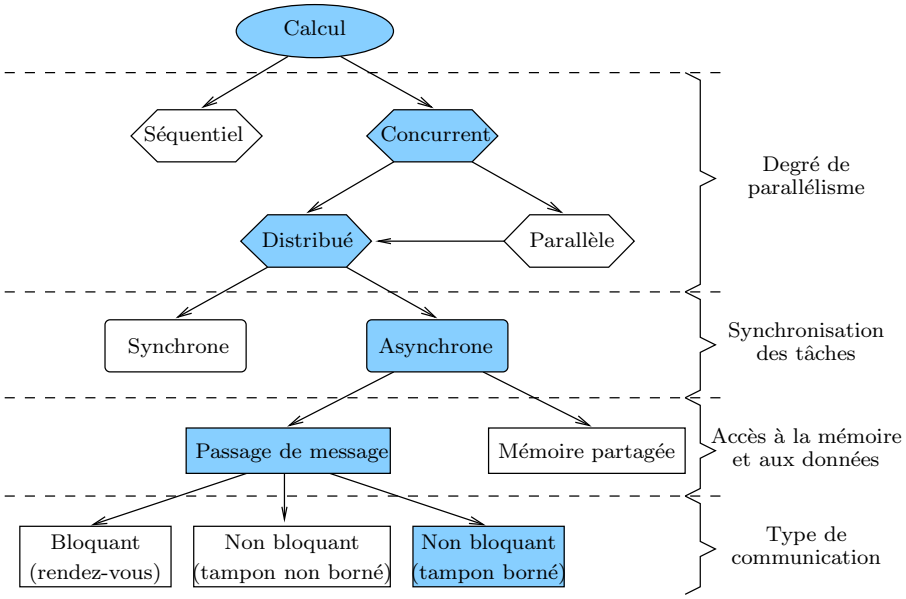


Figure 1.3: Résumé des choix (cases bleues) adoptés aux différents niveaux de conception d'un programme distribué

Remarque 1.4
En pratique, il est également possible de spécifier des nœuds ou des ensembles de nœuds particuliers avec lesquels une communication doit être réalisée. Par exemple, la primitive RECEIVE permet d'effectuer une réception bloquante d'un message msg (dont le type $Evl, Exp, Trm, Idl, Act, Ack$ sera défini avec notre algorithme distribué de résolution de SEB au chapitre 2) venant d'un nœud particulier *sender* ou bien de tous les nœuds présents dans le calcul. Ce comportement facilite les phases d'attente passive d'un nœud lorsqu'il devient inactif, puisqu'il est ainsi possible d'indiquer à l'instance du programme de s'endormir jusqu'à la réception d'un message venant de n'importe quel autre nœud. ■

Le lecteur familier avec le domaine du calcul haute performance s'apercevra que des primitives similaires sont présentes dans d'autres interfaces de passage de message connues et largement utilisées dans leur communauté, telles que MPI (*Message Passing Interface*) [MPI]. Seulement, nos primitives présentent des spécificités propres au calcul de vérification qui nécessite une utilisation fine de la mémoire et une bonne réactivité, en minimisant les temps d'attente et de synchronisation. Par exemple, elles doivent nous permettre de traiter les cas d'émissions asynchrones qui échouent, ainsi que la mise en tampon explicite des messages en attente d'émission ou de réception, et enfin l'absence de barrière de synchronisation après chaque séquence d'émissions et de réceptions, contrairement aux programmes MPI.

Environnement de prototypage : la boîte à outils CADP. Nous choisissons d'utiliser la boîte à outils CADP (*Construction and Analysis of Distributed Processes*) [GLM02] pour expérimenter nos résultats sur des problèmes de vérification réels et tirer parti des avancées théoriques et techniques qui y sont intégrées régulièrement.

Une architecture logicielle existante, très bien documentée, constamment améliorée, disponible sous différents systèmes d'exploitation (UNIX, WINDOWS, LINUX, MACOS) et largement diffusée dans le

monde académique et industriel, apporte de nombreux avantages directs.

- Un exemple de tel avantage est la distribution et le stockage des données pour lesquels la boîte à outils CADP offre une bibliothèque générique de fonctions de hachage ainsi qu'une bibliothèque de tables de hachage, permettant ainsi une expérimentation rapide et efficace de nos algorithmes et des connexions facilitées aux différents outils existants de CADP.
- En respectant l'interface de programmation (API) de la bibliothèque CÆSAR_SOLVE de résolution séquentielle de SEBs, nous pouvons intégrer nos nouveaux algorithmes de résolution distribuée de manière efficace et directement utilisable par les outils de vérification de CADP. Le but final est d'obtenir des versions distribuées, de façon totalement transparente pour l'utilisateur, pour chacun des outils séquentiels de vérification de CADP utilisant la bibliothèque CÆSAR_SOLVE.

Le succès de la diffusion de CADP (plus de 340 sites) réside en partie sur sa facilité d'installation (au moyen d'un assistant graphique) et du faible pourcentage de logiciels tiers.

Pour cela, et afin d'avoir une version implémentée de nos algorithmes facilement intégrée et distribuée dans CADP, nous avons préféré utiliser dans un premier temps une bibliothèque de communication "minimale" (contenant seulement 40 appels de procédure, contre 200 dans MPI, ayant prouvé leur capacité à modéliser de nombreux programmes de vérification distribuée), faiblement couplée, basée sur des sockets UNIX avec tampons de communication bornés et protocoles TCP/IP, et qui fait usage de protocoles standards de connexion à distance, tels que rsh/rcp, ssh/scp, etc. Une telle couche de communication présente deux autres avantages qui sont le contrôle fin de la consommation mémoire, en réglant la taille des tampons de communication de manière appropriée, et l'agrégation des messages (particulièrement efficace pour les petits messages) avant leur émission afin de limiter le nombre d'appels système très coûteux en temps d'exécution.

1.3 Modélisation des systèmes distribués

Dans le but de valider nos algorithmes distribués dans le contexte des applications de vérification, il est nécessaire d'introduire quelques notions supplémentaires sur les *systèmes* que l'on souhaite vérifier ainsi que sur leur représentation formelle en termes de modèles.

Les applications réparties (telles que les protocoles de communication et les systèmes distribués) peuvent être représentées comme des *systèmes asynchrones* composés de plusieurs entités ("boîtes noires") qui évoluent de manière concurrente et communiquent par échange de messages. Le comportement d'un tel système est décrit par les *actions* (ou *événements*) qu'il effectue au cours de son exécution. Une particularité d'un tel formalisme réside dans la *sémantique d'entrelacement*, où plusieurs actions différentes ne peuvent être exécutées simultanément. Il existe d'autres sémantiques du parallélisme, notamment celles utilisées dans les langages *synchrones*, où plusieurs actions peuvent se produire en même temps. Dans cette étude, nous ne considérons que des langages de description ayant une sémantique d'entrelacement, car plus adaptés à la modélisation de l'indéterminisme et de l'asynchronisme.

1.3.1 Système de transitions étiquetées

Conformément à la sémantique d'entrelacement, le comportement d'un programme parallèle peut être naturellement modélisé par un graphe (ou automate) dont les sommets représentent les états du programme et les arcs (ou transitions) dénotent les changements d'état du programme suite aux actions qu'il effectue durant son exécution.

Il existe généralement deux classes de modèles de programmes parallèles, employés suivant le fait que les informations pertinentes sont attachées aux états ou aux transitions du programme : les *structures de Kripke* et les *systèmes de transitions étiquetées* (STEs). Les définitions formelles de ces modèles sont données ci-dessous.

On considère un ensemble $\mathcal{Q}$ dont les éléments sont appelés *états*, un ensemble $\mathcal{A}$ dont les éléments sont appelés *actions*, et une action particulière, τ, appelée action *invisible* ou *interne*, et non incluse dans $\mathcal{A}$.

Définition 1.4 (Structure de Kripke)
Une *structure de Kripke* K est un quadruplet (Q, P, L, T), où :

- Q est un ensemble d'*états*, notés q_1, q_2, ... ;
- P est un ensemble de *propositions atomiques*, notées p_1, p_2, ... ;
- $L : Q \to 2^P$ est un *étiquetage*, associant à chaque état q l'ensemble des propositions atomiques satisfaites par q ;
- $T \subseteq Q \times Q$ est la *relation de transition*. Les éléments $(q_1, q_2) \in T$ (notés aussi $q_1 \to q_2$) sont appelés *transitions*.

Un *chemin* π est une séquence (finie ou infinie) $q_0 \to q_1 \to \ldots$ d'états. Le $k^{\text{ième}}$ état de π est noté $\pi(k)$ et le $k^{\text{ième}}$ suffixe $(q_k \to q_{k+1} \to \ldots)$ de π est noté π^k. Un chemin π est dit *maximal* soit s'il est infini, soit s'il est fini et que son dernier état q_n n'a pas de successeur (c'est-à-dire qu'il n'existe pas d'état $q \in Q$ tel que $q_n \to q$). L'ensemble des chemins maximaux d'une structure de Kripke est noté R. ■

Définition 1.5 (Système de transitions étiquetées)
Un *système de transitions étiquetées* M est un quadruplet (Q, A, T, q_0), où :

- $Q \subseteq \mathcal{Q}$ est l'ensemble des *états* de M, notés $q_1,\ q_2,\ \ldots$;
- $A \subseteq \mathcal{A} \cup \{\tau\}$ est l'ensemble des *actions* de M, notées $a_1,\ a_2\ \ldots$;
- $T \subseteq Q \times A \times Q$ est la *relation de transition*. Les éléments $(q_1,\ a,\ q_2) \in T$ (notés aussi $q_1 \xrightarrow{a} q_2$) sont appelés *transitions* et
- $q_0 \in Q$ est l'*état initial* de M.

Un *chemin* π est une séquence (finie ou infinie) $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \ldots$ d'états. Le $k^{\text{ième}}$ état de π est noté $\pi(k)$, la $k^{\text{ième}}$ action de π est notée $l(\pi, k)$ et le $k^{\text{ième}}$ suffixe $(q_k \xrightarrow{a_k} q_{k+1} \xrightarrow{a_{k+1}} \ldots)$ de π est noté π^k. Un chemin π est dit *maximal* soit s'il est infini, soit s'il est fini et que son dernier état q_n n'a pas de successeur (c'est-à-dire qu'il n'existe pas d'état $q \in Q$, ni d'action $a \in A$ tels que $q_n \xrightarrow{a} q$). L'ensemble des chemins maximaux d'un STE est noté R. ■

Remarque 1.5
Un système de transitions étiquetées peut être vu comme un graphe orienté, muni d'un état initial, et dont les arcs sont étiquetés par une action. Par conséquent, l'ensemble du vocabulaire défini pour les graphes reste valide, en particulier les notions usuelles de *chemin* (voir la définition donnée ci-dessus), *chaîne* et *circuits*. ■

Les structures de Kripke sont des machines d'états finis non-déterministes dont les états sont étiquetés avec des variables booléennes, qui sont les valeurs des expressions dans cet état. Les structures de Kripke n'attachent aucune information aux actions effectuées par le programme ; elles sont utilisées comme modèles d'interprétation pour des logiques temporelles comme LTL [Lam80], PTL [MP92],

CTL [CES86], CTL* [CES86, EH86], etc., qui permettent d'exprimer des propriétés portant sur les états. Ces modèles conviennent aux langages de description comme ESTELLE [ISO89], SDL [IT92] ou PROMELA [Hol91], et sont au cœur d'outils de vérification tels que NUSMV [CCGR00], ou encore SPIN [Hol97].

En revanche, les systèmes de transitions étiquetées n'attachent aucune information aux états du programme ; ils sont utilisés comme modèle d'interprétation pour des logiques modales et temporelles comme HML [HM85], ACTL [NV90], ACTL* [NV90], μ-calcul modal [Koz83], etc., qui permettent d'exprimer des propriétés portant sur les actions. Ces modèles conviennent aux algèbres de processus comme CCS [Mil80], CSP [Hoa78] ou ACP [BK84], ainsi qu'aux langages de description basés sur ces algèbres, comme ELOTOS [ISO01] ou μCRL2 [Gro05].

Dans cette étude, nous adoptons le modèle STE, car il est bien adapté à la vérification de problèmes traduisibles vers la résolution de SEB, pour laquelle il existe de nombreux algorithmes.

1.3.2 Représentation explicite et implicite

Le STE est un modèle indépendant du langage source décrivant le système à vérifier. Il est également indépendant des outils de vérification.

Un exemple de STE est représenté sur la figure 1.4. Il contient 12 états et 20 transitions, 4 étiquettes d'action nommées `PUT !0`, `PUT !1`, `GET !0`, `GET !1` et enfin une action invisible τ (indiquée par i dans la figure). Cet exemple illustre de manière explicite les états d'un protocole de communication (en l'occurrence, celui du *Bit Alterné* [Gar89] avec deux messages différents) spécifié dans un langage formel (*full* LOTOS avec données). La taille du graphe du STE dépend ici du nombre possible de messages différents échangés.

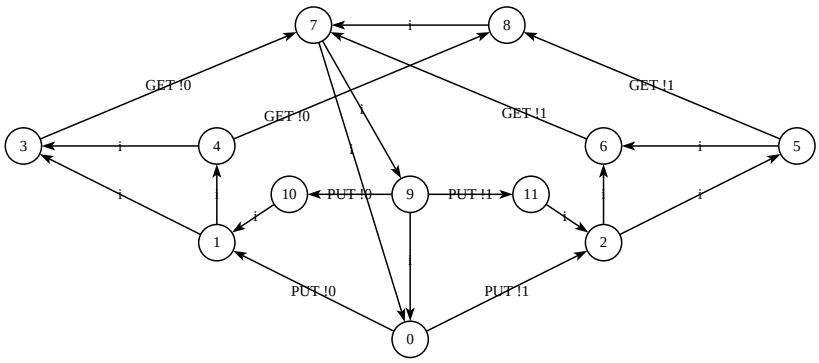


Figure 1.4: Système de transitions étiquetées (STE) représentant le protocole du Bit Alterné avec deux messages

Une méthode ou outil capable de vérifier un modèle STE généré à partir d'un programme parallèle

doit naturellement reposer sur une représentation concrète (ou *implémentation*) de ce modèle. Il existe habituellement deux représentations différentes du modèle STE : la représentation explicite et la représentation implicite.

Représentation explicite. Elle consiste à mémoriser l'ensemble de tous les états, les actions et la relation de transition du STE, ainsi que diverses autres informations provenant du programme source à partir duquel il a été généré (types, fonctions, etc.). Les outils qui l'exploitent peuvent ainsi fournir à l'utilisateur des diagnostics précis dans les termes du programme source. Cette représentation permet d'accéder aux valeurs contenues dans les états et dans les étiquettes des transitions, ainsi que d'explorer la relation de transition du STE (accès à l'état initial, aux successeurs et prédécesseurs des états et des transitions).

Représentation implicite. Elle consiste à fournir uniquement la fonction successeur du STE. Une telle fonction successeur peut être décrite formellement de la manière suivante.

Définition 1.6 (Parcours interactif de STE)
Pour parcourir interactivement les STEs, nous utiliserons la fonction successeur définie comme suit :

$$\begin{array}{rcl} succ & : & Q \rightarrow 2^{A \times Q} \\ succ(s) & = & \{(a, s') \mid s \xrightarrow{a} s' \in T\} \end{array}$$

■

La représentation implicite permet l'accès à l'état initial et aux transitions successeurs d'un état donné, afin d'explorer efficacement des STEs qui ne nécessitent pas d'être construits entièrement avant d'être parcourus.

Etant donné que nous cherchons à développer des algorithmes distribués de résolution à la volée de SEB pour effectuer la vérification à la volée de programmes parallèles, nous utiliserons la représentation implicite des STEs (et par suite des SEBs), pour effectuer nos calculs de vérification, appelée dans ce cas *vérification énumérative*, car nous choisissons de générer explicitement les états et transitions (et non de les regrouper ou fusionner comme dans les méthodes symboliques). Comme indiqué dans la section 1.2.7, nous cherchons à développer des outils de vérification distribuée à la volée les plus génériques possibles. Une façon d'atteindre cet objectif est d'utiliser l'environnement OPEN/CÆSAR [Gar98], présent dans la boîte à outils CADP, qui permet non seulement l'exploration à la volée (pour des outils de simulation, vérification ou génération de tests) des STEs à partir de programmes parallèles, mais aussi l'indépendance vis-à-vis du langage de description de ces programmes à vérifier.

——— ⋆ ——— ⋆ ——— ⋆ ———

Ainsi, en ayant un regard averti sur les différentes technologies présentes dans le domaine des méthodes formelles et dans celui du calcul scientifique de haute performance, il nous est alors possible de développer de nouveaux algorithmes et implémentations conciliant la puissance des architectures de calcul de haute performance à mémoire distribuée avec la résolution de problèmes de vérification. C'est ce que nous nous proposons de présenter dans la suite de ce document.

Chapitre 2

Résolution distribuée à la volée de SEBs monoblocs

Pour répondre à la limitation en taille des modèles à états finis pouvant être vérifiés ou traités par les techniques actuelles, nous proposons la distribution du modèle et de son traitement sur un ensemble de machines interconnectées, afin de tirer profit de la puissance de calcul et de mémoire globalement additionnée. Cette solution complémentaire aux techniques de minimisation, de construction compositionnelle du modèle ou encore d'abstraction du problème à traiter, permet un gain immédiat en temps et en mémoire utilisée par machine, ainsi que la possibilité de traiter des problèmes de taille proportionnelle à la puissance de calcul utilisée. Cette technique est donc très prometteuse, sachant que les processeurs, mémoires et systèmes d'interconnexion ne cessent de s'améliorer et de gagner en efficacité.

——— ⋆ ——— ⋆ ——— ⋆ ———

Ce chapitre est dédié à la description de notre algorithme DSolve pour la résolution distribuée à la volée de Sebs monoblocs sur un ensemble de machines homogènes de type grappe de Pcs [JM05, JM04]. Nous introduisons, dans la section 2.1, le modèle des systèmes d'équations booléennes, qui constitue une représentation unifiée des différents problèmes de vérification énumérative, ainsi que le mécanisme séquentiel de résolution associée. Dans la section 2.2, nous considérons les schémas de distribution des tâches et des données de la résolution séquentielle de Seb. Dans la section 2.3, nous décrivons l'algorithme DSolve en détail, en considérant la partie de résolution de Seb et la communication entre les processus. Nous donnons également des estimations de complexité à la fois en termes de temps de calcul, de mémoire utilisée, et de messages échangés. Nous examinons ensuite, dans la section 2.4, le processus responsable de la coordination et de la détection de terminaison du calcul distribué. Dans la section 2.5 nous développons le contexte de validation expérimentale de notre algorithme distribué. Finalement, la section 2.6 présente nos résultats d'expérimentation, notamment des mesures d'accélération, d'efficacité, de consommation mémoire et de surcoût de communication.

2.1 Systèmes d'équations booléennes

Les systèmes d'équations booléennes [Mad97] sont une structure formelle largement étudiée pour la vérification de systèmes d'états finis concurrents. Il est possible, par exemple, de formuler les

problèmes de vérification par évaluation de logiques temporelles, de comparaison par équivalences, ou encore de réduction par τ-confluence, en terme de résolution de SEBs.

Les premiers algorithmes avec une complexité polynomiale pour résoudre des SEBs d'alternance 1, ont été proposés par Arnold et Crubillé [AC88], par Cleaveland et Steffen [CS91b], et par Vergauwen et Lewi [VL92]. Ces algorithmes étaient *globaux*, c'est-à-dire qu'ils nécessitaient que le SEB soit complètement construit avant de commencer la résolution, et ils calculaient la solution pour *toutes* les variables de ce SEB ; les algorithmes donnés dans [CS91b, VL92] ont une complexité linéaire dans la taille du SEB (nombre de variables et d'opérateurs).

Hors, il arrive très souvent que nous ne soyons intéressés que par la résolution d'une seule variable du SEB, appelé *variable principale* ou *variable d'intérêt*, par exemple, lorsque l'on traduit en termes de SEBs le problème de comparaison par équivalence ou celui de la vérification de logique temporelle. Des travaux de recherche ont été entrepris pour concevoir des algorithmes à la volée (*locaux*), qui sont capables de calculer la valeur d'une seule variable en examinant seulement les portions du SEB influençant cette variable. Les premiers algorithmes à la volée pour résoudre des SEBs d'alternance 1 ont été proposés par Larsen [Lar92], par Andersen [And94], et par Vergauwen et Lewi [VL94] ; les algorithmes donnés dans [And94, VL94] ont une complexité linéaire dans la taille du SEB. Un autre algorithme local en temps linéaire, appelé LMC, a été proposé par Du, Smolka et Cleaveland [DSC99] : l'algorithme accélère la résolution du SEB en détectant les composantes fortement connexes du graphe de dépendances entre les variables et a été utilisé avec succès pour l'analyse de protocoles de communication de taille industrielle.

La théorie sous-jacente aux SEBs et leur relation avec les logiques modales et le μ-calcul ont été étudiés par Mader [Mad97], qui a également proposé plusieurs algorithmes efficaces locaux et globaux basés sur des méthodes d'élimination de Gauss.

Cependant, les algorithmes séquentiels de résolution et les ressources des ordinateurs actuellement disponibles ne permettent pas un passage à l'échelle de la résolution pour des SEBs de grande taille (par exemple, contenant plus de 10^8 variables) ; en conséquence, des solutions distribuées deviennent nécessaires.

Une formulation alternative de la comparaison par équivalence et de la vérification de logique temporelle en termes de *graphes de jeux*, a aussi été proposée par Stevens et Stirling [SS98]. Des travaux récents [BLW01b, BLW01a, BLW02, LSW03, HLL04] ont fait usage des graphes de jeux comme représentation intermédiaire du problème de vérification distribuée (globale et locale) de formules de logique temporelle exprimée en μ-calcul modal d'alternance 1.

Dans la suite de ce document, nous examinons le modèle relativement récent [And94] permettant de décrire des problèmes de vérification sous forme de SEBs et de les résoudre linéairement dans la taille des SEBs. Nous en donnons une définition formelle, deux types de représentation et enfin deux techniques de vérification les manipulant.

2.1.1 Définition

Définition 2.1 (Système d'équations booléennes)
Soit $\mathcal{X}$ un ensemble de variables booléennes. Un *système d'équations booléennes* (SEB) sur $\mathcal{X}$ [And94, Mad97] est un tuple $B = (X, M_1, \ldots, M_n)$, où :

- $X \in \mathcal{X}$ est une variable booléenne et
- $M_i = \{X_j \overset{\sigma_i}{=} op_j \boldsymbol{X}_j\}_{j \in [1,m_i]}$ ($i \in [1,n]$) sont des blocs d'équations de plus petit (*resp.* plus grand) point fixe de signe $\sigma_i = \mu$ (*resp.* $\sigma_i = \nu$), et $op_j \in \{\wedge, \vee\}$.

La partie droite de chaque équation j est une formule purement disjonctive ou conjonctive obtenue en appliquant un opérateur booléen $op_j \in \{\vee, \wedge\}$ à un ensemble de variables $\boldsymbol{X}_j \subseteq \mathcal{X}$, chacune de ces

dernières étant définie en partie gauche d'une équation d'un bloc M_i de B. Les constantes booléennes F et T sont représentées par la disjonction vide $\vee\emptyset$ et la conjonction vide $\wedge\emptyset$. ■

La figure 2.1 montre un SEB composé de trois blocs de polarités différentes, deux (M_1 et M_2) sont définis par un plus petit point fixe, et le troisième (M_3) par un plus grand point fixe. Cet exemple montre une représentation textuelle possible d'un SEB avec pour chaque variable booléenne utilisée en partie droite d'une équation booléenne, l'existence d'une définition de cette variable en partie gauche. Pour chaque variable externe au bloc courant, comme X_4 et X_7 dans le bloc M_1, un cercle blanc repère ce type de variable dans la figure et un lien vers le bloc et l'équation booléenne la définissant est également illustré.

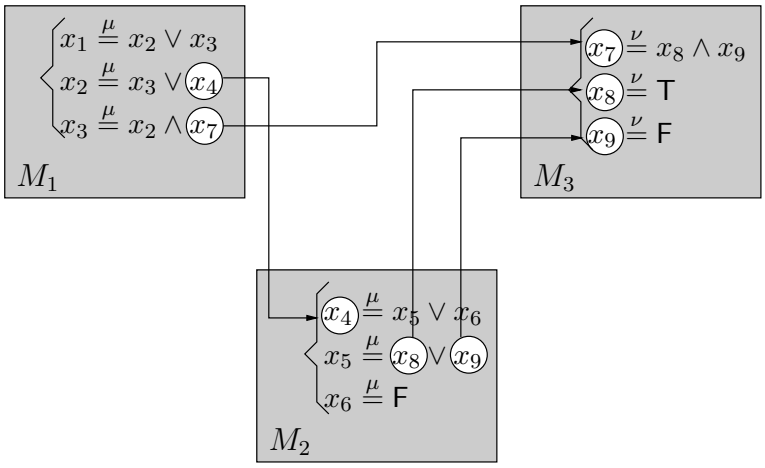


Figure 2.1: Un exemple de SEB multibloc représenté sous forme textuelle

Remarque 2.1
Cette classe de SEBs, appelés *simples* [AC88], est commode pour concevoir des algorithmes de résolution efficaces, tout en restant généraux : n'importe quel SEB contenant des combinaisons de disjonctions et de conjonctions dans la partie droite des équations peut être rendu simple (au prix d'une augmentation linéaire en taille) en factorisant les sous formules purement disjonctives ou conjonctives grâce à des variables et équations additionnelles. ■

Définition 2.2 (Sémantique d'une équation booléenne)
On nomme contexte δ, une fonction partielle $\delta : \mathcal{X} \rightarrow \mathbf{Bool}$, avec $\mathbf{Bool} = \{\mathsf{F}, \mathsf{T}\}$, qui associe une valeur booléenne à certains éléments de $\mathcal{X}$. La sémantique d'une *formule booléenne* $op_i\{X_1, \ldots, X_k\}$ dans un *contexte* δ, donnant une valeur booléenne à $X_1, \ldots, X_k$, est la valeur booléenne définie comme suit :

$$[\![op_i\{X_1, \ldots, X_k\}]\!]\delta = \delta(X_1)\ op_i\ \ldots\ op_i\ \delta(X_k)$$

■

Définition 2.3 (Sémantique d'un bloc d'équations booléennes)
La sémantique d'un *bloc d'équations booléennes* $M_i = \{X_j \overset{\sigma_i}{=} op_j \boldsymbol{X}_j\}_{j\in[1,m_i]}$ dans un contexte δ est le point fixe σ_i d'une fonctionnelle vectorielle $\Phi_{i\delta} : \mathbf{Bool}^{m_i} \rightarrow \mathbf{Bool}^{m_i}$:

$$[\![\{X_j \overset{\sigma_i}{=} op_j \boldsymbol{X}_j\}_{j\in[1,m_i]}]\!]\delta = \sigma_i \Phi_{i\delta}$$

où

$$\Phi_{i\delta}(b_1,\ldots,b_{m_i}) = (\llbracket op_j \boldsymbol{X}_j \rrbracket(\delta \oslash [b_1/X_1,\ldots,b_{m_i}/X_{m_i}]))_{j\in[1,m_i]}$$

et où $\delta \oslash [b_1/X_1,\ldots,b_n/X_n]$ représente un contexte identique à δ excepté pour les variables booléennes $X_1,\ldots,X_n$, auxquelles sont affectées les valeurs booléennes $b_1,\ldots,b_n$, respectivement. ■

Remarque 2.2
Il faut noter que l'absence de négations dans la partie droite des formules booléennes assure la monotonicité de la fonctionnelle $\Phi_{i\delta}$ et l'existence de son plus grand ou plus petit point fixe. ■

Définition 2.4 (Variables et blocs acycliques, conjonctifs et disjonctifs)
Une variable X_j dépend d'une variable X_l si $X_l \in \boldsymbol{X}_j$. Un bloc M_i dépend d'un bloc M_k si au moins une variable de M_i dépend d'une variable définie dans M_k. Un bloc est *fermé* s'il ne dépend d'aucun autre bloc. Un bloc M_i est *acyclique* s'il n'y a aucune dépendance cyclique entre les variables définies dans M_i. Une variable X_j est appelée *disjonctive* (*resp. conjonctive*) si $op_j = \vee$ (*resp.* $op_j = \wedge$). Un bloc M_i est disjonctif (*resp.* conjonctif) si chacune de ses variables est soit disjonctive (*resp.* conjonctive), soit dépend d'au plus une variable définie dans M_i, tous ses autres dépendances étant des constantes ou des variables définies dans d'autres blocs. ■

Remarque 2.3
Un SEB n'ayant pas de dépendance cyclique entre ses blocs n'implique pas le fait qu'il ne contienne pas de dépendance cyclique entre les variables d'un de ses blocs. ■

Définition 2.5 (Sémantique d'un SEB d'alternance 1)
Un SEB est *d'alternance* 1, également appelé *sans alternance*, s'il n'a aucune dépendance cyclique entre ses blocs ; dans ce cas, les blocs sont considérés comme étant triés topologiquement de telle sorte qu'un bloc M_i dépend seulement de blocs M_k avec $k > i$.

La sémantique d'un SEB d'alternance 1 $(X, M_1, \ldots, M_n)$ est la valeur de sa variable principale X donnée par la solution de M_1, c'est-à-dire, par $\delta_1(X)$, où les contextes δ_i sont calculés de la manière suivante :

$$\begin{array}{rcl} \delta_1(X) & = & \llbracket (X, M_1, \ldots, M_n) \rrbracket \\ \delta_n & = & \llbracket M_n \rrbracket [] \\ \delta_i & = & (\llbracket M_i \rrbracket \delta_{i+1}) \oslash \delta_{i+1} \text{ pour } i \in [1, n-1] \end{array}$$

■

La figure 2.1 illustre un SEB multibloc d'alternance 1. En effet, il n'existe pas de dépendance cyclique entre les blocs M_1, M_2 et M_3.

Remarque 2.4
On peut noter que le contexte pour interpréter M_n est vide (puisque M_n est fermé dans le cas d'un SEB d'alternance 1), et qu'un bloc M_i est interprété dans le contexte de tous les blocs M_k avec $k > i$ (c'est-à-dire M_i dépend de $M_{i+1} \ldots M_n$. ■

2.1.2 Représentation explicite et implicite

Nous développons nos algorithmes de résolution en termes de *graphes booléens* [And94], qui fournissent une représentation graphique plus intuitive des SEBs.

Définition 2.6 (Graphe booléen)
Etant donné un bloc d'équations $M_i = \{X_j \stackrel{\sigma_i}{=} op_j \boldsymbol{X}_j\}_{j\in[1,m_i]}$, le graphe booléen correspondant est un tuple $G = (V, E, L)$, où :

- $V = \{x_j \mid j \in [1, m_i]\}$ est l'ensemble des *sommets* (variables booléennes),
- $E = \{x_j \to x_k \mid j \in [1, m_i] \wedge X_k \in \boldsymbol{X}_j\}$ est l'ensemble des *arcs* (dépendances entre variables booléennes), et
- $L : V \to \{\vee, \wedge\}$, $L(x_j) = op_j$ est *l'étiquetage des sommets* (opérateur booléen disjonctif ou conjonctif).

L'ensemble des successeurs d'un sommet x est noté $E(x)$, la fonction successeur associée au graphe !booléen étant :

$$E \ : \ V \to 2^V, \ E(x) \ = \ \{y \mid x \to y \in E\}$$

Une représentation *implicite* du graphe booléen n'utilisera que la fonction successeur E, alors qu'une représentation *explicite* permettrait en plus l'utilisation d'une fonction prédécesseur à partir du graphe booléen construit entièrement avant toute résolution. Les sommets $\vee$ (*resp.* les sommets $\wedge$) puits représentent des variables égales à F (*resp.* T). ■

Remarque 2.5
La convention syntaxique que nous avons adoptée ci-dessus et dans le reste de cette étude est d'utiliser des majuscules (c'est-à-dire, X_k) pour les variables du SEB et des minuscules correspondantes (c'est-à-dire, x_k) pour le graphe booléen afin de faciliter la lecture d'expressions décrivant la traduction de l'un des formalismes dans l'autre. ■

Remarque 2.6
Nous utiliserons par la suite indistinctement la notation $\vee$-variable et $\wedge$-variable à la place de variable de type $\vee$ et variable de type $\wedge$ afin qu'il n'y ait pas d'ambiguïté d'interprétation. De la même manière, nous utiliserons cette terminologie pour les successeurs et prédécesseurs de variables booléennes. Ainsi, $\vee$-successeur (*resp.* $\wedge$-successeur) représentera une variable booléenne successeur de type $\vee$ (*resp.* $\wedge$), alors que $\vee$-prédécesseur (*resp.* $\wedge$-prédécesseur) représentera une variable booléenne prédécesseur de type $\vee$ (*resp.* $\wedge$). ■

2.1.3 Résolution globale et locale

De nombreux algorithmes pour résoudre les SEBs ont été proposé (voir [Mad97, chapitre 6] pour un état de l'art).

Deux sortes de problèmes de résolution sont habituellement définis pour les SEBs :

- la résolution *globale* requiert d'évaluer (c'est-à-dire, de calculer leurs valeurs telles que données par la fonctionnelle Φ_δ) toutes les variables $x_1, \ldots, x_n$ du SEB
- la résolution *locale* (ou à la volée) requiert d'évaluer une seule variable d'intérêt x_k.

Définition 2.7 (Résolution globale)
Soit un SEB $B = (X, M_1, \ldots, M_n)$ d'alternance 1. Le calcul de la sémantique d'un bloc se fait généralement selon l'algorithme suivant :

1. En traitant par ordre inverse des dépendances :

$$\begin{array}{rcl} \delta_1(X) & = & [\![(X, M_1, \ldots, M_n)]\!] \\ \delta_n & = & [\![M_n]\!][] \\ \delta_i & = & ([\![M_i]\!]\delta_{i+1}) \oslash \delta_{i+1} \text{ pour } i \in [1, n-1] \end{array}$$

2. Appliquer directement le théorème de Knaster-Tarski [Tar55] :

$$\begin{array}{lll} [\![\{X_j \stackrel{\mu}{=} op_j \boldsymbol{X}_j\}_{j\in[1,m_i]}]\!]\delta & = \mu\Phi_\delta = & \bigcup_{k\geq 0} \Phi_\delta^k(\mathsf{F}, \ldots, \mathsf{F}) \\ [\![\{X_j \stackrel{\nu}{=} op_j \boldsymbol{X}_j\}_{j\in[1,m_i]}]\!]\delta & = \nu\Phi_\delta = & \bigcap_{k\geq 0} \Phi_\delta^k(\mathsf{T}, \ldots, \mathsf{T}) \end{array}$$

3. Pour finalement prendre la valeur de X dans $M_1 : \delta_1(X)$

Les inconvénients d'une telle méthode est de nécessiter la construction complète du SEB et le risque de calculer des informations "inutiles" pour la valeur finale du calcul. ■

Définition 2.8 (Résolution locale)
Soit un SEB $B = (X, M_1, \ldots, M_n)$ d'alternance 1. Le calcul d'une variable d'un bloc se fait généralement selon l'algorithme suivant :

1. Une routine de résolution R_i est associée à M_i,
2. $R_i(X_j)$ calcule la valeur de X_j dans M_i, en évaluant les parties droites des équations et en effectuant les substitutions par les valeurs calculées,
3. La pile des appels $R_1(X) \rightarrow \ldots \rightarrow R_n(X_n)$ est bornée par la profondeur du graphe de dépendances entre blocs,
4. Chaque résolution R_i garde son contexte, comme dans le cas de coroutine.

Les avantages de cette approche sont qu'elle permet de construire le SEB à la volée, et qu'elle calcule uniquement des informations utiles pour la valeur finale de la variable d'intérêt. ■

Ici, nous nous concentrons sur la résolution locale, parce qu'elle autorise l'exploration du SEB de manière incrémentale et est ainsi adaptée à la vérification à la volée. De plus, les algorithmes locaux sont capables de détecter des erreurs dans des systèmes complexes même si les SEBs correspondants sont trop larges pour être construits explicitement.

Il a été montré dans [And94, VL94] que résoudre localement une variable x_k d'un SEB revenait à trouver un sous graphe (V', E', L') contenant x_k et une fonction totale, nommée *marquage*, $m : V' \rightarrow$ **Bool**, associant une valeur booléenne à chaque sommet de V', qui sont tous deux *stables* (pour chaque $y \in V'$, $m(y)$ correspond à la solution du SEB) et *fermés* (pour chaque $y \in V'$, $m(y)$ ne dépend pas des sommets en dehors de V').

Remarque 2.7
Dans la suite, nous considérons que les graphes booléens $G = (V, E, L)$ sont représentés *implicitement* par leur fonction successeur $E : V \rightarrow 2^V$, qui leur permet d'être construits à la volée par une exploration en avant. ■

Définition 2.9 (Principe des algorithmes de résolution locale)
Un algorithme de résolution locale d'un graphe booléen $G = (V, E, L)$ consiste typiquement en :

- une exploration *en avant* de G en commençant par une variable d'intérêt $x_k \in V$,
- une propagation *en arrière* de variables stables (ou calculées) dont les valeurs sont F dans le cadre d'un bloc de plus grand point fixe,
- la terminaison du calcul lorsque x_k devient stable, ou lorsque toute la portion de graphe accessible depuis x_k a été explorée.

■

Diverses stratégies (par exemple, profondeur d'abord [And94, MS03], largeur d'abord [Mat03a], chaotique [VL94], etc.) peuvent être utilisées, donnant lieu à des algorithmes de résolution avec des complexités linéaires en temps et en espace.

La figure 2.2 montre un SEB avec une sémantique de plus grand point fixe, le graphe booléen correspondant, et un sous graphe (compris dans la surface *gris clair*) calculé par un algorithme de résolution locale (basé sur un parcours en profondeur d'abord (DFS)) à partir du sommet x_1 (les sommets blancs et noirs dénotent les variables T et F, respectivement).

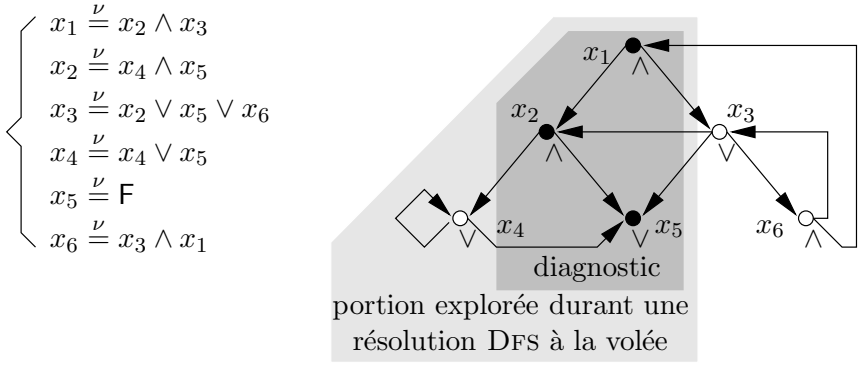


Figure 2.2: Un SEB, son graphe booléen, et le résultat d'une résolution locale pour x_1

Une autre caractéristique des méthodes de résolution locale est la génération de *diagnostics* (portions des SEBs expliquant la valeur de vérité d'une variable, comme celle comprise dans la surface *gris obscur* de la figure 2.2), ce qui fournit une aide considérable pour mettre au point les applications et pour comprendre les formules de logique temporelle [Mat00].

D'un point de vue pratique, contrairement à la situation dans le domaine de la vérification symbolique, pour laquelle un nombre significatif de logiciels basés sur les BDDs sont disponibles (voir [YBO+98] pour un état de l'art, et [KC90, PSC94, SB96, RSBSV96] pour des algorithmes et implémentation de résolution parallèle de BDDs), il n'existe à notre connaissance qu'un seul environnement générique de résolution de SEB disponible pour la vérification à la volée. Il s'agit de CÆSAR_SOLVE [Mat03a], une bibliothèque générique pour la résolution de SEB d'alternance 1 et la génération de diagnostic, créée en utilisant l'environnement OPEN/CÆSAR [Gar98] pour la vérification à la volée. CÆSAR_SOLVE fournit une représentation des SEBs indépendante de l'application, en termes de graphes booléens [And94], de la même manière que OPEN/CÆSAR fournit une représentation des STEs, indépendante du langage. Quatre algorithmes sont actuellement disponibles dans la bibliothèque. Les algorithmes A_1 et A_2 sont généraux (ils ne font aucune supposition à propos des parties droites des équations du SEB), A_2 effectuant un parcours en largeur d'abord et étant optimisé pour produire des diagnostics (exemples et contre-exemples) de faible profondeur. Les algorithmes A_3 et A_4 sont spécialisés pour la résolution en profondeur d'abord efficace en mémoire des SEBs acycliques et des SEBs disjonctifs ou conjonctifs, qui sont rencontrés fréquemment en pratique. La bibliothèque CÆSAR_SOLVE est au cœur de plusieurs outils de vérification à la volée de la boîte à outils CADP [FGK+96] et sert notamment de moteur pour : le comparateur par équivalence BISIMULATOR [Mat03a, BDJM05], qui compare des STEs modulo cinq relations d'équivalence très largement utilisées, l'évaluateur de logique temporelle EVALUATOR pour le μ-calcul modal régulier d'alternance 1 [MS03], et aussi l'outil de réduction par τ-confluence [PLM03].

2.2 Définition des tâches et des données distribuées

Dans la résolution distribuée de SEB, chaque nœud est responsable de l'exploration d'une partie du graphe booléen. La figure 2.3 reprend le SEB donné dans la figure 2.2 et le distribue sur trois nœuds. Les tâches de base de la résolution peuvent être séparées en :

- premièrement, une exploration en avant (également appelée *expansion*) des transitions du graphe booléen (c'est-à-dire, des dépendances entre les variables booléennes) et
- deuxièmement, une propagation en arrière (également appelée *stabilisation*) des sommets

stabilisés (c'est-à-dire, des variables booléennes dont la valeur finale a été calculée).

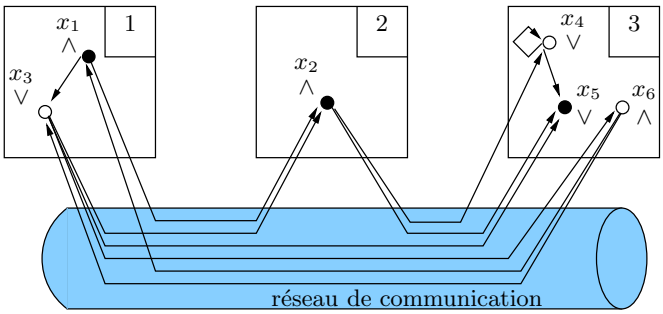


Figure 2.3: Le résultat de la résolution distribuée à la volée de x_1 avec 3 nœuds

Ainsi, les tâches exécutées sur un nœud sont indépendantes et asynchrones par rapport à celles exécutées sur les autres nœuds. Par conséquent, les différentes tâches nécessaires à la résolution des SEBs peuvent être distribuées efficacement. Les données distribuées sont les sommets du graphe booléen (c'est-à-dire, les variables booléennes du SEB). A noter que, le type d'exploration (profondeur d'abord, largeur d'abord, chaotique, ...) du graphe booléen, localement à un nœud, peut affecter globalement le degré de parallélisme de la résolution distribuée de SEBs. En effet, l'ensemble des données (c'est-à-dire, la portion du graphe booléen) traitées par un nœud dépend de l'ordre dans lequel les sommets sont explorés. Il est construit au fur et à mesure de l'exploration globale du graphe booléen. Ainsi, un parcours en largeur d'abord favorisera la construction simultanée sur plusieurs nœuds d'un ensemble de sommets du graphe booléen à explorer, alors qu'un parcours en profondeur d'abord construira et explorera une seule donnée (ou sommet) par nœud (il s'agit de la variable booléenne au sommet de la pile du parcours en profondeur), ce qui résultera globalement à traiter séquentiellement le graphe booléen.

2.3 Algorithme de résolution DSOLVE

Notre algorithme DSOLVE de résolution à la volée de SEBs (voir figure 2.4) prend en entrée un graphe booléen $G = (V, E, L)$ représenté implicitement (c'est-à-dire, par sa fonction successeur), une variable d'intérêt $x \in V$, une fonction statique de hachage $h : V \to [1, P]$, et l'index i du nœud courant. DSOLVE effectue deux parcours entrelacés du graphe booléen en commençant par l'expansion des sommets du graphe booléen et en effectuant aussi tôt que possible la stabilisation de ces sommets. Une fois terminé, l'algorithme retourne la valeur booléenne calculée pour x.

Après une phase d'*initialisation* des structures de données locales (lignes 2-12), la fonction DSOLVE boucle jusqu'à la terminaison de la résolution. Chaque boucle calcule une des trois tâches suivantes, dont la priorité est donnée par les expressions if-then-else en cascade :

- l'*expansion* des variables instables (lignes 27-36),
- la *stabilisation* des variables par propagation arrière (lignes 14-26), et
- la *réception* de messages dénotant des requêtes distantes de stabilisation ou d'expansion (lignes 43-44).

Dans ce qui suit, nous décrivons chacune de ces tâches en détail.

```
 1 function DSolve(x, (V, E, L), h, i) : Bool is
 2   if h(x) = i then /* initialisation */
 3     if L(x) = ∨ then
 4       c(x) := |E(x)|
 5     else
 6       c(x) := 1
 7     endif ;
 8     d(x) := ∅; W_i := {x}; S_i := {x}; B_i := ∅
 9   else
10     W_i := ∅; S_i := ∅; B_i := ∅
11   endif ;
12   term_i := inactive_i := F; sent_i := recv_i := 0;
13   while ¬term_i do
14     if B_i ≠ ∅ then /* stabilisation */
15       while B_i ≠ ∅ do
16         x_i := choose(B_i);
17         B_i := B_i \ {x_i};
18         forall w_i ∈ d(x_i) ∧ c(w_i) ≠ 0 do
19           if h(w_i) = i then
20             Stabilize(w_i, x_i)
21           else
22             Sending(Evl(w_i, x_i), h(w_i))
23           endif
24         endfor ;
25         d(x_i) := ∅
26       endwhile
27     elsif W_i ≠ ∅ then  /* expansion */
28       x_i := choose(W_i);
29       W_i := W_i \ {x_i};
30       forall y_i ∈ E(x_i) do
31         if h(y_i) = i then
32           Expand(x_i, y_i)
33         else
34           Sending(Exp(x_i, y_i), h(y_i))
35         endif
36       endfor
37     else /* reception */
38       if ¬inactive_i then
39         inactive_i := true;
40         sent_i := sent_i + 1;
41         Send(Idl(sent_i − recv_i), superviseur)
42       endif ;
43       Receive(msg_i, sender_i) ;
44       ReadWorker(msg_i, sender_i)
45     endif
46   endwhile ;
47   return c(x) = 0
48 end

49 procedure Sending(msg_j, node_j) is
50   while ¬ISend(msg_j, node_j) ∧¬term_i do
51     if IReceive(msg_i, sender_i) then
52       ReadWorker(msg_i, sender_i)
53     else
54       WaitEvent({0..P}, node_j)
55     endif
56   endwhile ;
57   sent_i := sent_i + 1
58 end

59 procedure Expand(x_i, y_i) is
60   if y_i ∉ S_i then
61     S_i := S_i ∪ {y_i};
62     d(y_i) := ∅;
63     if L(y_i) = ∨ then
64       c(y_i) := |E(y_i)|
65     else
66       c(y_i) := 1
67     endif ;
68     if c(y_i) ≠ 0 then
69       W_i := W_i ∪ {y_i}
70     endif
71   endif ;
72
73   if c(y_i) = 0 then
74     if h(x_i) = i then
75       Stabilize(x_i, y_i)
76     else
77       B_i := B_i ∪ {y_i};
78       d(y_i) := d(y_i) ∪ {x_i}
79     endif
80   else
81     d(y_i) := d(y_i) ∪ {x_i}
82   endif
83 end

84 procedure Stabilize(w_i, y_i) is
85   c(w_i) := c(w_i) − 1;
86   if c(w_i) = 0 then
87     if L(w_i) = ∧ then
88       s(w_i) := y_i
89     endif ;
90     B_i := B_i ∪ {w_i};
91     term_i :=  c(x) = 0
92   endif
93 end

94  procedure ReadWorker(msg_i, sender_i) is
95    recv_i := recv_i + 1 ;
96    if sender_i ≠ superviseur ∧ inactive_i then
97      inactive_i := false;
98      sent_i := sent_i + 1;
99      Send(Act, superviseur)
100   endif ;
101   case msg_i is
102     Evl(x_i, y_i) → if c(x_i) ≠ 0 then
103       Stabilize(x_i, y_i) endif
104     Exp(x_sender_i, y_i) →
105       Expand(x_sender_i, y_i)
106     Ack(stamp) →
107       if inactive_i then
108         sent_i := sent_i + 1;
109         Send(Ack(stamp), superviseur)
110       endif
111     Trm →  term_i := true
112   endcase
113 end
```

Figure 2.4: Résolution distribuée à la volée d'un Seb monobloc en utilisant son graphe booléen

2.3.1 Initialisation

Plusieurs structures de données locales sont nécessaires, soit pour calculer, soit pour mémoriser les variables booléennes et l'information qui leur est attachée. Un compteur $c(y)$ indique le nombre de successeurs instables d'une variable y qui doivent être stabilisés dans le but de stabiliser y lui-même. Dans le contexte d'un SEB monobloc de plus grand point fixe, pour une variable y, $c(y)$ est initialisé avec le nombre de successeurs $|E(y)|$ si y est une $\vee$-variable, ou avec 1 sinon (lignes 3-7). Le compteur $c(y)$ sera plus tard décrémenté chaque fois qu'un successeur sera stabilisé. Une variable y est dite *stable* (ce qui signifie que sa valeur est F dans le cadre d'une équation de plus grand point fixe) lorsque $c(y) = 0$. Un ensemble $d(y) \subseteq V$ (dépendances arrières) contient les variables prédécesseurs de y déjà rencontrées durant les expansions locales et distantes. Cette information sera utilisée lors de la propagation des valeurs des variables stabilisées suivant les dépendances arrières. Trois ensembles sont utilisés par le nœud i pour explorer le graphe booléen :

- l'*ensemble de recherche* $S_i \subseteq V$ stocke toutes les variables booléennes x_i visitées par le nœud i ($h(x_i) = i$),
- l'*ensemble de travail* $W_i \subseteq S_i$ stocke toutes les variables en attente d'être expansées par le nœud i, et
- l'*ensemble de stabilisation arrière* $B_i \subseteq S_i \setminus W_i$ stocke toutes les variables stables à propager par le nœud i le long des dépendances arrières.

Le processus global de résolution est démarré par le nœud *initiateur* dont l'index est $i = h(x)$, qui est responsable de la manipulation de la variable d'intérêt x (lignes 2-8).

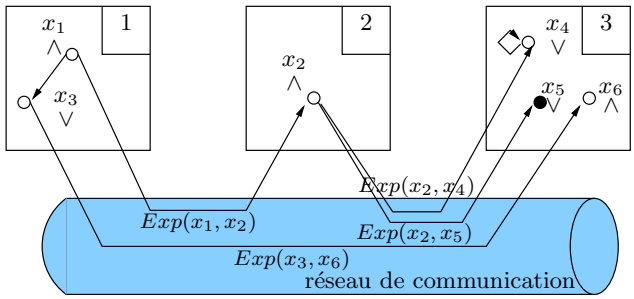
2.3.2 Expansion

L'exploration en avant du graphe booléen consiste en l'expansion des nouvelles variables booléennes locales. Une variable x_i est extraite de l'ensemble de travail W_i (lignes 28-29), qui contient seulement les variables instables ($\forall x_i \in W_i \,.\, c(x_i) \neq 0$). Les transitions sortantes de x_i et les variables successeurs correspondantes y_i sont ensuite calculées (ligne 30). Si la fonction de hachage h détermine que y_i doit être manipulée par un nœud distant d'index $h(y_i)$ (ligne 34), alors un message d'expansion $Exp(x_i, y_i)$ est envoyé à ce nœud. Sinon (ligne 32), les structures de données locales sont mises à jour en appelant la procédure EXPAND(x_i,y_i), dont l'invariant d'appel est : $c(x_i) \neq 0 \wedge x_i \in V \wedge y_i \in V \wedge h(y_i) = i$. Si y_i n'a pas encore été visité, il est ajouté à l'ensemble des variables visitées S_i (lignes 60-71) et son compteur $c(y_i)$ est initialisé comme décrit dans la section 2.3.1 (lignes 63-67). Si y_i est une $\vee$-variable puits ou une variable stabilisée précédemment (lignes 73-80), sa valeur (qui est F) est propagée aux prédécesseurs locaux de y_i en appelant la procédure STABILIZE(x_i, y_i), et aux prédécesseurs distants en mettant à jour l'ensemble B_i de stabilisation arrière ainsi que les dépendances arrières $d(y_i)$ qui seront ensemble utilisés plus tard pour envoyer un message de stabilisation $Evl(x_i, y_i)$. Dans les autres cas, les dépendances arrières $d(y_i)$ sont mises à jour avec la variable x_i (ligne 81).

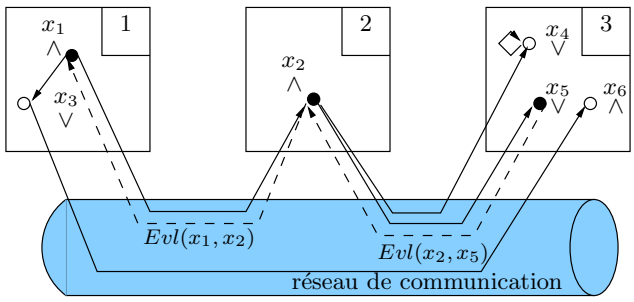
Remarque 2.8
Au lieu de mettre à jour B_i, nous aurions pu envoyer immédiatement un message Evl aux prédécesseurs distants. Seulement, cela aurait créé une possible boucle infinie d'appels d'envoi de messages d'expansion. En effet, dans le cas d'un échec d'émission (ligne 50), une réception est faite (lignes 51-52), qui elle-même peut entraîner l'invocation de la procédure EXPAND (ligne 105) sur réception d'un message Exp (ligne 104), qui en dernier lieu peut initier un envoi de message vers le nœud destinataire avec lequel l'émission a précédemment échoué. Cette considération respecte le modèle de communication que nous avons choisi, dans lequel les émissions et réceptions peuvent échouer. Leur prise en compte impose donc des choix algorithmiques afin d'éviter des situations d'échec en émission comme en réception. ■

2.3.3 Stabilisation

L'exploration arrière du graphe booléen consiste à propager les valeurs des variables stabilisées suivant les dépendances arrières. Pour chaque variable stable $x_i \in B_i$, sa valeur (qui est F dans un SEB monobloc de plus grand point fixe) est propagée à chaque variable prédécesseur $w_i \in d(x_i)$ (lignes 14-26). Si w_i est stocké sur une machine distante, un message de stabilisation $Evl(w_i, x_i)$ est envoyé au nœud d'index $h(w_i)$ (ligne 22). Sinon, les structures de données locales sont mises à jour en appelant la procédure STABILIZE(w_i, x_i), dont l'invariant d'appel est : $c(w_i) \neq 0 \wedge w_i \in d(x_i) \wedge w_i \in S_i$. Le compteur de successeurs instables $c(w_i)$ est décrémenté (ligne 85) et, si w_i devient stable, il est ajouté à l'ensemble des variables stables B_i (ligne 90). Dans le cas où le successeur y_i a stabilisée w_i directement (cas de la propagation d'une variable stabilisée à F vers une $\wedge$-variable dans le cadre d'un plus grand point fixe), nous le mémorisons dans $s(w_i)$ (ligne 88) afin de pouvoir générer un diagnostic à l'issue de la résolution. Une $\vee$-variable stabilisée (à F dans le cadre d'un plus grand point fixe) ne nécessite pas de telle mémorisation d'information, puisque tous ses successeurs doivent être au préalable stabilisés afin que cette dernière le soit également. Finalement, la stabilisation élimine les dépendances arrières entre les variables booléennes (ligne 25) pour réduire la consommation mémoire.



(a)



(b)

Figure 2.5: Un exemple de résolution distribuée à la volée d'un SEB monobloc en utilisant trois nœuds. (a) Initialisation et expansion ; (b) Propagation en arrière des variables stabilisées.

2.3.4 Réception

Symétriquement à la transmission de messages durant la stabilisation et l'expansion, il existe une activité qui gère la réception de messages (lignes 94-113). Selon le type du message (ligne 101), la transition reçue est traitée par la procédure STABILIZE (ligne 103) ou EXPAND (ligne 105). En considérant les échecs d'émission et de réception, ainsi que les tampons de communication pleins, et en introduisant à la fois des primitives de communication bloquantes (lignes 43 et 52) et non bloquantes (ligne 51), l'algorithme DSOLVE devient légèrement plus compliqué. Cependant, cela permet un contrôle de flux à grain fin de la communication, et cela réduit la consommation mémoire liée aux tampons d'émission et de réception.

2.3.5 Exécution de DSOLVE

Un exemple d'exécution de DSOLVE est présenté sur la figure 2.5. Elle illustre, par deux prises de vue de l'exécution, comment DSOLVE distribue et résout sur trois nœuds le SEB indiqué précédemment sur la figure 2.3. La figure 2.5(a) montre l'initialisation de la résolution du SEB, ainsi que l'expansion locale et distante (messages *Exp*) selon un parcours en largeur d'abord du graphe booléen en commençant par x_1, jusqu'aux variables constantes ou circuits. La figure 2.5(b) montre la propagation arrière locale et distante (messages *Evl*) des variables stabilisées (c'est-à-dire, dont la valeur finale a été déterminée) jusqu'à la variable d'intérêt x_1. La résolution termine lorsque x_1 est stabilisée. x_1 a alors pour valeur finale F (sommet de couleur noir) car cette variable a été stabilisée explicitement par une des variables la définissant, et qu'elle appartient à une équation de plus grand point fixe.

2.3.6 Estimation de la complexité

Notre algorithme distribué est basé sur la théorie des graphes booléens sur laquelle reposent également les algorithmes séquentiels [And94, VL94]. Il est composé de deux parcours entrelacés du graphe booléen (en avant et en arrière), dont la complexité en temps dans le pire des cas est $O(|V|+|E|)$. Les mêmes bornes de complexité s'appliquent à la complexité en mémoire, puisque les dépendances $d(y)$ sont stockées durant l'exploration de graphe. La complexité en messages est $O(|E|)$, le pire des cas étant obtenu avec deux messages (expansion et stabilisation) échangés par transition.

2.4 Agent central de coordination et de terminaison

2.4.1 Tâches de coordination

En plus des P processus (un par nœud), également appelés *travailleurs*, réalisant la résolution distribuée du SEB, nous introduisons un processus *superviseur* spécial d'index 0, qui fournit une interface utilisateur pour la résolution du SEB. Parmi ses activités, le superviseur a la charge de mettre en place l'environnement distribué, de lancer les P travailleurs, de tracer et d'afficher la progression de la résolution du SEB, de collecter des statistiques sur la structure du SEB, et également de gérer les terminaisons anticipées requises par l'utilisateur ou les terminaisons urgentes causées par des défaillances distantes de matériel ou de logiciel. Bien que l'algorithme définissant le comportement du nœud superviseur (décrit dans la figure 2.6) soit indépendant de la résolution du SEB (invariant préservé : $\nexists\, x \in V \mid h(x) = 0$), le superviseur est dans une position privilégiée, puisqu'il a une vue globale du calcul distribué. En particulier, cela permet la détection distribuée de terminaison (DDT)

```
114 :procedure SUPERVISOR is
115 :trm_status := DETECT;
116 :sent := recv := 0;
117 :stamp := 0;
118 :total_msg := nb_idle := nb_ack := 0;
119 :forall i in [1..P] do
120 :   nb_msg(i) := 0
121 :endfor ;
122 :while trm_status ≠ TERM do
123 :   case trm_status is
124 :   CONF → while bcast_node ≤ P ∧
125 :      ISEND(Ack(stamp), bcast_node) do
126 :         bcast_node := bcast_node + 1;
127 :         sent := sent + 1
128 :      endwhile
129 :   STOP → while bcast_node ≤ P ∧
130 :      ISEND(Trm, bcast_node) do
131 :         bcast_node := bcast_node + 1
132 :      endwhile ;
133 :      if bcast_node > P then
134 :         trm_status := TERM
135 :      endif
136 :   endcase ;
137 :   if trm_status = DETECT then
138 :      RECEIVE(msg, sender) ;
139 :      READSUPERVISOR(msg, sender)
140 :   elsif IRECEIVE(msg, sender) then
141 :      READSUPERVISOR(msg, sender)
142 :   endif
143 :endwhile
144 :end
```

```
145 :procedure READSUPERVISOR(m, s) is
146 :recv := recv + 1;
147 :case m is
148 :   Act → nb_idle := nb_idle − 1;
149 :      total_msg := total_msg − nb_msg(s);
150 :      if trm_status = CONF then
151 :         trm_status := DETECT
152 :      endif
153 :   Idl(k) → nb_msg(s) := k ;
154 :      nb_idle := nb_idle + 1;
155 :      total_msg := total_msg + nb_msg(s);
156 :      if total_msg = −(sent − recv)
157 :      ∧ nb_idle = P then
158 :         trm_status := CONF;
159 :         bcast_node := 1;  nb_ack := 0;
160 :         stamp := stamp + 1
161 :      endif
162 :   Ack(s) → if s = stamp then
163 :         if trm_status = DETECT then
164 :            if total_msg = −(sent − recv)
165 :            ∧ nb_idle = P then
166 :               trm_status := CONF;
167 :               bcast_node := 1;  nb_ack := 0;
168 :               stamp := stamp + 1
169 :            endif
170 :         elsif trm_status = CONF then
171 :            nb_ack := nb_ack + 1;
172 :            if total_msg = −(sent − recv)
173 :            ∧ nb_ack = P then
174 :               trm_status := STOP;
175 :               bcast_node := 1
176 :            endif
177 :         endif
178 :      endif
179 :endcase
180 :end
```

Figure 2.6: Algorithme de détection de terminaison par le nœud superviseur

de la résolution du SEB en utilisant autant de messages que les mécanismes traditionnels asymétriques de DDT [MC98].

2.4.2 Détection de terminaison distribuée

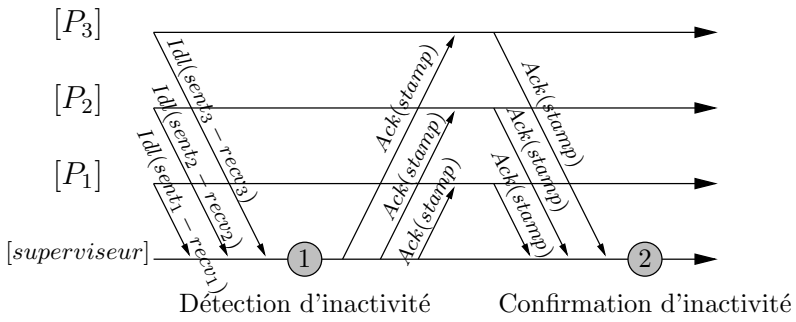
La résolution distribuée du SEB est terminée lorsque la variable d'intérêt x a été explicitement stabilisée ($c(x) = 0$ à la ligne 91) durant la propagation arrière de variables stables, ou bien lorsque le graphe booléen a complètement été exploré, c'est-à-dire, lorsque tous les ensembles de variables de travail sont vides ($\forall i \in [1..P].W_i = B_i = \emptyset$), et que plus aucun message ne transite à travers le réseau. La première condition est détectée par le nœud travailleur initiateur, dont l'index est $h(x)$, lors de la propagation arrière des valeurs booléennes jusqu'à x (ligne 91). La seconde condition requiert un algorithme de DDT. Dans les deux cas, la variable booléenne $term_i$ est positionnée à T.

Notre algorithme de terminaison est basé sur deux vagues de détection d'inactivité globale, entre le nœud superviseur et les processus de résolution. Nous avons utilisé un algorithme dérivé d'une combinaison des algorithmes de DDT de [HK91] (utilisation d'un agent central selon une topologie en

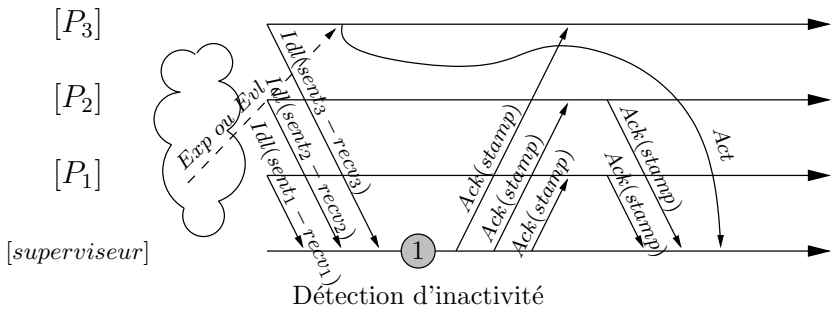
étoile centrée sur l'agent) et [Mat87] (utilisation de la technique des quatre compteurs et des deux vagues de détection d'inactivité). Il repose sur le nœud superviseur (d'index *superviseur*=0), qui est habituellement le nœud depuis lequel l'utilisateur lance la résolution distribuée du SEB.

La DDT est composée des deux vagues de détection suivantes :

1. Détection de l'inactivité globale par le nœud superviseur (c'est-à-dire, $trm\_status=DETECT$) et confirmation de l'inactivité locale par chacun des nœuds travailleurs et
2. Détection de la confirmation de l'inactivité globale confirmée par chacun des nœuds au superviseur (c'est-à-dire, $trm\_status=CONF$).



(a)



(b)

Figure 2.7: Algorithme de détection distribuée de la terminaison. (a) Tentative de DDT avec succès ; (b) Tentative de DDT avec échec à cause de l'occurrence d'un message d'activité.

Sur chaque travailleur, ainsi que sur le superviseur, deux compteurs $sent_i$ (ou $sent$) et $recv_i$ (ou $recv$) gardent le nombre de messages échangés en émission et en réception. Lorsqu'un travailleur i devient inactif, il envoie un message $Idl(sent_i - recv_i)$ au superviseur (lignes 38-42). Lorsqu'il redevient actif, le travailleur envoie un message *Act* au superviseur (lignes 96-100). Le superviseur retient pour chaque travailleur la quantité de messages émis moins ceux reçus ($nb\_msg(i)$, ligne 153), ainsi que le nombre de travailleurs qui ont manifesté leur état d'activité (nb_idle, lignes 148 et 154). Par conséquent, lorsque le superviseur détecte que tous les travailleurs sont inactifs (c'est-à-dire, $\forall i \in [1..P].inactive_i=\mathsf{T}$ et $nb\_idle=P$), il vérifie également si aucun message n'est encore en transit (c'est-à-dire, $total\_msg=\sum_{i=1}^{P}(sent_i - recv_i)$ et $total\_msg+(sent-recv)=0$). Si les deux conditions sont respectées (lignes 156-157), une vague de confirmation d'inactivité, indexée par le compteur

stamp, est démarrée. Le superviseur diffuse à tous les travailleurs un message $Ack(stamp)$ (lignes 124-128), purgeant ainsi de possibles messages résiduels qui transitent encore entre les travailleurs et le superviseur. Chaque travailleur inactif accuse la réception d'un message $Ack(stamp)$ en renvoyant le même message $Ack(stamp)$ au superviseur (lignes 106-110). Si un travailleur est actif lors de la réception d'un message $Ack(stamp)$, il ignore simplement le message. Dans ce cas, un message *Act* de ce nœud devra par la suite arriver au superviseur. Finalement, le superviseur détecte la terminaison globale s'il reçoit P messages de type $Ack(stamp)$ (c'est-à-dire, $nb\_ack = P$, lignes 162-178). Il peut alors diffuser la détection de terminaison (c'est-à-dire, $trm\_status{=}STOP$) à tous les travailleurs (lignes 129-135).

La figure 2.7 présente deux prises de vue reflétant deux états dans lequel le système distribué peut être lorsque le nœud P_1 initialise l'algorithme de DDT.

Puisque n'importe quel processus travailleur peut initier l'algorithme de DDT, la construction de plusieurs prises de vue peut être faite au même moment. Au point d'observation 1, le superviseur détecte un état global dans lequel plus aucun message n'est en transit, et plus aucun processus n'a de données locales à calculer. La résolution distribuée semble être entrée dans un état dans lequel plus aucune progression ne peut être faite. En analysant un tel état global, il peut être conclu que nous sommes confrontés à un interblocage, ou bien que la résolution distribuée du SEB a été correctement terminée.

Après la phase de confirmation au point d'observation 2 dans la figure 2.7(a), le superviseur peut conclure que le calcul distribué est terminé correctement puisque plus aucune activité n'a été détectée entre les points d'observation 1 et 2. Par contre, la terminaison ne peut être détectée dans la prise de vue montrée dans la figure 2.7(b), car il y a un message *Act* en transit entre le nœud P_3 (qui est redevenu actif entre-temps après avoir reçu un message de travail *Exp* ou *Evl*) et le superviseur.

Remarque 2.9
Théoriquement notre algorithme de détection de terminaison a une complexité $O(|E|)$, avec au plus une DDT démarrée pour chaque message échangé entre travailleurs et au plus un message par transition du graphe booléen. Néanmoins, il se révèle très efficace en pratique avec seulement 0.01% du nombre total de messages échangés utilisé pour la détection de terminaison. Ceci est dû au fait que le superviseur ne démarre la DDT que lorsqu'il a une vue de l'état d'inactivité globale de la résolution qui est très proche de l'état de terminaison. ■

2.5 Implémentation

2.5.1 Algorithme DSOLVE

Nous avons implémenté DSOLVE (10 000 lignes de code C) dans l'environnement OPEN/CÆSAR. Notre algorithme est un composant à part entière de la bibliothèque de résolution de SEBS CÆSAR_SOLVE [Mat03a], qui dispose d'une documentation détaillée dans la distribution CADP. L'interface d'utilisation de DSOLVE est conforme à celle définie dans cette bibliothèque. Ainsi, DSOLVE peut être utilisé immédiatement par tous les outils de vérification basés sur CÆSAR_SOLVE (comme par exemple EVALUATOR [MS03], BISIMULATOR [BDJM05] et TAU_CONFLUENCE [PLM03]). Enfin, notre implémentation repose sur la bibliothèque de communication CÆSAR_NETWORK, qui permet d'assurer la couche basse (écriture/lecture des sockets TCP/IP) des transmissions et réceptions de messages.

2.5.2 Générateur de SEBs monoblocs aléatoires

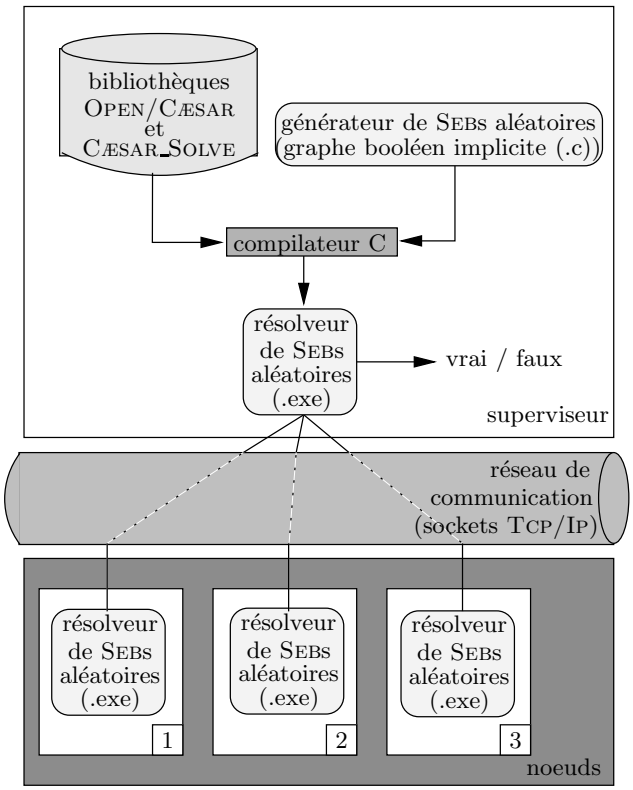


Figure 2.8: L'outil de génération et de résolution aléatoire de SEBs monoblocs

Dans le but de tester notre implémentation de DSOLVE, nous avons développé un générateur de SEBs aléatoires (1000 lignes de code C) qui produit un SEB implicite (au moyen d'une fonction successeur qui fournit les transitions sortant d'une variable dans le graphe booléen) aléatoirement. La figure 2.8 montre l'interface de programmation et l'environnement d'exécution du solveur de SEBs obtenu en connectant le générateur de SEBs aléatoires et l'algorithme DSOLVE.

C'est le rôle du superviseur de dupliquer et de lancer les programmes exécutables sur tous les nœuds. Le solveur de SEBs aléatoires fournit un moyen très précis de mesurer les performances de DSOLVE, puisque le coût du calcul de sa fonction successeur est négligeable par rapport à la résolution distribuée du SEB. De plus, un réglage approprié des paramètres permet de construire une grande variété de SEBs incluant ceux rencontrés dans des domaines d'application spécifiques, tels que la vérification (comparaison par équivalences, évaluation de propriétés logiques, réduction d'ordre partiel) ou la résolution de clauses de Horn. Les paramètres utilisés pour la définition d'une classe de SEBs sont : le pourcentage d'alternance de type de variable, c'est-à-dire, la proportion de variables $\wedge$ (*resp.* $\vee$) succédant à une $\vee$-variable (*resp.* $\wedge$) ; le pourcentage de constantes booléennes dans le SEB ; le nombre minimal et maximal de variables ; et la longueur moyenne d'une équation booléenne (facteur de branchement du graphe booléen). Les variables booléennes (représentées par des entiers) et leurs types ($\wedge$, $\vee$) sont générés aléatoirement au moyen d'une racine également donnée en paramètre à la génération du SEB.

2.6 Analyse de performances

2.6.1 Architecture de la plate-forme de tests

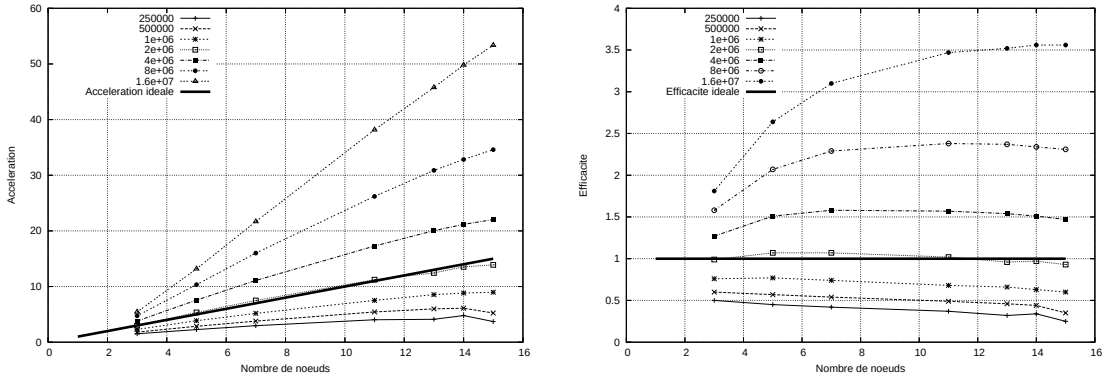
Pour utiliser efficacement la mémoire (étant donné que la charge de travail est distribuée également parmi les nœuds par la fonction de hachage statique), nous supposons que tous les nœuds sont homogènes en termes de système d'exploitation, de processeur et de mémoire. Nos expérimentations ont principalement été menées sur une grappe de 15 PCs XEON 2.4 GHz avec 1.5 GO de RAM, exécutant LINUX, et interconnectés par un réseau GIGABIT. Nous avons également fait quelques expériences sur une grappe de 216 PCs PENTIUM III 733 MHz, avec 256 MO de RAM, exécutant LINUX, essentiellement pour des mesures de passage à l'échelle. Enfin, nous avons utilisé un réseau local de stations de travail SUN lors du développement.

2.6.2 Accélération et efficacité

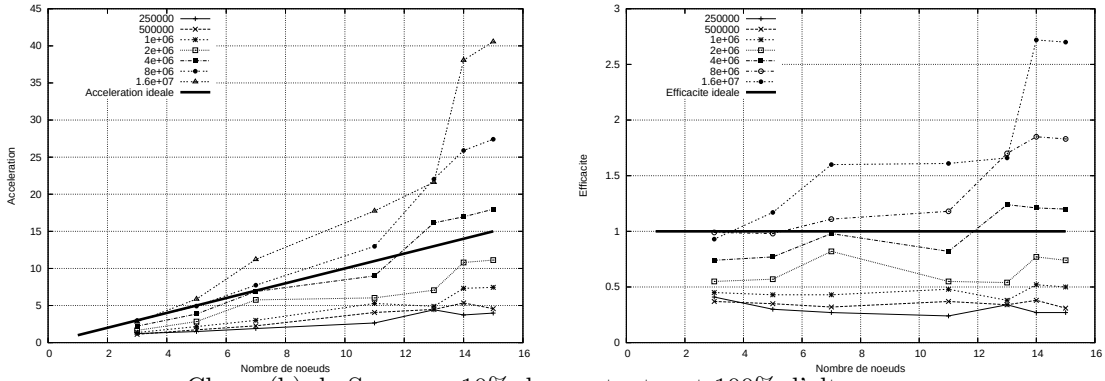
Pour obtenir le temps (T_s) du meilleur algorithme séquentiel pour résoudre le problème, nous utilisons l'algorithme de [Mat03a] basé sur un parcours en largeur d'abord. Globalement, le comportement de l'algorithme séquentiel est proche de celui de DSOLVE avec $P = 1$ qui est également basé sur un parcours en largeur d'abord.

La figure 2.9 montre l'accélération et l'efficacité obtenues pour trois différentes classes de SEBs. L'accélération et l'efficacité sont calculées sur la base de temps réel de résolution distribuée, c'est-à-dire, en excluant la phase de lancement de coût fixe et dépendante du système d'exploitation (copies de fichiers, mise en place des canaux de communication, création de structures de données). Un des nœuds est également utilisé pour exécuter le processus superviseur en plus du processus travailleur. Les résultats montrés dans la figure 2.9 ont été obtenus avec des équations booléennes dont la partie droite contenait 10 variables en moyenne, et une taille de SEB à résoudre entre $25 \cdot 10^4$ et $1.6 \cdot 10^7$ variables. Il est à noter que pour chaque point sur chaque courbe, la résolution de SEB a été exécutée dix fois. Les courbes de performances, comme l'accélération dans la figure 2.9, mais également de passage à l'échelle dans la figure 2.10 (colonne gauche) et de consommation mémoire dans la figure 2.10 (colonne droite), montrent la moyenne de chacun des ensembles de dix exécutions en excluant la valeur minimale et la valeur maximale.

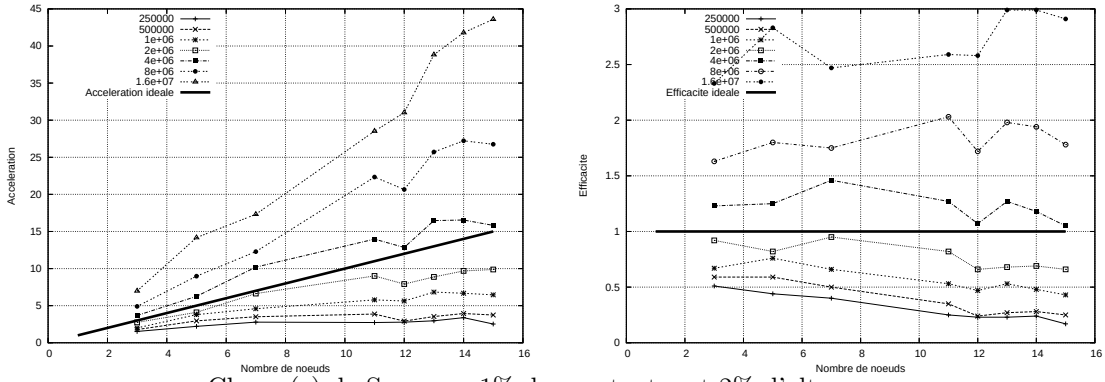
La figure 2.9(a) représente la classe de SEBs sur laquelle DSOLVE réalise les meilleures accélération et efficacité. Cette classe est caractérisée par 0% d'alternance de type de variables et 0% de constantes booléennes. Cette classe de SEBs correspond à la vérification par équivalences (comparaison d'automates) lorsqu'un automate est déterministe (pour l'équivalence forte) et ne comporte pas de transitions invisibles (pour les équivalences faibles) [Mat03a] et les deux automates sont équivalents. Puisqu'aucune constante n'est présente dans le SEB, aucune stabilisation ne survient, et la résolution du SEB est réduite à l'exploration en avant du graphe booléen dans sa totalité. Chaque courbe sur la figure 2.9(a) donne l'accélération (colonne gauche), et l'efficacité (colonne droite), pour un problème donné, et en utilisant un nombre grandissant de nœuds XEON (de 3 à 15). Il peut être observé que l'accélération et l'efficacité augmentent avec la taille du SEB et deviennent super linéaires pour des problèmes de plus de $2 \cdot 10^6$ variables. Ceci peut s'expliquer par le fait qu'à la fois DSOLVE et l'algorithme séquentiel dans [Mat03a] utilisent des tables de hachage (fournies par l'environnement OPEN/CÆSAR) pour stocker les ensembles de variables booléennes. Ces dernières ont une complexité en nombre de recherches $O(N/H)$, où H est le nombre d'entrées (listes de collisions) de la table de hachage et N est le nombre de variables présentes dans la table. A partir de là, mettre à jour les tables de hachage durant l'exécution de DSOLVE prend $(|E|/P) \cdot ((|V|/P)/H) = (|E| \cdot (|V|/H))/P^2$ opérations, en supposant que l'ensemble des dépendances E a été divisé également en P sous en-



Classe (a) de SEB avec 0% de constantes et 0% d'alternance



Classe (b) de SEB avec 10% de constantes et 100% d'alternance



Classe (c) de SEB avec 1% de constantes et 2% d'alternance

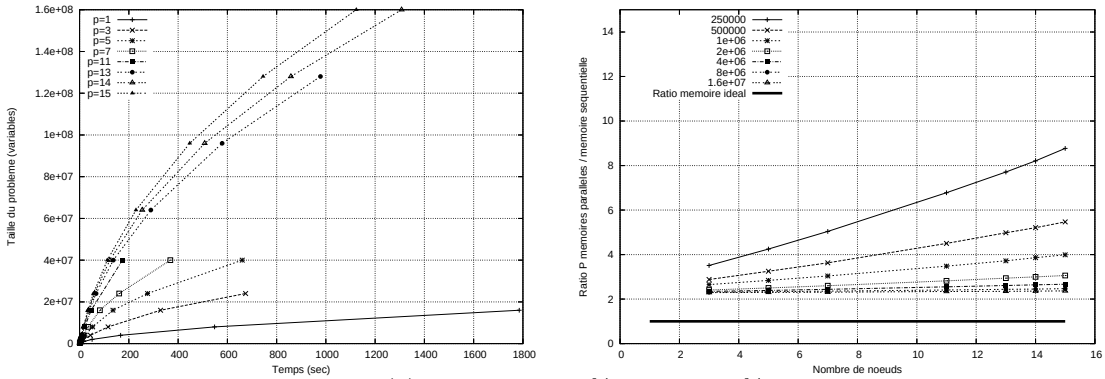
Figure 2.9: Accélération (colonne gauche) et efficacité (colonne droite) pour trois classes de SEBs monoblocs

sembles, alors qu'elle prend $|E| \cdot (|V|/H)$ opérations dans l'algorithme séquentiel. C'est pourquoi, pour des SEBs de grande taille (c'est-à-dire, $|V| >> H$, par exemple, $6 \cdot 10^6$ variables) et pour un grand nombre de nœuds (c'est-à-dire, division par un grand facteur P^2), nos expériences peuvent donner des résultats superlinéaires. Le coût de la mise à jour des tables de hachage pendant la résolution séquentielle devient plus important que le surcoût lié à la communication et la gestion de tampons de la résolution distribuée. Nous pensons qu'un comportement similaire apparaîtrait avec d'autres structures de données (par exemple, arbres équilibrés de recherche) utilisées pour gérer les ensembles de variables booléennes.

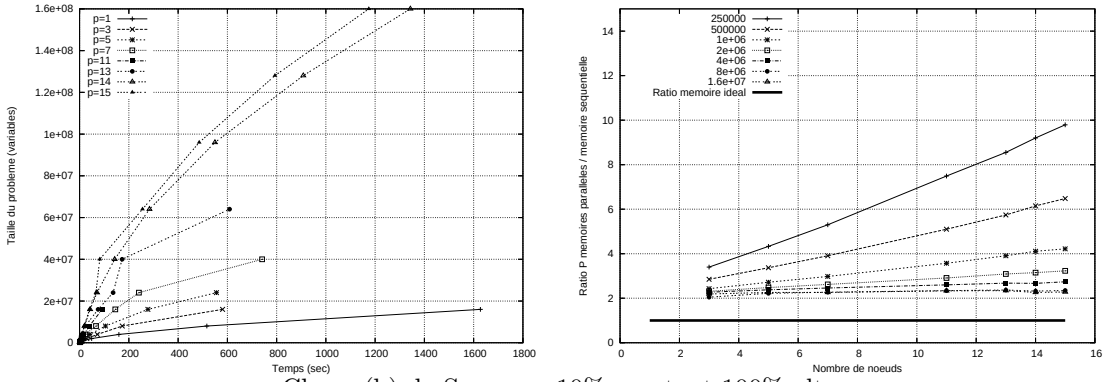
La figure 2.9(b) représente une classe de SEBs avec 100% d'alternance de type de variable (c'est-à-dire, chaque $\vee$-variable n'a que des $\wedge$-successeurs, et vice-versa), et 10% de constantes booléennes. Cette classe correspond à la vérification de systèmes non déterministes (comparaison par équivalence [Mat03a] et réduction d'ordre partiel [PLM03]), et à la résolution de clauses de Horn [LS98]. L'accélération et l'efficacité sont légèrement moins bonnes que pour la figure 2.9(a), puisque les constantes booléennes provoquent de la stabilisation (et donc des messages additionnels) durant la résolution du SEB. Ainsi, le coût de communication global dans la résolution distribuée est doublé, puisque chaque message d'expansion entraînera très probablement un message de stabilisation. Cependant, la propagation arrière des variables stables est limitée à cause du paramètre d'alternance de 100%, qui ne permet de propager les variables stabilisées que sur les prédécesseurs immédiats (par exemple, une $\wedge$-variable stabilisée à F ne stabilisera pas forcément ses $\vee$-prédécesseurs). Par conséquent, aussi bien la résolution séquentielle que la résolution distribuée doivent explorer la plupart du SEB, et la résolution séquentielle est une fois de plus pénalisée lorsque les SEBs deviennent très larges (l'accélération et l'efficacité deviennent super linéaires pour des SEBs excédant $4 \cdot 10^6$ variables).

La figure 2.9(c) montre la classe de SEBs avec les résultats les plus faibles en accélération et efficacité. Le graphe correspond aux SEBs avec une alternance de type de variable de 2% et 1% de constantes booléennes. De tels SEBs, contenant de longs chemins de variables $\wedge$ terminés par des constantes F (variables $\vee$ puits), sont souvent rencontrés en vérification (comparaison par équivalence d'automates déterministes et évaluation de formules logiques [Mat03a]). La propagation arrière de variables stables peut être très rapide et atteint souvent la variable d'intérêt. L'algorithme séquentiel implémente un mécanisme de propagation efficace (puisque toute l'information sur les dépendances prédécesseurs est stockée localement), alors que la communication par messages de stabilisation dans DSOLVE (qui sont aussi nombreux que les messages d'expansion) ne peut pas être recouverte complètement par les activités de calcul. Cet inconvénient est à peine visible lorsque peu de nœuds sont utilisés, mais devient important lorsque de nombreux nœuds sont utilisés sur de petits SEBs. Ceci explique la diminution plus marquée de l'efficacité de DSOLVE avec cette classe de SEBs comparée aux classes de SEBs précédentes. Comme pour la classe de SEBs précédente, l'accélération et l'efficacité deviennent super linéaires pour des SEBs excédant $4 \cdot 10^6$ variables. Nous observons le même comportement pour des SEBs caractérisés par un pourcentage d'alternance de type de variable compris dans $]0, 100[$ et un pourcentage de constantes booléennes dans $]0, 100[$.

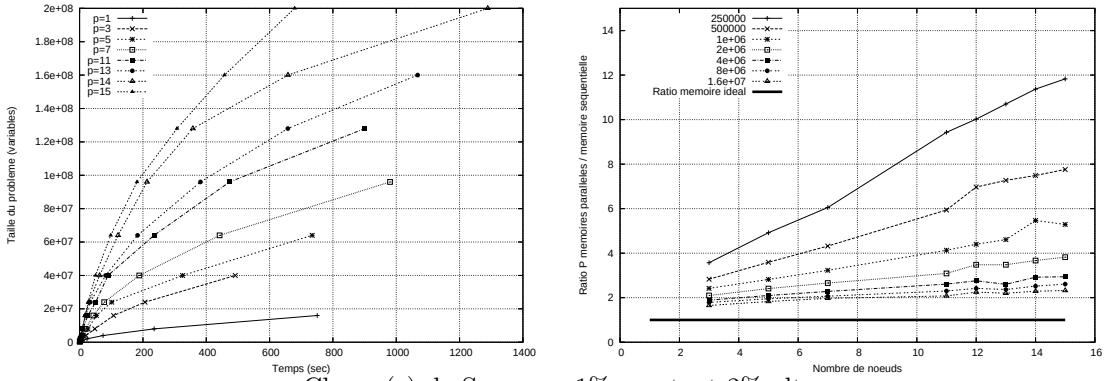
Plus généralement, pour toutes classes de SEBs, nous observons que plus grand est le SEB, meilleures sont l'accélération et l'efficacité réalisées par DSOLVE. Ceci est également illustré par tous les six graphes de la figure 2.9. Une autre information donnée par les courbes d'efficacité est le compromis entre ajouter plus de nœuds pour le calcul et obtenir une meilleure accélération. Ainsi, l'efficacité est également un indicateur du bon passage à l'échelle de DSOLVE, puisque la plupart des courbes ont une asymptote horizontale.



Classe (a) de SEB avec 0% const. et 0% alt.



Classe (b) de SEB avec 10% const. et 100% alt.



Classe (c) de SEB avec 1% const. et 2% alt.

Figure 2.10: Passage à l'échelle en termes de tailles de SEBs monoblocs et de nombres de nœuds (colonne gauche), et surcoût en mémoire (colonne droite) pour trois classes de SEBs monoblocs

2.6.3 Passage à l'échelle

Le passage à l'échelle peut être exprimé en termes de variation de la vitesse de traitement (en augmentant la taille du SEB sur un nombre fixe de nœuds), ou en termes de temps d'exécution (en augmentant le nombre de nœuds sur une taille fixe de SEB).

La figure 2.10 (colonne gauche) présente les deux types de passage à l'échelle obtenus pour les trois différentes classes de SEBs présentées dans la section 2.6.2. Chaque courbe représente la vitesse de traitement des variables booléennes avec un ensemble fixe de nœuds XEON variant entre 1 et 15. Chaque point des courbes est une instance de résolution pour une taille de SEB entre $25 \cdot 10^4$ à $2 \cdot 10^8$ variables (axe y), et dont le temps d'exécution varie de 0.1 à 1 800 secondes (axe x). Les points sur une même ligne horizontale représentent les instances de résolution d'un même SEB.

La forme des courbes pour des SEBs de grande taille est proche du linéaire (elle ne converge pas immédiatement vers une asymptote horizontale) montrant ainsi que la vitesse de traitement passe bien à l'échelle avec le nombre de nœuds utilisés. En augmentant le nombre de nœuds sur une taille fixe de SEB, on peut même obtenir des gains super linéaires (voir la discussion de la figure 2.9). Par exemple, la résolution d'un SEB de 100 millions de variables sera plus que deux fois plus rapide en utilisant 14 nœuds qu'avec 7 nœuds. Nous observons par ailleurs que DSOLVE a une vitesse de traitement plus élevée sur la classe (c) de SEBs que sur les classes (a) et (b). Un exemple est la résolution d'un SEB de 128 millions de variables, 700 millions de transitions sur 14 nœuds XEON en à peu près 6 minutes pour la classe (c) de SEBs, alors que cette résolution prend environ 15 minutes pour les classes (a) et (b). En effet, les SEBs de la classe (c) contiennent de longs chemins de variables $\vee$ qui peuvent être stabilisés sans requérir l'exploration de tous leurs successeurs. Ainsi, de nombreux calculs d'exploration et de stabilisation sont évités. En fait, seulement 327 millions de transitions ont été réellement explorées dans la résolution du SEB de classe (c), alors que 700 millions de transitions le furent pour les classes (a) et (b). Avec la classe (b), la stabilisation est réalisée, mais elle est strictement limitée aux prédécesseurs à un pas. Ainsi, la plupart des transitions doivent être explorées également, et la stabilisation apporte un léger surcoût en communication.

Remarques 2.1
Les mêmes SEBs ont été résolus avec 81 nœuds PENTIUM, et les résultats obtenus en temps d'exécution et en nombre de messages échangés sont similaires à ceux exposés dans cette section. ■

Du fait de restrictions d'implémentation (dont la correction est en cours), l'algorithme séquentiel ne peut pas traiter plus de $1.6 \cdot 10^7$ variables sur un seul nœud. C'est pourquoi les courbes de référence présentées pour la version séquentielle ne dépassent pas des tailles de SEBs plus grandes que $1.6 \cdot 10^7$ variables. ■

2.6.4 Coût en mémoire et en communication

Toutes nos expériences montrent un équilibrage de charge parfait réalisé par la fonction de hachage : pendant la résolution, l'ensemble des variables booléennes explorées est divisé également parmi les P nœuds. La figure 2.10 (colonne droite) présente le surcoût en mémoire de DSOLVE par rapport à la consommation mémoire de la version séquentielle. Des expériences ont été menées sur les trois classes de SEBs étudiés précédemment dans les sections 2.6.2 et 2.6.3. Nous prenons en compte seulement la mémoire utilisée par la résolution du SEB (ce qui inclut les tables de hachage utilisées pour stocker les variables booléennes) et par la bibliothèque de communication dans le cas de l'algorithme DSOLVE (ce qui inclut les différents tampons de communication). Nous pouvons observer que le surcoût mémoire dû à la distribution de la résolution est insignifiant comparé à la mémoire utilisée pour les données : si M_s est le coût mémoire séquentiel, alors M_P, le coût mémoire de chacun des

P nœuds, est approximativement M_s/P. Seulement avec de petits SEBs ($\leq$ 500 000 variables), nous obtenons un surcoût mémoire qui grossit avec le nombre de nœuds utilisés. Ceci peut être expliqué par l'importance relative du surcoût mémoire qu'engendre la résolution distribuée par rapport à la taille du problème à résoudre. De manière générale, plus grand est le SEB, plus petit sera le surcoût mémoire lors de l'ajout d'un nœud supplémentaire dans le calcul distribué. Globalement, pour n'importe quelle classe de SEBs, le surcoût mémoire reste constant lorsque l'on augmente le nombre de nœuds.
Le coût moyen en communication peut également être évalué, étant donné que les messages sont envoyés pour chaque dépendance inter nœuds (c'est-à-dire, transitions $(x, y) \in E$ telles que $h(x) \neq h(y)$). Puisque la fonction de hachage h distribue les variables également parmi les nœuds, elle distribue aussi les dépendances de manière égale. Ainsi, le nombre de dépendances inter nœuds peut être évalué à $((P\text{-}1)/P)\cdot|E|$, étant donné que statistiquement seulement $|E|/P$ transitions seront locales à chacun des nœuds. Donc, au plus $((P\text{-}1)/P)\cdot|E|$ messages d'expansion et autant de messages de stabilisation correspondants sont échangés.

Afin d'illustrer le coût réel en mémoire et en communication, nous reprenons ici l'exemple de la section 2.6.3, à savoir la résolution du SEB de 128 millions de variables sur 14 nœuds XEON. Pour les classes (b) et (c), nous avons observé que, grâce à la stabilisation, moins de dépendances doivent être stockées durant la résolution. Ainsi, la résolution prend respectivement pour les classes (a), (b) et (c), approximativement 20.6 GO, 14.7 GO et 11.6 GO de mémoire principale sur l'ensemble des nœuds. Enfin, nous avons évalué le coût en communication des différentes résolutions. Le nombre de messages échangés était d'environ 651 millions pour la classe (a), parmi lesquels 38 000 messages étaient utilisés pour la détection de terminaison. Pour la classe (b), 920 millions de messages ont été échangés parmi lesquels 294 000 pour la détection de terminaison. Enfin, pour la classe (c), 307 millions de messages furent échangés parmi lesquels 100 pour la détection de terminaison. Ce coût relativement élevé de la communication est dû à un manque de localité des données provoquée par la fonction de hachage statique. Ce volume important de données échangées explique en partie la forme des courbes de passage à l'échelle dans la figure 2.10 (colonne gauche) qui sont toutes non linéaires. Il serait intéressant d'étudier l'effet de la stratégie d'exploration du graphe booléen sur le passage à l'échelle de DSOLVE, par exemple en considérant une combinaison de parcours en profondeur d'abord (réalisé localement par chaque nœud) et de parcours distribué en largeur d'abord (réalisé globalement par tous les nœuds). Nous laissons ce sujet pour de futurs travaux.

——— ⋆ ——— ⋆ ——— ⋆ ———

Notre algorithme DSOLVE, permettant la résolution distribuée à la volée de SEBs monoblocs, est donc très prometteur pour les applications pouvant traduire leurs problèmes sous forme de SEBs monoblocs, qui sont nombreuses et couvrent à la fois le domaine de la vérification formelle et celui de la logique propositionnelle.

Dans la suite du document, nous illustrerons le pouvoir d'expression des SEBs dans le cadre de la vérification de systèmes de transitions étiquetées, et le mécanisme de résolution qui leur sont associé. Nous détaillerons en premier lieu les problèmes de comparaison par équivalence et de réduction par τ-confluence qui peuvent être encodés uniquement par des SEBs monoblocs de plus grand point fixe. Nous verrons par la suite les problèmes d'évaluation de propriétés logiques, telles que la propriété de vivacité, qui sont quant à eux représentables par des SEBs contenant plusieurs blocs de plus petit et de plus grand points fixes.

Chapitre 3

Application 1 : Comparaison à la volée par équivalence

Après avoir utilisé avec succès notre algorithme distribué DSolve sur un problème simple (la génération de Sebs aléatoires), il peut être envisagé d'appliquer avec efficacité notre méthode sur des problèmes plus concrets tels que ceux rencontrés dans une démarche de vérification de systèmes. En effet, avant même de mettre en place la connexion du comparateur par équivalence Bisimulator [Mat03a] avec DSolve, nous pouvions décrire la forme et les caractéristiques des Sebs résultants de ce problème de vérification à partir de la version séquentielle de l'outil. Par exemple, nous avons pu déterminer le facteur de branchement moyen du graphe booléen correspondant, ou encore le pourcentage d'alternance entre des $\wedge$-variables et des $\vee$-variables. En utilisant cette connaissance, nous avons testé DSolve sur des Sebs ayant les caractéristiques précises du problème de comparaison par équivalence, en positionnant respectivement les paramètres d'entrée du générateur. Ainsi, au vu des résultats très positifs de performance obtenus, nous avons entrepris de mettre en place la distribution de la comparaison à la volée par équivalence de Bisimulator.

—— ⋆ —— ⋆ —— ⋆ ——

Ce chapitre présente une application de l'algorithme DSolve dans le domaine de la vérification : la comparaison à la volée par équivalence de deux systèmes de transitions étiquetées (Stes) sur un ensemble de machines homogènes de type grappes de Pcs [JM04, BDJM05]. Pour ce problème de vérification, nous définissons dans la section 3.1 les différentes relations d'équivalences de Stes que nous allons utiliser. Nous introduisons ensuite dans la section 3.2 le mécanisme de comparaison à la volée par équivalence. La traduction en termes de Sebs de ce problème de vérification à la volée est présentée dans la section 3.3. Enfin, nous décrivons la connexion de l'outil Bisimulator avec DSolve dans la section 3.4.1 et les gains en performances obtenus de cette manière dans la section 3.4.2.

3.1 Expression des relations d'équivalences comportementales

3.1.1 Définition générale

La comparaison par équivalence consiste à comparer un STE $M_1 = (Q_1, A_1, T_1, q_{01})$ représentant le comportement d'un système (c'est-à-dire, un *protocole*) avec le STE $M_2 = (Q_2, A_2, T_2, q_{02})$ représentant son comportement externe désiré (par exemple, un *service*) modulo une relation d'équivalence appropriée.

Dans un premier temps, nous rappellons quelques définitions et notations générales concernant les relations d'équivalence sur un ensemble d'états Q.

Définition 3.1 (Relation d'équivalence)
Pour tout ensemble Q, une *relation d'équivalence* R définie sur Q est une partie de Q^2 ($R \subseteq Q \times Q$) telle que :

- R est *réflexive* :
$$\forall\ q \in Q\ .\ (q,q) \in R$$
- R est *symétrique* :
$$\forall\ (p,q) \in Q^2\ .\ (p,q) \in R\ \Leftrightarrow\ (q,p) \in R$$
- R est *transitive* :
$$\forall\ (p,q,r) \in Q^3\ .\ ((p,q) \in R \wedge\ (q,r) \in R)\ \Rightarrow\ (p,r) \in R$$

D'autre part, R est une *relation de préordre* si et seulement si R est réflexive et transitive. ■

Remarque 3.1
Dans la suite, pour toute relation binaire R, on emploiera indifféremment les deux notations équivalentes
$$(p,q) \in R \text{ et } pRq$$
■

L'ensemble des relations d'équivalence sur Q^2, muni de l'opérateur d'inclusion $\subseteq$, forme un treillis complet dont les extremums sont respectivement :

- Le *maximum*, qui est la relation *universelle* Q^2 et dans laquelle chaque élément est en relation avec tous les autres ;
- Le *minimum*, qui est la relation *identité* $\{(p,p) \mid p \in Q\}$ et dans laquelle chaque élément est en relation uniquement avec lui-même.

L'ensemble des opérateurs définis sur les relations d'équivalence (c'est-à-dire de 2^Q vers 2^Q) admet respectivement l'intersection $\cap$ et l'union $\cup$ comme borne inférieure et borne supérieure.

On dira qu'une relation R_1 est *plus forte* qu'une relation R_2 si et seulement si R_1 *est contenue* dans R_2 ($R_1 \subseteq R_2$). Clairement, si R_1 est plus forte que R_2, alors deux éléments de Q équivalents modulo R_1 seront équivalents modulo R_2 :
$$R_1 \subseteq R_2\ \Leftrightarrow\ \forall\ (p,q) \in Q^2\ .\ pR_1q\ \Rightarrow\ pR_2q$$

Enfin, pour toute fonction totale, croissante, F de 2^Q vers 2^Q, on notera :

- $\mu\pi.F(\pi)$ le plus petit point fixe de F par rapport à $\subseteq$
- $\nu\pi.F(\pi)$ le plus grand point fixe de F par rapport à $\subseteq$

3.1.2 Relations de bisimulation et de simulation

Van Glabbeek [vG90] a donné en 1990 une classification des équivalences basée sur la distinction entre temps linéaire et temps arborescent pour la description de systèmes parallèles. Ainsi, l'équivalence qui distingue le moins de systèmes est l'*équivalence de trace* [Hoa78] et celles qui en distinguent le plus sont les équivalences dites de *bisimulation* [Par81]. Nous nous intéressons en particulier à cette dernière classe de relations d'équivalence pour plusieurs raisons :

- elles ont une définition suffisamment générale pour pouvoir être étendues à des problèmes spécifiques,
- la sémantique de ces relations d'équivalence est assez intuitive, pour en permettre l'utilisation systématique lors de la définition de spécifications comportementales, et
- enfin, il existe un bon nombre d'algorithmes efficaces de comparaison et de minimisation de STEs modulo ces relations d'équivalence.

Soit Λ un ensemble de langages disjoints sur $\mathcal{A} \cup \{\tau\}$, et soit λ un élément de Λ ($\lambda \subseteq (\mathcal{A} \cup \{\tau\})^*$), une relation de bisimulation est définie comme une famille de relations binaires sur $Q_1 \times Q_2$ paramétrée par Λ :

Définition 3.2 (Equivalence de bisimulation)
Soit l'opérateur $\mathcal{B}_\Lambda \ : \ 2^{Q_1 \times Q_2} \longrightarrow 2^{Q_1 \times Q_2}$ défini par :

$$\begin{aligned}\mathcal{B}_\Lambda(R) \ = \ \{(p_1,p_2) \mid \forall \lambda \in \Lambda \ . \quad & \forall q_1 \ . \ (p_1 \xrightarrow{\lambda}_{T_1} q_1 \Rightarrow \ \exists q_2 \ . \ (p_2 \xrightarrow{\lambda}_{T_2} q_2 \ \wedge \ (q_1,q_2) \in R)) \wedge \\ & \forall q_2 \ . \ (p_2 \xrightarrow{\lambda}_{T_2} q_2 \Rightarrow \ \exists q_1 \ . \ (p_1 \xrightarrow{\lambda}_{T_1} q_1 \ \wedge \ (q_1,q_2) \in R))\}\end{aligned}$$

L'*équivalence de bisimulation* $\sim_\Lambda$ pour le langage Λ est alors définie comme le plus grand point fixe de l'opérateur $\mathcal{B}_\Lambda$ [Par81] :

$$\sim_\Lambda \ = \ \nu R \ . \ \mathcal{B}_\Lambda(R)$$

■

De façon similaire, la relation de simulation est obtenue à partir de l'équivalence de bisimulation en considérant le préordre associé. Elle est donc généralement définie comme une famille de relations sur $Q_1 \times Q_2$ paramétrée par Λ :

Définition 3.3 (Equivalence de simulation)
Soit l'opérateur $\mathcal{I}_\Lambda \ : \ 2^{Q_1 \times Q_2} \longrightarrow 2^{Q_1 \times Q_2}$ défini par :

$$\mathcal{I}_\Lambda(R) \ = \ \{(p_1,p_2) \mid \forall \lambda \in \Lambda \ . \quad \forall q_1 \ . \ (p_1 \xrightarrow{\lambda}_{T_1} q_1 \Rightarrow \ \exists q_2 \ . \ (p_2 \xrightarrow{\lambda}_{T_2} q_2 \ \wedge \ (q_1,q_2) \in R))\}$$

On appelle alors *préordre de simulation* $\sqsubseteq_\Lambda$ pour le langage Λ le plus grand point fixe de l'opérateur $\mathcal{I}_\Lambda$:

$$\sqsubseteq_\Lambda \ = \ \nu R \ . \ \mathcal{I}_\Lambda(R)$$

On appelle *équivalence de simulation*, notée $\approx_\Lambda$, l'intersection de la relation $\sqsubseteq_\Lambda$ et de sa relation inverse :

$$x \approx_\Lambda y \Leftrightarrow x \sqsubseteq_\Lambda y \wedge y \sqsubseteq_\Lambda x$$

■

3.1.3 Equivalences : forte, branchement, observationnelle, $\tau^*.a$, sûreté

Nous introduisons par la suite cinq relations d'équivalence très connues et utilisées dans ce chapitre. Chacune d'entre elles présente un degré d'abstraction différent, est spécifique à une classe de propriétés comportementales et dispose d'algorithmes correspondants performants pour vérifier des systèmes vis-à-vis de ces propriétés.

Définition 3.4 (Equivalence forte [Par81])
L'*équivalence forte* considère comme observable chacune des actions concrètes du système. Cette relation d'équivalence correspond à l'ensemble de langages Λ_f suivant :

$$\Lambda_f \ = \ \{\{b\} \mid b \in \mathcal{A} \cup \{\tau\}\}$$

Soit l'opérateur $\mathcal{B}_f \ : \ 2^{Q_1 \times Q_2} \longrightarrow 2^{Q_1 \times Q_2}$ défini par :

$$\begin{array}{rl} \mathcal{B}_f(R) \ = & \{(p_1,p_2) \mid \forall b \in \mathcal{A} \cup \{\tau\} \ . \\ & \forall q_1 \ . \ (p_1 \xrightarrow{b}_{T_1} q_1 \Rightarrow \ \exists q_2 \ . \ (p_2 \xrightarrow{b}_{T_2} q_2 \ \wedge \ (q_1,q_2) \in R)) \wedge \\ & \forall q_2 \ . \ (p_2 \xrightarrow{b}_{T_2} q_2 \Rightarrow \ \exists q_1 \ . \ (p_1 \xrightarrow{b}_{T_1} q_1 \ \wedge \ (q_1,q_2) \in R))\} \end{array}$$

L'*équivalence forte*, notée $\sim$, pour le langage Λ_f est alors définie comme le plus grand point fixe de l'opérateur $\mathcal{B}_f$:

$$\sim \ = \ \nu R \ . \ \mathcal{B}_f(R)$$

Bien qu'aucune abstraction ne soit effectuée lors de la comparaison, et que cette relation ne soit pas adaptée à la vérification de spécifications comportementales, l'équivalence forte présente un intérêt tout particulier. En effet, elle représente la relation d'"égalité" entre deux STEs, et préserve toutes les propriétés exprimées en logiques temporelles. Elle est l'équivalence comportementale la plus forte. De plus, il existe de nombreux algorithmes efficaces de calcul du quotient d'un STE pour cette relation, ce qui facilite et favorise son utilisation lors de la vérification de systèmes. ■

Les relations d'équivalence, dites faibles, présentées ci-après, sont obtenues à partir de l'équivalence forte en abstrayant une partie de la relation de transition des systèmes considérés. Seules les actions que l'on souhaite prendre en compte sont observables (ou *visibles*). Les autres actions, dites *invisibles* (ou *internes*), sont renommées par l'étiquette spéciale τ. Ainsi, une action observable est une action visible qui peut être précédée ou suivie d'une ou plusieurs actions internes.

Définition 3.5 (Equivalence de branchement [vGW96])
L'*équivalence de branchement* impose une contrainte sur les états atteints à partir desquels un choix non déterministe est possible, pendant l'exécution d'une action observable. Afin de préserver la structure de branchement des processus, un comportement observable ne peut pas contenir d'actions internes à la suite de l'action visible.

Soit l'opérateur $\mathcal{B}_{br} \ : \ 2^{Q_1 \times Q_2} \longrightarrow 2^{Q_1 \times Q_2}$ défini par :

$$\begin{array}{rl} \mathcal{B}_{br}(R) \ = & \{(p_1,p_2) \mid \forall b \in \mathcal{A} \cup \{\tau\} \ . \\ & \forall q_1 \ . \ (p_1 \xrightarrow{b}_{T_1} q_1 \Rightarrow \ (b \ = \ \tau \ \wedge \ (q_1,p_2) \in R) \vee \\ & (\exists q_2 q_2^{'} \ . \ (p_2 \xrightarrow{\tau^*}_{T_2} q_2^{'} \ \wedge \ q_2^{'} \xrightarrow{b}_{T_2} q_2 \ \wedge \ (p_1,q_2^{'}) \in R \ \wedge \ (q_1,q_2) \in R))) \\ & \forall q_2 \ . \ (p_2 \xrightarrow{b}_{T_2} q_2 \Rightarrow \ (b \ = \ \tau \ \wedge \ (p_1,q_2) \in R) \vee \\ & (\exists q_1 q_1^{'} \ . \ (p_1 \xrightarrow{\tau^*}_{T_1} q_1^{'} \ \wedge \ q_1^{'} \xrightarrow{b}_{T_1} q_1 \ \wedge \ (q_1^{'},p_2) \in R \ \wedge \ (q_1,q_2) \in R))) \end{array}$$

L'*équivalence de branchement*, notée $\approx_b$, est définie comme le plus grand point fixe de l'opérateur $\mathcal{B}_{br}$:

$$\approx_b = \nu R \,.\, \mathcal{B}_{br}(R)$$

L'équivalence de branchement est l'une des relations d'équivalence les plus utilisées car elle préserve la structure de branchement, dispose d'algorithmes de comparaison et de minimisation efficaces, et enfin peut remplacer la vérification de système à l'aide d'équivalences plus faibles, comme l'équivalence observationnelle. ■

Définition 3.6 (Equivalence observationnelle [Mil89])
Le critère d'abstraction proposé par Milner pour l'algèbre de processus CCS, est le *critère d'observation*, sur lequel est basée l'*équivalence observationnelle* (ou *équivalence faible*) [Mil80]. Une action observable est alors définie comme une action visible qui peut être précédée ou suivie d'une ou plusieurs actions internes.

Plus formellement, l'équivalence observationnelle, notée $\approx_o$, est définie pour l'ensemble de langages Λ_{ob} suivant :

$$\Lambda_{ob} = \{\tau^*\} \cup \{\tau^* a \tau^* \mid a \in \mathcal{A}\}$$

Soit l'opérateur $\mathcal{B}_{ob} : 2^{Q_1 \times Q_2} \longrightarrow 2^{Q_1 \times Q_2}$ défini par :

$$\begin{array}{rll}
\mathcal{B}_{ob}(R) = & \{(p_1,p_2) \mid & \forall q_1 \,.\, (p_1 \xrightarrow{\tau}_{T_1} q_1 \Rightarrow \exists q_2 \,.\, (p_2 \xrightarrow{\tau^*}_{T_2} q_2 \ \wedge\ (q_1,q_2) \in R)) \wedge \\
 & & \forall q_2 \,.\, (p_2 \xrightarrow{\tau}_{T_2} q_2 \Rightarrow \exists q_1 \,.\, (p_1 \xrightarrow{\tau^*}_{T_1} q_1 \ \wedge\ (q_1,q_2) \in R)) \wedge \\
 & \forall a \in \mathcal{A} \,. & \forall q_1 \,.\, (p_1 \xrightarrow{a}_{T_1} q_1 \Rightarrow \exists q_2 \,.\, (p_2 \xrightarrow{\tau^* a \tau^*}_{T_2} q_2 \ \wedge\ (q_1,q_2) \in R)) \wedge \\
 & & \forall q_2 \,.\, (p_2 \xrightarrow{a}_{T_2} q_2 \Rightarrow \exists q_1 \,.\, (p_1 \xrightarrow{\tau^* a \tau^*}_{T_1} q_1 \ \wedge\ (q_1,q_2) \in R))\}
\end{array}$$

L'*équivalence observationnelle* pour le langage Λ_{ob} est définie comme le plus grand point fixe de l'opérateur $\mathcal{B}_{ob}$:

$$\approx_o = \nu R \,.\, \mathcal{B}_{ob}(R)$$

Un des inconvénients majeurs de cette relation est qu'elle ne préserve pas la structure de branchement des STEs car elle ne respecte pas la notion intuitive d'égalité des comportements observables. Elle ne permet donc pas la vérification de certaines formules de logiques arborescentes. De plus, la procédure de décision associée à cette relation (fermeture transitive des relations de transition) engendre un coût en temps prohibitif. Son utilité principale vient de sa définition attrayante qui permet de comparer une plus grande classe de systèmes que celle de l'équivalence de branchement. ■

Définition 3.7 (Equivalence $\tau^*.a$ [FM91a])
L'équivalence $\tau^*.a$ est définie comme étant l'équivalence de bisimulation obtenue pour l'ensemble de langage $\Lambda_{\tau^* a}$, tel que :

$$\Lambda_{\tau^* a} = \{\tau^* a \mid a \in \mathcal{A}\}$$

Soit l'opérateur $\mathcal{B}_{\tau^*.a} : 2^{Q_1 \times Q_2} \longrightarrow 2^{Q_1 \times Q_2}$ défini par :

$$\begin{array}{rll}
\mathcal{B}_{\tau^*.a}(R) = & \{(p_1,p_2) \mid & \forall q_1 \,.\, (p_1 \xrightarrow{\tau^* a}_{T_1} q_1 \Rightarrow \exists q_2 \,.\, (p_2 \xrightarrow{\tau^* a}_{T_2} q_2 \ \wedge\ (q_1,q_2) \in R)) \\
 & & \forall q_2 \,.\, (p_2 \xrightarrow{\tau^* a}_{T_2} q_2 \Rightarrow \exists q_1 \,.\, (p_1 \xrightarrow{\tau^* a}_{T_1} q_1 \ \wedge\ (q_1,q_2) \in R))\}
\end{array}$$

L'*équivalence* $\tau^*.a$ pour le langage $\Lambda_{\tau^* a}$ est définie comme le plus grand point fixe de l'opérateur $\mathcal{B}_{\tau^*.a}$:

$$\approx_{\tau^*.a} = \nu R \,.\, \mathcal{B}_{\tau^*.a}(R)$$

Comme pour l'équivalence observationnelle, cette relation ne permet pas de préserver la structure de branchement. Son principal intérêt est qu'elle est plus forte que l'équivalence de sûreté, préserve les propriétés de sûreté, et dispose d'algorithmes efficaces pour la procédure de décision qui lui est associée. ■

Définition 3.8 (Equivalence de sûreté [BFG+91])
Formellement, le *préordre de sûreté*, noté $\sqsubseteq_s$, est le préordre de simulation $\sqsubseteq_\Lambda$ obtenu pour l'ensemble de langages $\Lambda_{\tau^* a}$ défini par :

$$\Lambda_{\tau^* a} \;=\; \{\tau^* a \mid a \in \mathcal{A}\}$$

Soit l'opérateur $\mathcal{I}_s \;:\; 2^{Q_1 \times Q_2} \;\longrightarrow\; 2^{Q_1 \times Q_2}$ défini par :

$$\mathcal{I}_s(R) \;=\; \{(p_1, p_2) \mid \quad \forall q_1 \,.\, (p_1 \stackrel{\tau^* a}{\rightarrow}_{T_1} q_1 \Rightarrow\; \exists q_2 \,.\, (p_2 \stackrel{\tau^* a}{\rightarrow}_{T_2} q_2 \;\wedge\; (q_1, q_2) \in R))\}$$

L'*équivalence de sûreté*, notée $\approx_s$, est définie comme la relation d'équivalence associée à ce préordre, soit l'équivalence de simulation obtenue pour l'ensemble de langages $\Lambda_{\tau^* a}$:

$$\approx_s \;=\; \sqsubseteq_s \cap \sqsubseteq_s^{-1}$$

avec

$$\sqsubseteq_s \;=\; \nu R \,.\, \mathcal{I}_s(R)$$

Il s'agit de l'équivalence la plus faible des cinq relations considérées. Bien que sa procédure de décision soit peu efficace même en appliquant les méthodes de comparaison et de minimisation, et qu'elle ne préserve pas les propriétés d'absence de blocage, elle est adaptée à la vérification de l'ensemble des propriétés de sûreté qui peuvent être exprimées dans une sémantique arborescente. ■

3.1.4 Classement des relations d'équivalence

D'après la thèse de Laurent Mounier [Mou92], nous pouvons résumer l'expressivité comparée de chacune des équivalences en construisant le treillis des relations d'équivalence selon la force d'une équivalence par rapport aux autres. La figure 3.1 décrit cette classification, où un arc orienté d'une équivalence à une autre signifie que l'origine de l'arc est une relation plus forte que la destination.

Cette classification a un double intérêt : tout d'abord théorique mais également pratique afin de déterminer des stratégies pour vérifier qu'un STE satisfait une spécification donnée selon les outils et procédures de décision mis à disposition d'un utilisateur.

3.2 Comparaison locale par équivalence

Les procédures de décision couramment utilisées dans les outils de vérification classiques basés sur les relations d'équivalence comportementales reposent sur le calcul d'une partition compatible avec une relation de transition. Il s'agit du principe général sur lequel s'appuient les algorithmes de comparaison et réduction des systèmes de transitions étiquetées modulo une équivalence de bisimulation. Dans le cadre de l'équivalence forte, le calcul consiste d'abord à obtenir une forme pré-normale, résultant de la modification des relations de transition des systèmes initiaux, en fonction de la relation d'équivalence considérée. Sinon pour l'ensemble des relations d'équivalence, il est nécessaire de construire les classes d'équivalence pour une relation donnée, soit à partir de la forme pré-normale soit directement sur les ensembles d'états des systèmes (c'est-à-dire la partition qui lui est associée). En général, cette seconde phase utilise un algorithme de raffinement de Paige & Tarjan [PT87].

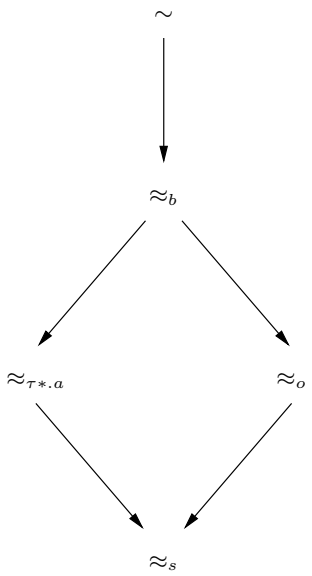


Figure 3.1: Classement des relations d'équivalences vis-à-vis de l'inclusion

Nous nous sommes orientés vers une autre technique, dite *à la volée*, qui peut être mise en œuvre au fur et à mesure de la génération du STE représentant le programme à vérifier, sans construire explicitement sa relation de transition. Une autre contrainte ainsi satisfaite est d'avoir un coût en mémoire raisonnable en pratique et de permettre la comparaison de STEs de tailles réalistes. Cette technique est généralement basée sur l'existence d'une relation de bisimulation entre deux STEs exprimée comme une propriété sur l'ensemble des séquences d'exécution élémentaires de leur produit synchrone qui soit décidable. Plusieurs algorithmes séquentiels pour la comparaison à la volée d'équivalences sont actuellement disponibles :

- Les approches *directes* [FM90, CS01] parcourent le produit synchrone des deux STEs selon la formule d'équivalence décrite dans la section 3.1 en termes d'états et de transitions de ces STEs,
- Les approches basées sur les *logiques temporelles* traduisent les équivalences (forte et observationnelle) en μ-calcul modal [CS91a] ou en des formules caractéristiques du μ-calcul modal et d'ACTL [SI94, NFGR91], et
- Les approches basées sur les *graphes de jeux* [SS98, BLW01a], surtout utilisées pour la vérification de formules logiques en μ-calcul modal, similaires aux approches basées sur les *systèmes d'équations booléennes* [AV95], traduisent l'équivalence (de branchement) en un SEB d'alternance 2.

Dans cette thèse, nous considérons l'approche séquentielle à la volée de [Mat03a, Mat06] basée sur la traduction des relations d'équivalence en termes de SEB d'alternance 1 tel qu'elle est montrée dans le tableau 3.1 de la section suivante. Le problème de comparaison à la volée est réduit à la construction de la fonction successeur du SEB correspondant à la relation d'équivalence entre les deux STEs. L'algorithme de résolution de SEBs utilise ensuite cette fonction successeur pour parcourir incrémentalement les STEs et déterminer leur équivalence modulo la relation d'équivalence choisie. Nous obtenons une version distribuée de la comparaison par équivalence en utilisant simplement l'algorithme DSOLVE pour résoudre le SEB sous-jacent produit à la volée par BISIMULATOR. A notre

connaissance, aucune autre version parallèle d'algorithmes de comparaison à la volée par équivalence n'a été proposée à ce jour.

Nous considérons également par la suite un deuxième aspect important du problème de vérification, qui est de fournir une aide à la localisation des erreurs éventuelles suite à la vérification non-satisfaisante du fonctionnement d'un programme. Il existe différentes méthodes de *diagnostic* qui permettent d'expliquer la non-équivalence entre deux STEs. L'une d'elle est d'exprimer le diagnostic à l'aide de formules de logique temporelle [Cle90], d'autres font usage de séquences d'exécution [Mou92]. La solution que nous adoptons ici est de produire un *graphe direct acyclique* (DAG) inclus à la fois dans l'implémentation et dans la spécification, résultant du SEB parcouru lors de la comparaison des systèmes, selon l'approche proposée en [Mat00]. Nous utilisons la fonction successeur diagnostic retournée par l'algorithme DSOLVE afin de parcourir et construire le diagnostic distribué à la volée selon le même principe que le calcul de comparaison.

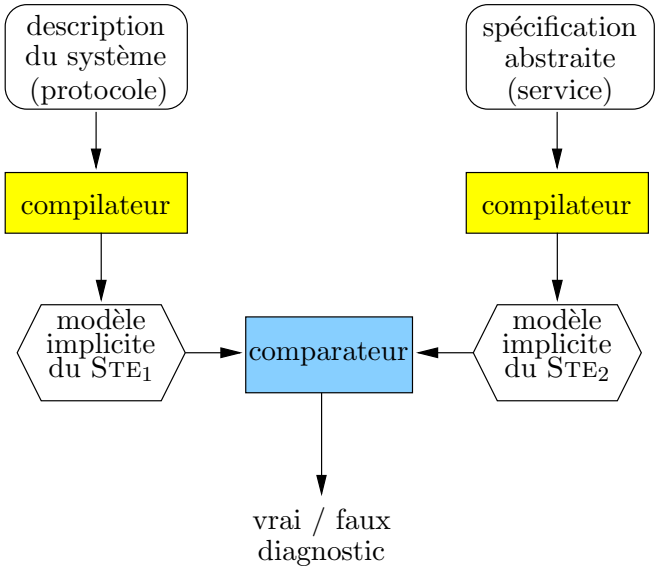


Figure 3.2: Comparaison locale par équivalence

Le mécanisme général de comparaison à la volée par équivalence peut être illustré par la figure 3.2. Les fonctions successeurs décrivant les STEs du système à vérifier (par exemple un protocole) et de ses propriétés comportementales (par exemple le service définissant le protocole) sont utilisés par le comparateur par équivalence qui retourne la valeur de vérité de la comparaison des deux STEs modulo la relation d'équivalence spécifiée, ainsi qu'un diagnostic justifiant (exemple ou contre-exemple) la valeur de vérité.

3.3 Traduction de la comparaison par équivalence en termes de SEBs

De nombreuses relations d'équivalence entre STEs ont été caractérisées en termes de formules du μ-calcul modal [CS91a] ou en termes de SEBs avec alternance 2 [AV95]. Le tableau 3.1 montre la traduction en SEBs d'alternance 1 [Mat03a] des cinq relations d'équivalence précédemment citées et qui sont largement utilisées dans la littérature et en pratique : l'équivalence forte [Par81], de

branchement [vGW96], observationnelle [Mil89], $\tau^*.a$ [FM91a], et de sûreté [BFG$^+$91]. Chaque relation est représentée par un STE de plus grand point fixe, dont les variables $X_{p,q}$ indiquent si les états $p \in Q_1$ et $q \in Q_2$ sont équivalents ou pas. Nous utilisons comme convention de notation, le fait que $a \in \mathcal{A}$ et $b \in \mathcal{A} \cup \{\tau\}$, où τ dénote l'action silencieuse. Pour chaque équivalence, la relation de préordre correspondante peut être traduite en ne gardant que la partie *grisée* de ces équations.

RELATION	TRADUCTION
Forte	$\left\{ X_{p,q} \overset{\nu}{=} \left(\bigwedge_{p \xrightarrow{b} p'} \bigvee_{q \xrightarrow{b} q'} X_{p',q'}\right) \wedge \left(\bigwedge_{q \xrightarrow{b} q'} \bigvee_{p \xrightarrow{b} p'} X_{p',q'}\right) \right\}$
Branchement	$\left\{ \begin{array}{l} X_{p,q} \overset{\nu}{=} \bigwedge_{p \xrightarrow{b} p'} \left((b = \tau \wedge X_{p',q}) \vee \bigvee_{q \xrightarrow{\tau^*} q' \xrightarrow{b} q''} (X_{p,q'} \wedge X_{p',q''})\right) \wedge \\ \qquad \bigwedge_{q \xrightarrow{b} q'} \left((b = \tau \wedge X_{p,q'}) \vee \bigvee_{p \xrightarrow{\tau^*} p' \xrightarrow{b} p''} (X_{p',q} \wedge X_{p'',q'})\right) \end{array} \right\}$
Observationnelle	$\left\{ \begin{array}{l} X_{p,q} \overset{\nu}{=} \left(\bigwedge_{p \xrightarrow{\tau} p'} \bigvee_{q \xrightarrow{\tau^*} q'} X_{p',q'}\right) \wedge \left(\bigwedge_{p \xrightarrow{a} p'} \bigvee_{q \xrightarrow{\tau^* a \tau^*} q'} X_{p',q'}\right) \wedge \\ \qquad \left(\bigwedge_{q \xrightarrow{\tau} q'} \bigvee_{p \xrightarrow{\tau^*} p'} X_{p',q'}\right) \wedge \left(\bigwedge_{q \xrightarrow{a} q'} \bigvee_{p \xrightarrow{\tau^* a \tau^*} p'} X_{p',q'}\right) \end{array} \right\}$
$\tau^*.a$	$\left\{ X_{p,q} \overset{\nu}{=} \left(\bigwedge_{p \xrightarrow{\tau^* a} p'} \bigvee_{q \xrightarrow{\tau^* a} q'} X_{p',q'}\right) \wedge \left(\bigwedge_{q \xrightarrow{\tau^* a} q'} \bigvee_{p \xrightarrow{\tau^* a} p'} X_{p',q'}\right) \right\}$
Sûreté	$\left\{ \begin{array}{l} X_{p,q} \overset{\nu}{=} Y_{p,q} \wedge Y_{q,p} \\ Y_{p,q} \overset{\nu}{=} \left(\bigwedge_{p \xrightarrow{\tau^* a} p'} \bigvee_{q \xrightarrow{\tau^* a} q'} Y_{p',q'}\right) \end{array} \right\}$

Table 3.1: Traduction en termes de SEBs de cinq relations d'équivalence largement utilisées

Tous les SEBs montrés dans le tableau 3.1 peuvent être rendus simples (au prix d'une augmentation linéaire en taille) en introduisant des variables additionnelles telles que la partie droite des équations devienne soit une formule disjonctive, soit une formule conjonctive. Par exemple, le SEB pour l'équivalence forte est transformé de la manière suivante :

$$\left\{ \begin{array}{l} X_{p,q} \overset{\nu}{=} \bigwedge_{p \xrightarrow{b} p'} Y_{p',b,q} \wedge \bigwedge_{q \xrightarrow{b} q'} Z_{p,b,q'} \\ Y_{p',b,q} \overset{\nu}{=} \bigvee_{q \xrightarrow{b} q'} X_{p',q'} \\ Z_{p,b,q'} \overset{\nu}{=} \bigvee_{p \xrightarrow{b} p'} X_{p',q'} \end{array} \right\}$$

La vérification consiste alors à résoudre la variable $X_{q_{0_1},q_{0_2}}$ — qui dénote l'équivalence des états initiaux des deux STEs — en appliquant un algorithme de résolution à la volée. La construction incrémentale du SEB durant la résolution revient à une exploration des deux STEs à la volée, puisque les formules en partie droite des équations sont évaluées en traversant les transitions du STE par un parcours en avant.

3.4 Réalisation et application

L'ensemble des relations d'équivalence citées précédemment, ainsi que les différents types d'algorithmes effectuant la comparaison ont fait l'objet d'implémentations au sein de plusieurs boîtes à outils de vérification, telles que CADP [GLM02], CWB [CPS89] et FC2TOOLS [BRRd96], et il est ainsi possible d'une part de comparer leurs comportements en pratique de manière objective, et d'autre part de déceler leurs principales limites lorsqu'ils sont appliqués à des STEs de grandes tailles.

3.4.1 Développement de la connexion à BISIMULATOR

L'outil BISIMULATOR [Mat03a, BDJM05] (voir la figure 3.3) a été développé au sein de CADP [GLM02] en utilisant l'environnement générique OPEN/CÆSAR [Gar98] pour l'exploration à la volée de STEs.

BISIMULATOR consiste en deux parties : une partie avant, responsable de la traduction des cinq relations d'équivalence en SEBs et de l'interprétation des contre-exemples fournis par la résolution de SEB en termes des deux STEs donnés en entrée ; et une partie arrière, qui effectue la résolution de SEB et sert comme moteur de vérification. La version séquentielle [Mat03a, BDJM05] et distribuée [JM04] de l'outil sont obtenues en utilisant comme partie arrière soit l'algorithme séquentiel de la bibliothèque CÆSAR_SOLVE [Mat03a, Mat06], soit l'algorithme DSOLVE, respectivement. Les successeurs d'une variable booléenne $X_{p,q}$ dénotant l'équivalence des états p et q modulo une relation donnée, sont calculés par la partie avant de BISIMULATOR — qui est appelée séquentiellement et indépendamment sur chaque nœud travailleur — par une exploration en avant des deux STEs en commençant par p et q, selon la traduction en SEB de cette relation (voir le tableau 3.1). Il est à noter que pour les relations d'équivalence faibles (c'est-à-dire, branchement, observationnelle, $\tau^*.a$ et sûreté), la partie avant doit effectuer des fermetures transitives sur les transitions τ dans chacun des STEs. Cette architecture modulaire ne pénalise pas les performances : la version séquentielle de BISIMULATOR [Mat03a, BDJM05] se compare favorablement aux algorithmes spécifiques à la volée implémentés dans ALDÉBARAN [FM91b].

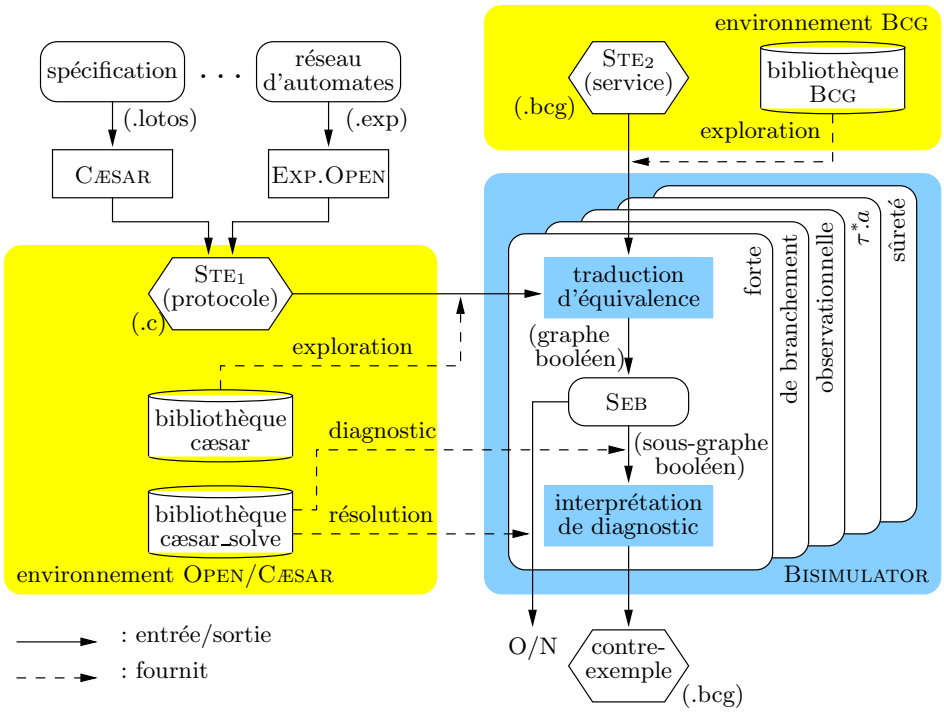


Figure 3.3: Architecture de l'outil BISIMULATOR

Dans le but de comparer les versions distribuée et séquentielle (basée sur l'algorithme de parcours en largeur d'abord de CÆSAR_SOLVE) de BISIMULATOR, nous avons effectué un ensemble d'expériences étendues sur différents STEs obtenus à partir d'études de cas de CADP et de la suite de tests VLTS [VLT03], qui a été conçue pour être un critère de référence pour les réalisations scientifiques d'algorithmes et d'outils opérant sur des STEs de grande taille. Seules huit expériences, qui prennent au moins quelques secondes de calcul, sont montrées dans cette section. Notez que pour obtenir une image précise des performances, nous excluons les coûts (fixes) des activités dépendantes du système (chargement du code sur les nœuds distants, initialisation des connexions, copie des fichiers de STE), et nous gardons seulement les coûts (variables) de la résolution distribuée et de la détection

de terminaison. Nous avons effectué chacune des expériences dix fois. Chaque point de chaque courbe correspond à la moyenne des huit valeurs correspondant aux mesures obtenues, en excluant les valeurs maximale et minimale.

3.4.2 Analyse de performances

Accélération La figure 3.4 montre des données expérimentales décrivant l'accélération absolue (T_s / T_P sur l'axe des ordonnées) de la version distribuée de BISIMULATOR (T_P avec P nœuds XEON, $P \in [2..15]$ donné par l'axe des abscisses) par rapport à la version séquentielle basée sur un parcours en largeur d'abord (T_s). Pour chaque relation d'équivalence R et STE M, les expériences concernent la comparaison modulo R de M avec M_R, sa version minimisée modulo R. Le choix de cette comparaison a été motivé par deux raisons :

(a) il reproduit une situation fréquemment rencontrée en pratique, lorsque le concepteur spécifie à la fois le comportement du système (*protocole*) et son comportement externe (*service*), qui correspondent ici à M et M_R ;

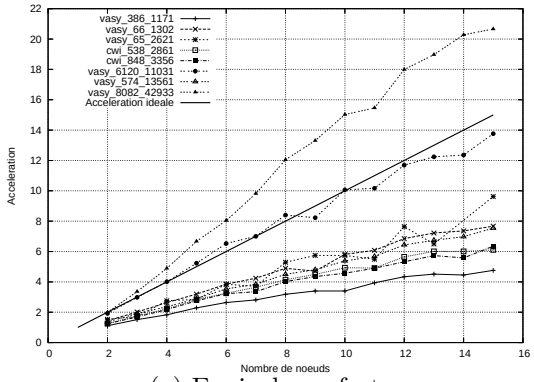
(b) il représente un comportement dans le pire des cas pour la comparaison à la volée par équivalences, car les algorithmes doivent explorer le SEB (et, par conséquent, les deux STEs) entièrement avant de décider de l'équivalence de M et M_R.

Les expériences ont été choisi de manière à représenter le plus fidèlement possible le spectre globale des valeurs possibles d'accélération obtenues avec les cinq relations d'équivalence sur l'ensemble des 65 études de cas de CADP et de la suite de tests VLTS. Nous avons ainsi retenu les huit expériences suivantes : *vasy_386_1171*, *vasy_66_1302*, *vasy_65_2621*, *cwi_538_2861*, *cwi_848_3356*, *vasy_6120_11031*, *vasy_574_13561* et *vasy_8082_42933*.

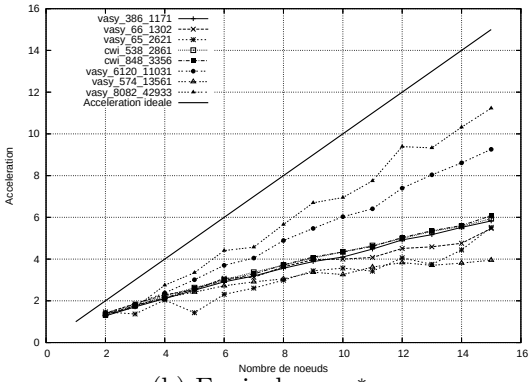
Nous avons aussi réalisé diverses expérimentations comparant des STEs non équivalents : dans tous les cas, les versions distribuée et séquentielle de BISIMULATOR furent extrêmement rapides (quelques secondes) dans la découverte de contre exemples.

Equivalence forte. La figure 3.4(a) montre les accélérations obtenues pour la comparaison par équivalence forte avec BISIMULATOR distribué sur un ensemble de STEs ordonnés par nombre croissant de transitions, allant de $386 \cdot 10^3$ états et $1.171 \cdot 10^6$ transitions (*vasy_386_1171*) à $8.082 \cdot 10^6$ états et $42.933 \cdot 10^6$ transitions (*vasy_8082_42933*). L'équivalence forte est bien adaptée pour la distribution : nous observons très peu de temps passé dans la partie avant de l'outil (aucune fermeture transitive sur les transitions τ n'est nécessaire), et les courbes montrent une accélération linéaire variant de faible (mais encore meilleure que la version séquentielle) à presque optimale. De plus, l'accélération devient meilleure lorsque la taille du STE augmente. Par exemple, la vérification séquentielle du STE *vasy_6120_11031* (c'est-à-dire $6.120 \cdot 10^6$ états et $11.031 \cdot 10^6$ transitions) prend 160.82 secondes, alors que la vérification distribuée avec 15 travailleurs du même exemple prend 11.69 secondes (soit une accélération de 13.75).

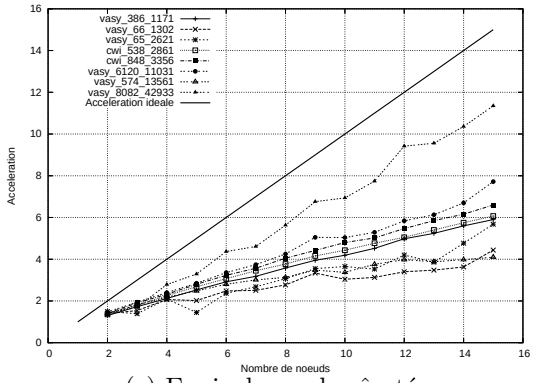
Equivalence $\tau^*.a$ et de sûreté. Les figures 3.4(b) et 3.4(c) montrent les accélérations obtenues pour les équivalences $\tau^*.a$ et de sûreté, respectivement. Les calculs de ces équivalences entraînent de nombreuses (mais moins que pour l'équivalence de branchement et l'équivalence observationnelle) fermetures transitives de transitions τ (réalisées séquentiellement par la partie avant présente sur chaque travailleur) et créent de très petits SEBs dans le cas de STEs avec un haut pourcentage de transitions τ. De là, les accélérations observées sont plus basses que pour l'équivalence forte, et commencent à être élevées sur de larges STEs, tels que *vasy_8082_42933*, où l'accélération augmente jusqu'à plus de 11 avec 15 travailleurs pour les deux équivalences concernées.



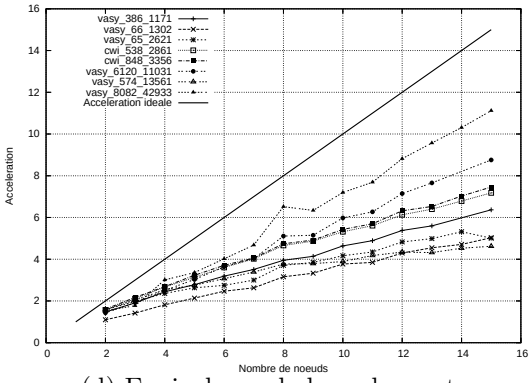
(a) Equivalence forte



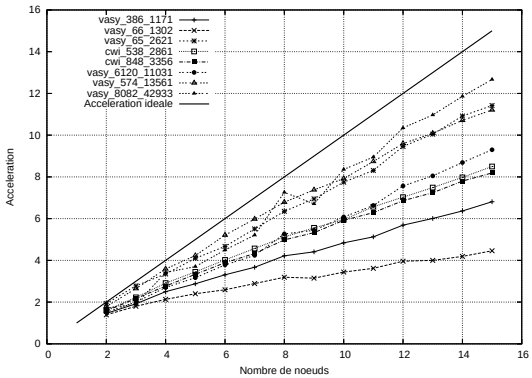
(b) Equivalence $\tau^*.a$



(c) Equivalence de sûreté



(d) Equivalence de branchement



(e) Equivalence observationnelle

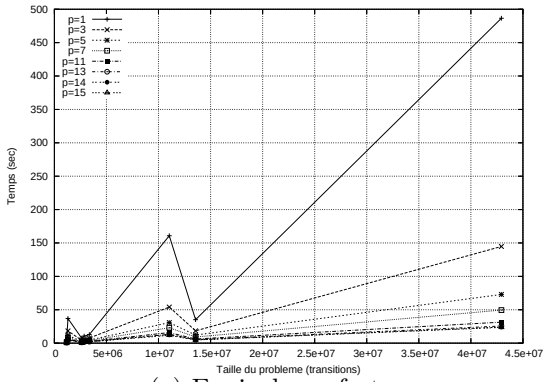
Figure 3.4: Accélération de BISIMULATOR distribué *vs.* BISIMULATOR séquentiel

Equivalence de branchement et observationnelle. Les figures 3.4(d) et 3.4(e) illustrent les accélérations obtenues pour les équivalences de branchement et observationnelle, respectivement. Contrairement aux équivalences $\tau^*.a$ et de sûreté, les SEBs traduisant les équivalences de branchement et observationnelle sont beaucoup plus larges, et en conséquence, la résolution distribuée a un plus grand impact sur les performances. Ainsi, les courbes montrent généralement de meilleures accélérations, en particulier pour des STEs avec un faible pourcentage de transitions τ ou un comportement déterministe, tels que *vasy_65_2621*, où l'accélération augmente jusqu'à 11.41 pour l'équivalence observationnelle avec 15 travailleurs.

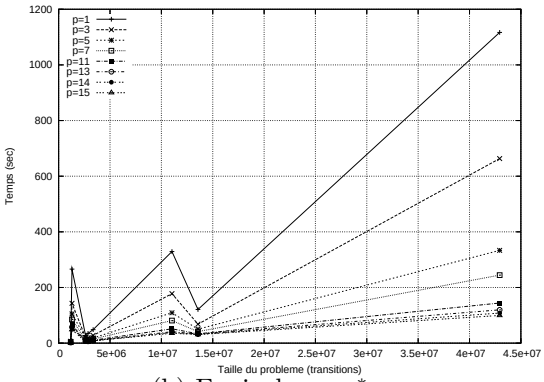
Des observations globales peuvent être faites par rapport à la nature des STEs vérifiés. Trois facteurs influencent les performances de BISIMULATOR distribué : la taille des STEs, le pourcentage de transitions τ, et le degré de non déterminisme. Ainsi, en cas d'absence de transitions τ et de non déterminisme dans les STEs, de bonnes accélérations sont réalisées pour toutes les relations d'équivalence (qui coïncide avec l'équivalence forte dans ce cas), tel que le montrent les expériences *vasy_65_2621*, or *vasy_8082_42933*. Au contraire, un pourcentage élevé de transitions τ entraîne de faibles accélérations pour les équivalences $\tau^*.a$ et de sûreté (à cause des calculs coûteux de la partie avant), mais donne encore de bonnes accélérations pour les équivalences forte (car non affectée par les transitions τ), observationnelle et de branchement (à cause des tailles importantes des SEBs), tel que l'illustre l'expérience *vasy_6120_11031* dans la figure 3.4. De manière similaire, l'augmentation à la fois du degré de non déterminisme et du pourcentage de transitions τ produit de larges SEBs ; dans ce cas, l'équivalence forte est celle qui effectue les plus grandes accélérations, comme le montre l'expérience *vasy_66_1302* dans la figure 3.4(a).

Passage à l'échelle D'autres caractéristiques intéressantes de DSOLVE sont fournies par les mesures expérimentales concernant le passage à l'échelle montrés dans la figure 3.5. Pour chaque équivalence, les chiffres représentent le temps nécessaire (sur l'axe des ordonnées) pour chaque expérience (les mêmes que celles utilisées pour les mesures d'accélération) sur un nombre fixe P de travailleurs XEON (entre 1 et 15). Huit courbes sont ainsi représentées, chacune décrivant le temps nécessaire à la vérification à la volée par équivalence sur respectivement 1, 3, 5, 7, 11, 13, 14 et 15 nœuds XEON. La progression proche du linéaire (et irrégulière dû à la diversité des problèmes traités) des courbes indique que DSOLVE est bien adapté aux augmentations en taille du problème (sur l'axe des abscisses), en faisant une utilisation efficace de la mémoire et du CPU. Plus le STE est large, plus la solution distribuée (par exemple la courbe avec 15 nœuds, $p = 15$) est intéressante par rapport à la solution séquentielle (c'est-à-dire la courbe $p = 1$), ce qui se traduit par un écart de plus en plus large entre les deux courbes $p = 15$ et $p = 1$. Contrairement aux courbes dans la figure 2.10 (colonne de gauche), la forme des courbes dans la figure 3.5 est irrégulière. Cela est dû à la fois à l'irrégularité des problèmes résolus, et à l'absence de STE, dans les expériences, plus large que $4.3 \cdot 10^7$ transitions. Pour un exemple de STE large en nombre de transitions, DSOLVE effectue la vérification par équivalence forte de l'étude de cas *b*188 (Protocole du Bit Alterné avec 188 messages différents), dont la taille du SEB généré est de $2.2 \cdot 10^8$ variables, en à peu près 22 minutes avec 14 travailleurs, alors que la résolution séquentielle de SEB ne peut la réaliser à cause de restrictions actuelles d'implémentation sur la taille du SEB ainsi que sur celle des STEs (maximum de $1.6 \cdot 10^7$ variables).

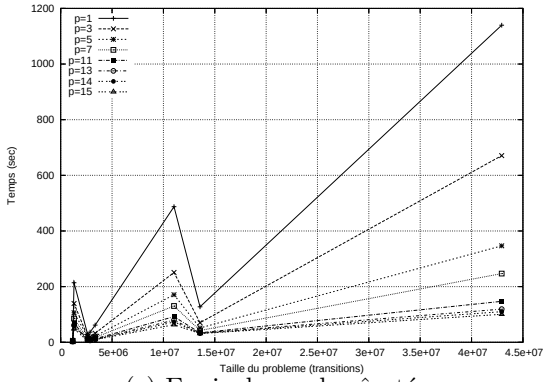
Coût en mémoire et en communication Les figures 3.4 et 3.5 ont montré un bon comportement de BISIMULATOR distribué par rapport au temps d'exécution. Cependant, les gains en mémoire obtenus par la distribution ont beaucoup plus d'importance pour la vérification par équivalence, car ils aident à surpasser les limitations des machines et des techniques séquentielles existantes. La figure 3.6 montre que DSOLVE utilise efficacement la mémoire. Elle présente des résultats obtenus pour chacune des cinq équivalences, sur les huit STEs précédemment extraits de la suite de tests VLTS et testés pour les



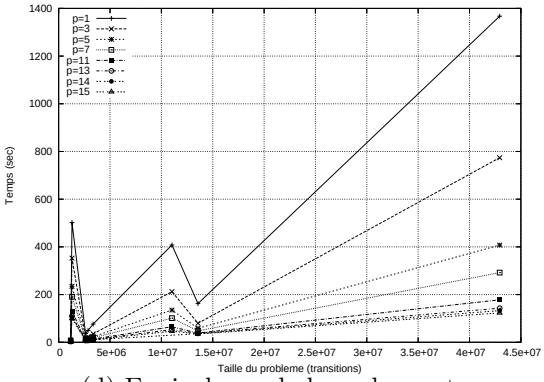
(a) Equivalence forte



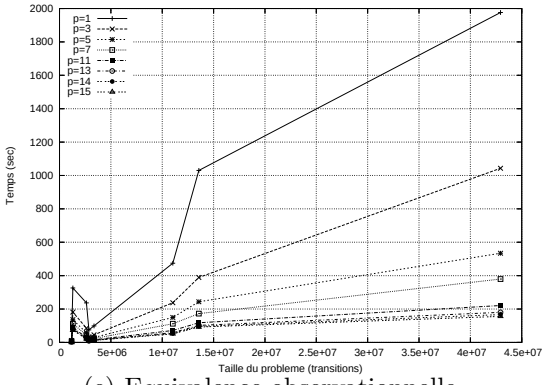
(b) Equivalence $\tau^*.a$



(c) Equivalence de sûreté

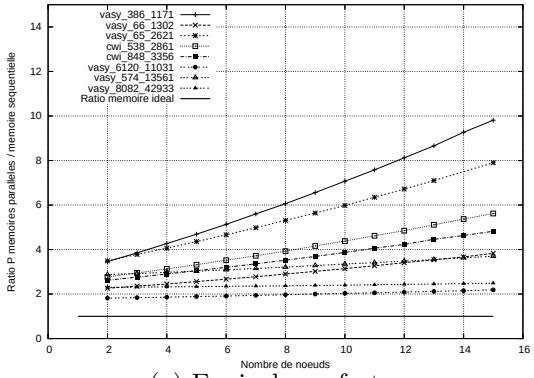


(d) Equivalence de branchement

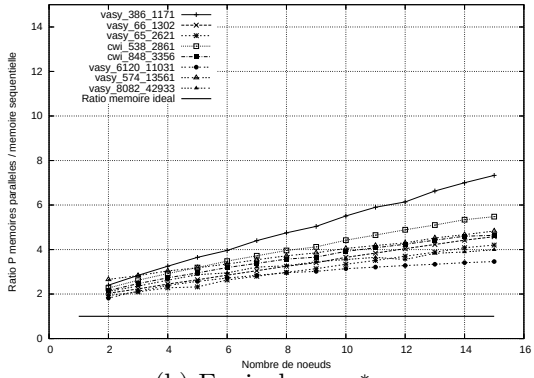


(e) Equivalence observationnelle

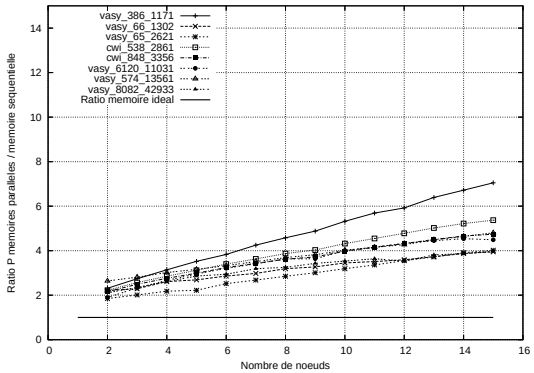
Figure 3.5: Passage à l'échelle de BISIMULATOR distribué selon la taille du STE et selon le nombre de travailleurs



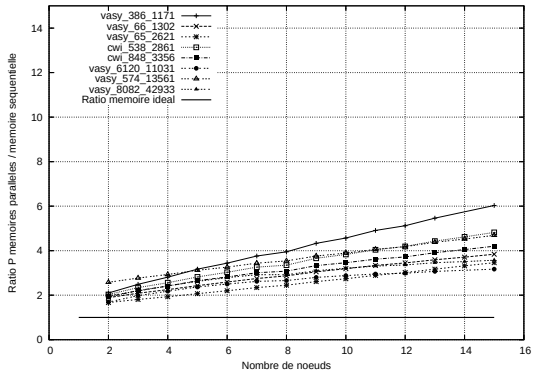
(a) Equivalence forte



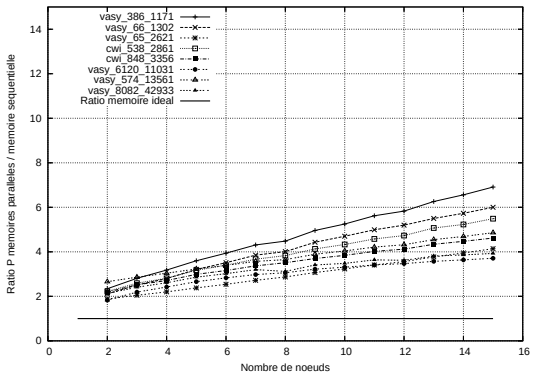
(b) Equivalence $\tau^*.a$



(c) Equivalence de sûreté



(d) Equivalence de branchement



(e) Equivalence observationnelle

Figure 3.6: Surcoût mémoire de BISIMULATOR distribué *vs.* BISIMULATOR séquentiel

mesures d'accélération et de passage à l'échelle. Les expériences sont ordonnées par nombre croissant de transitions dans le STE (de $386 \cdot 10^3$ états et $11 \cdot 10^5$ transitions à $8 \cdot 10^6$ états et $43 \cdot 10^6$ transitions) et par nombre croissant de nœuds travailleurs (de 1 à 15). L'impact de l'augmentation du nombre de travailleurs (sur l'axe des abscisses) est plutôt faible. En effet, le ratio entre la consommation mémoire distribuée totale et la consommation mémoire séquentielle (sur l'axe des ordonnées) pour une même expérience, augmente à peine avec un nombre croissant de travailleurs. Plus grand est le STE à vérifier, plus bas sera le ratio. Seulement pour de très petits STEs (par exemple, *vasy_*386_1171, *vasy_*66_1302 et *vasy_*65_2621), nous pouvons observer une augmentation significative du surcoût mémoire lors de l'ajout de plus de nœuds. Cet effet, qui peut être observé pour chacune des cinq équivalences sur l'exemple *vasy_*386_1171 (le plus petit STE de notre ensemble de tests en nombre de transitions), est dû à une proportion relativement importante du surcoût mémoire liée à la distribution (tampons de communication, tables de hachage) comparée à la taille du SEB résolu.

——— ⋆ ——— ⋆ ——— ⋆ ———

Les données expérimentales présentées jusqu'ici montrent des gains en performance de la version distribuée de BISIMULATOR par rapport à la version séquentielle qui sont encourageants : les accélérations sont quasi linéaires, le surcoût mémoire reste faible, et l'algorithme DSOLVE passe bien à l'échelle en nombre de nœuds travailleurs et en taille du problème à résoudre.

Cette première application de DSOLVE à un problème de vérification a montré le potentiel de notre méthode sur des exemples concrets de taille réaliste extraits d'études de cas industrielles. Elle constitue également un élément de résultat concernant la généricité de notre approche pour différents types de problèmes. Dans le prochain chapitre, nous confirmerons cette grande modularité en traitant un problème orthogonal à la comparaison par équivalence : la réduction à la volée de STE.

Chapitre 4

Application 2 : Réduction d'ordre partiel

Une approche populaire pour la vérification de systèmes distribués est basée sur une exploration exhaustive d'espace d'états. Cette approche souffrant du problème très connu d'explosion d'états, beaucoup de recherches ont été dédiées aux algorithmes qui génèrent un espace d'états réduit, mais essentiellement équivalent. Collectivement, ces méthodes sont appelées méthodes de *réduction d'ordre partiel*.

——— ⋆ ——— ⋆ ——— ⋆ ———

Ce chapitre introduit une nouvelle méthode pour la génération à la volée distribuée d'un espace d'états réduit qui est branchement bisimilaire avec l'original. Cette méthode est basée sur une application de l'algorithme DSolve à la détection de transitions τ-confluentes lors de la réduction d'ordre partiel par τ-confluence d'un Ste. Pour ce problème de vérification, nous définissons dans la section 4.1 la réduction d'ordre partiel de Ste par τ-confluence utilisée dans cette étude. Nous introduisons ensuite dans la section 4.2 le mécanisme de réduction à la volée par τ-confluence. La traduction en termes de SEBs de la détection à la volée de transitions τ-confluentes est présentée dans la section 4.3. Nous décrivons ensuite la connexion de l'outil Tau_Confluence avec DSolve dans la section 4.4.1 et les performances obtenues avec la distribution des calculs dans la section 4.4.2.

4.1 Expression des réductions d'ordre partiel par τ-confluence

La réduction d'ordre partiel essaie d'éliminer les entrelacements redondants induits par les actions indépendantes présentes dans un Ste. Un de ses principaux avantages est qu'elle peut être combinée naturellement à un calcul plus complexe, comme la vérification de propriétés logiques ou comportementales d'un Ste, en l'intégrant au parcours du Ste. La réduction par τ-confluence [GP00] est une forme de réduction d'ordre partiel sur des STEs qui préserve l'équivalence de branchement. Intuitivement, une transition τ sortante d'un état est confluente si son exécution ne change pas le comportement futur du système à partir de cet état. En d'autres mots, toutes les autres transitions sortantes de l'état restent exécutables après la transition τ-confluente, et ainsi elles peuvent être ignorées de manière sûre jusqu'à ce que plus aucune transition τ-confluente ne soit trouvée. Ainsi, calculer l'ensemble des τ-transitions ayant la propriété de τ-confluence permet de réduire le Ste à un autre Ste généralement plus petit, et équivalents par branchement entre eux. Cette réduction, basée sur la détection de transitions τ-confluentes, est appelée τ-prioritisation dans [GP00].

4.1.1 Définition générale

La confluence est une propriété sur des ensembles de transitions invisibles (ou τ-transitions) dans un STE. Elle est plus couramment appelée τ-*confluence*. Différents niveaux de τ-confluence ont été définis dans la littérature. Certaines propriétés de τ-confluence, comme la τ-confluence *ultra faible* de [GS96] ou la τ-confluence *faible* de [Yin00], englobent plus de τ-transitions (et permettent ainsi des réductions plus puissantes), mais sont plus coûteuses pour calculer un ensemble confluent approprié. D'autres, comme la τ-confluence *forte* [GS96], sont plus restrictives, mais permettent des déductions d'ensemble de τ-confluence moins coûteuses. Dans cette étude, nous nous concentrerons sur la détection de τ-confluence forte, à laquelle nous nous référerons dans le reste de cette étude simplement par τ-confluence. Le lecteur intéressé est renvoyé à [Blo01] pour une hiérarchie complète des notions de τ-confluence.

Définition 4.1 (τ-confluence)
Etant donné un STE $M = (Q, A, T, q_0)$, une transition $q_1 \xrightarrow{\tau} q_2$ est dite τ-confluente, si pour toute transition $q_1 \xrightarrow{b} q_3$, une des conditions suivantes est vérifiée :

(a) il existe une transition $q_2 \xrightarrow{b} q_3$,

(b) il existe un état q_4 et deux transitions $q_2 \xrightarrow{b} q_4$ et $q_3 \xrightarrow{\tau} q_4$ telles que cette dernière transition est aussi τ-confluente, ou

(c) $b = \tau$ et il existe une transition $q_3 \xrightarrow{\tau} q_2$ qui est aussi τ-confluente.

■

Intuitivement, une transition $q_1 \xrightarrow{\tau} q_2$ est τ-confluente si toutes les autres transitions sortantes de q_1 peuvent être simulées après avoir exécuté la transition τ-confluente. Pour toute transition τ-confluente, ses états source et destination sont équivalents par branchement [GP00].

Graphiquement, la τ-confluence peut être décrite par la figure 4.1 où nous distinguons les trois différents cas possibles cités précédemment. Les transitions en ligne continue sont données (universellement quantifiées), alors que les transitions en pointillé indiquent que leur existence doit être prouvée.

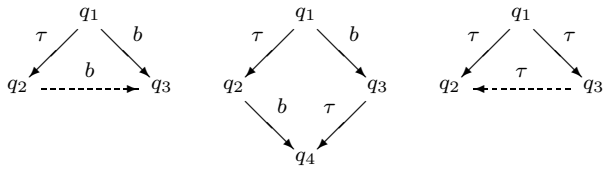


Figure 4.1: Distinction des trois cas possibles de transitions τ-confluentes dans un STE

4.1.2 τ-prioritisation

La τ-prioritisation est une technique pour remplacer un STE avec un plus petit en donnant la priorité aux transitions τ-confluentes sur les autres transitions.

Définition 4.2 (τ-prioritisation)
Etant donné deux STEs M_1 et M_2, avec $M_i = (Q_i, A_i, T_i, q_{0,i})$, M_2 est une τ-prioritisation de M_1 par rapport à la relation de τ-confluence, si $T_2 \subseteq T_1$ et pour toute transition $q \xrightarrow{b}_1 q'$, soit $q \xrightarrow{b}_2 q'$, soit il existe q'', tel que $q \xrightarrow{\tau}_2 q''$ est une transition τ-confluente. ■

La figure 4.2 suivante montre deux exemples de τ-prioritisation (avec les états inaccessibles retirés) :

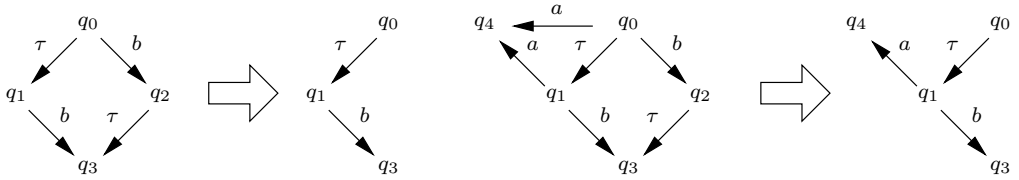


Figure 4.2: Deux exemples de τ-prioritisation

Remarque 4.1
Si M_2 est une τ-prioritisation d'un STE non divergent M_1 (c'est-à-dire ne contenant pas de séquence infinie de τ-transitions), par rapport à la relation de τ-confluence, alors M_1 et M_2 sont branchement bisimilaires. ■

La τ-prioritisation permet ainsi la réduction de systèmes non divergents par rapport à la relation de τ-confluence, en maintenant l'équivalence modulo la bisimulation de branchement. Le principal problème avec la τ-prioritisation est qu'elle est restreinte aux systèmes non divergents. Cependant, on peut étendre la prioritisation pour calculer et éliminer à la volée des cycles de τ [Mat05]. De manière alternative, d'autres techniques de réduction [Blo01, NG02] ont été définies dans la littérature, et peuvent être utilisées.

4.2 Réduction locale par τ-confluence

Plusieurs algorithmes séquentiels globaux [GP00] et à la volée [BvdP02, PLM03] de réduction par τ-confluence sont actuellement disponibles.

Dans [GP00], les auteurs introduisent un algorithme global pour réduire les espaces d'états concrets. En premier lieu, toutes les composantes fortement connexes (CFC) de τ-transitions sont fusionnées dans le but d'éliminer les cycles de τ-transitions. Ensuite, l'ensemble maximal de τ-transitions fortement confluentes est calculé, et la priorité est donnée aux τ-transitions fortement confluentes par rapport aux autres transitions lors de la génération du STE réduit. Ainsi, ces différentes étapes ne peuvent être appliquées qu'après la génération de l'espace d'états non réduit. En particulier, la nécessité de l'absence de cycles de τ-transitions est un obstacle sévère à la génération à la volée de l'espace d'états réduit.

Une première version d'algorithme de génération à la volée de STEs réduits a été présenté dans [BvdP02]. Les auteurs y introduisent une nouvelle notion de τ-confluence basée sur une représentation symbolique de l'espace d'états. Le format symbolique utilisé est celui des *opérateurs de processus linéaire* (LPOs), où un processus linéaire est composé d'un vecteur de variables d'état global, d'un vecteur d'états initiaux et d'un ensemble de règles de programme. La méthode proposée consiste alors à marquer certaines transitions symboliques comme étant confluentes en utilisant un prouveur de théorème automatisé, et de générer par la suite un espace d'états réduit par un algorithme de réduction à la volée.

Une autre méthode modulaire pour la réduction à la volée d'espace d'états, mais utilisant une représentation explicite de l'espace d'états a été décrite par [PLM03]. La détection des propriétés de confluence y est exprimée en termes de SEB. Lors de l'exploration à la volée de l'espace d'états, les τ-transitions, détectées confluentes au moyen des algorithmes séquentiels de la bibliothèque

CÆSAR_SOLVE_1 [Mat03a, Mat06], sont exploitées afin de réduire par τ-prioritisation à la volée le STE. Une version optimisée de l'algorithme de réduction à la volée, combinant la τ-prioritisation et la compression de chemins, a été proposé dans [Mat05]. Les auteurs proposent également des techniques utilisant l'information structurelle d'un système pour réaliser la réduction par τ-confluence de manière compositionnelle, et ainsi traiter des systèmes plus larges qu'avec des techniques basées sur le seul système aplati.

Plus récemment, plusieurs travaux se sont orientés vers une autre combinaison de techniques de vérification, à savoir la vérification distribuée et la réduction d'ordre partiel.

Une des premières tentatives pour combiner les deux méthodes fut réalisée par Lerda et Sisto [LS99] qui étendent le vérificateur distribué de modèles SPIN avec de la réduction d'ordre partiel. L'algorithme proposé permet seulement l'analyse d'accessibilité et utilise la condition conservatrice de fermeture de cycles (c'est-à-dire, les successeurs qui sont traités à l'extérieur du nœud où ils sont calculés, sont toujours supposés être à l'instant présent dans la pile de recherche).

Dans [PG02], une extension de l'algorithme *Twophase* à un environnement distribué, est proposé. Par rapport à l'algorithme de SPIN, Twophase est beaucoup plus simple puisqu'il fonctionne seulement avec des ensembles singletons de transitions indépendantes à partir de chaque état visité. Ces transitions respectent par définition un certains nombre de conditions [Pel96] dont l'inclusion à un sous-ensemble des transitions tirables depuis l'état source, la transparence de leur exécution sur la valeur de vérité de la propriété que la réduction doit conserver, et enfin la non-appartenance à un cycle ne contenant que des transitions indépendantes. Chaque fois qu'il n'y a pas de telle transition satisfaisant les conditions, l'état est complètement exploré. Lors de la génération d'ensembles singletons de transitions indépendantes, le calcul ne se croise pas sur un nœud différent et la condition de fermeture de cycle peut ainsi être vérifiée localement, le but étant de limiter les communications (surcoût principal des algorithmes distribués) inutiles entre les nœuds du réseau. Il est à noter que dans un contexte parallèle, l'algorithme réalise seulement l'analyse d'accessibilité. Pour expérimenter l'extension de Twophase, les auteurs ont développé une connexion de l'outil d'analyse PV [NG98] avec la plate-forme BANDERA [HD01] afin de bénéficier de la partie avant permettant de modéliser des programmes JAVA.

En comparaison à ces deux approches, l'algorithme que proposent les auteurs de [BCMS04], augmente la réduction en considérant non seulement les ensembles singletons de transitions indépendantes, mais également en utilisant une condition moins conservatrice sur la fermeture de cycles. Parallèlement, l'algorithme générant l'espace d'états réduit, peut être combiné avec des algorithmes distribués de vérification de formules logiques lors de la génération de l'espace d'états ou de la vérification de correction par rapport à une propriété LTL donnée. Cependant, cet algorithme n'a jusqu'à présent été expérimenté que pour le passage à l'échelle et l'efficacité de la réduction. Le test de son efficacité pour la vérification totale de propriétés LTL est un travail futur, et le mécanisme complexe de détection distribuée de cycles [BCMS04], nécessaire à la terminaison de la réduction, n'a pour l'instant pas été prouvé.

Par rapport à ces travaux autour de la réduction d'ordre partiel en distribué, nous nous plaçons dans un contexte différent :

- Les travaux décrits dans [PG02, BCMS04] concernent la réduction classique par ordre partiel, c'est-à-dire appliquée aux structures de Kripke, sans souci de préservation d'une bisimulation entre le *système de transitions à états* (STS) original et le STS réduit, et qui exploite la structure interne des états de manière à identifier les transitions indépendantes. En particulier, ces travaux font la supposition (assez restrictive) que toutes les transitions sont déterministes.

  L'approche que nous adoptons repose sur les STEs. Nous cherchons à préserver l'équivalence de branchement entre le STE original et le STE réduit. Pour ce faire, notre méthode ne dépend

ni du contenu des états du STE, ni des propositions atomiques exprimées en LTL et devant être préservées (comme l'invariance de robustesse vis-à-vis d'états silencieux [BCMS04]) lors de la réduction. Les seules informations utilisées sont celles présentes sur les transitions (pouvant être non déterministes), ce qui rend la technique plus générale et adaptée à une exploration à la volée du STE lors de sa réduction.

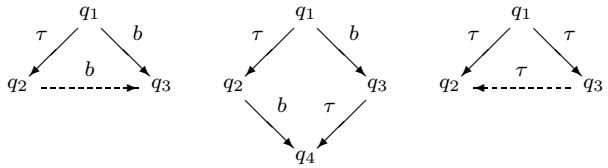
- Le but de notre étude est de montrer une application du solveur distribué DSOLVE pour un problème de vérification classique (la réduction d'ordre partiel). Tout comme le font les travaux des auteurs précédents, nous pouvons utiliser notre méthode pour réduire un STE à la volée de manière distribuée. Il existe néanmoins d'autres manières pour effectuer ce calcul distribué, notamment en combinant un générateur distribué à la volée avec un outil de réduction d'ordre partiel séquentiel (par exemple, l'outil DISTRIBUTOR [GMB+06, GMB+05, Jou02, GMS01] dispose d'une option $-tau\_confluence$ qui permet d'effectuer une telle réduction à la volée localement à chacun des nœuds distribués).

En outre, il s'agit, à notre connaissance, de la première tentative de parallélisation de la détection de τ-confluence. Les travaux précédents et mentionnés ci-dessus, ne sont pas basés sur la détection de τ-confluence, et utilisent un algorithme séquentiel de réduction d'ordre partiel qui est lui-même intégré à un calcul distribué effectuant la vérification de propriétés logiques ou bien l'analyse d'accessibilité. Les calculs distribués sont souvent limités à la seule distribution (par fonction de hachage) des états générés localement à chaque nœud.

Dans la suite de ce chapitre, nous partons de l'algorithme séquentiel à la volée de [PLM03], lequel est basé sur la traduction en SEB de la détection de τ-confluence donné dans la figure 4.3, et nous obtenons une version distribuée en utilisant directement l'algorithme DSOLVE pour résoudre le SEB correspondant.

4.3 Traduction de la détection de τ-confluence en termes de SEBs

La vérification de la τ-confluence d'une transition peut être traduite directement en utilisant un SEB avec un seul bloc ν [PLM03], tel que le montre la figure 4.3. Pour chaque transition $q_1 \xrightarrow{\tau} q_2$, le SEB définit une variable X_{q_1,q_2} indiquant si la transition est τ-confluente ou non (le point fixe maximal ν est utilisé dans le but de caractériser l'ensemble maximal de transitions τ-confluentes contenues dans le STE).



$$X_{q_1,q_2} \stackrel{\nu}{=} \bigwedge_{q_1 \xrightarrow{b} q_3} \left(\left(q_2 \xrightarrow{b} q_3 \right) \vee \left(\bigvee_{q_2 \xrightarrow{b} q_4, q_3 \xrightarrow{\tau} q_4} X_{q_3,q_4} \right) \vee \left((b = \tau) \wedge \bigvee_{q_3 \xrightarrow{\tau} q_2} X_{q_3,q_2} \right) \right)$$

Figure 4.3: Traduction de la détection de τ-confluence en termes de SEB

Chaque cas de τ-confluence décrit par un STE dans la partie supérieure de la figure 4.3 est traduit par la disjonction *grisée* (située juste en dessous) dans la partie inférieure de la figure 4.3 et en partie droite de l'équation booléenne de X_{q_1,q_2}. Dans chaque cas, si la transition $q_1 \xrightarrow{\tau} q_2$ est confluente (c'est-à-dire si X_{q_1,q_2} est vraie), alors pour toutes les transitions $q_1 \xrightarrow{b} q_3$ tirables depuis q_1, on a soit une transition $q_2 \xrightarrow{b} q_3$ portant la même action b, soit un nouvel état q_4 avec $q_2 \xrightarrow{b} q_4$ et une

transition $q_3 \xrightarrow{\tau} q_4$ qui doit être confluente également (c'est-à-dire X_{q_3,q_4} doit être vraie), ou soit une action b invisible et une transition $q_3 \xrightarrow{\tau} q_2$ qui doit être confluente à son tour (c'est-à-dire X_{q_3,q_2} doit être vraie). Lorsque la confluence de $q_1 \xrightarrow{\tau} q_2$ est détectée, toutes les autres transitions $q_1 \xrightarrow{b} q_3$ peuvent être effacées du SEB par τ-prioritisation, tout en préservant l'équivalence de branchement. Les auteurs de [PLM03] ont prouvé que la traduction ci-dessus était correcte. Un ensemble de lois permettant de déduire la τ-confluence en une expression de composition de STEs sans avoir besoin de réaliser des calculs coûteux a ainsi été démontré.

Le SEB de la figure 4.3 peut être rendu plus simple (au prix d'une augmentation linéaire en taille) en introduisant une variable additionnelle Y telle que la partie droite des équations devienne soit une formule disjonctive, soit une formule conjonctive. La figure 4.4 donne le SEB simple équivalent à celui donné dans la figure 4.3, et utilisé par DSOLVE lors de la résolution distribuée de la détection de τ-confluence.

$$\left\{ \begin{array}{lcl} X_{q_1,q_2} & \stackrel{\nu}{=} & \bigwedge_{q_1 \xrightarrow{\tau} q_2, q_1 \xrightarrow{b} q_3, q_2 \not\xrightarrow{b} q_3} Y_{b,q_2,q_3} \\ Y_{b,q_2,q_3} & \stackrel{\nu}{=} & \left(\bigvee_{q_2 \xrightarrow{b} q_4, q_3 \xrightarrow{\tau} q_4} X_{q_3,q_4} \right) \vee \left(\bigvee_{b=\tau, q_3 \xrightarrow{\tau} q_2} X_{q_3,q_2} \right) \end{array} \right\} \begin{array}{l} q_1, q_2 \in Q, \\ q_3, q_4 \in Q, \\ b \in A \cup \{\tau\} \end{array}$$

Figure 4.4: Traduction en SEB simple de la détection de τ-confluence

Remarque 4.2
Ce type de SEB peut aussi être résolu en utilisant des algorithmes séquentiels généraux comme les algorithmes $A1$ et $A2$ présentés dans [Mat03a, Mat06]. ■

Remarque 4.3
Il est à noter que la traduction donnée dans la figure 4.4 permet la construction à la volée du STE pendant la résolution du SEB. ■

Nos résultats expérimentaux préliminaires montrent que la τ-prioritisation réduit le nombre d'états et de transitions du STE jusqu'à un ordre de grandeur. Ces résultats sont présentés dans la section suivante.

4.4 Réalisation et application

Nous avons appliqué notre méthode sur des instances finies de plusieurs algorithmes distribués et protocoles de communication présents dans des études de cas industrielles. La plupart des expériences sont décrites en détail dans la base de tests de la boîte à outils CADP[1].

4.4.1 Développement de la connexion à TAU_CONFLUENCE

L'outil TAU_CONFLUENCE (illustré par la figure 4.5) [PLM03] a été développé dans CADP [GLM02] en utilisant l'environnement générique OPEN/CÆSAR [Gar98] pour l'exploration à la volée de STEs.

TAU_CONFLUENCE consiste en deux parties : une partie avant, responsable de la traduction en SEB de la détection de τ-confluence et de son utilisation pour réduire à la volée le STE en appliquant

[1] Un catalogue d'études de cas industrielles et d'exemples de démonstrations effectués avec la boîte à outils CADP peut être trouvé aux adresses http ://www.inrialpes.fr/vasy/cadp/case-studies et http ://www.inrialpes.fr/vasy/cadp/demos

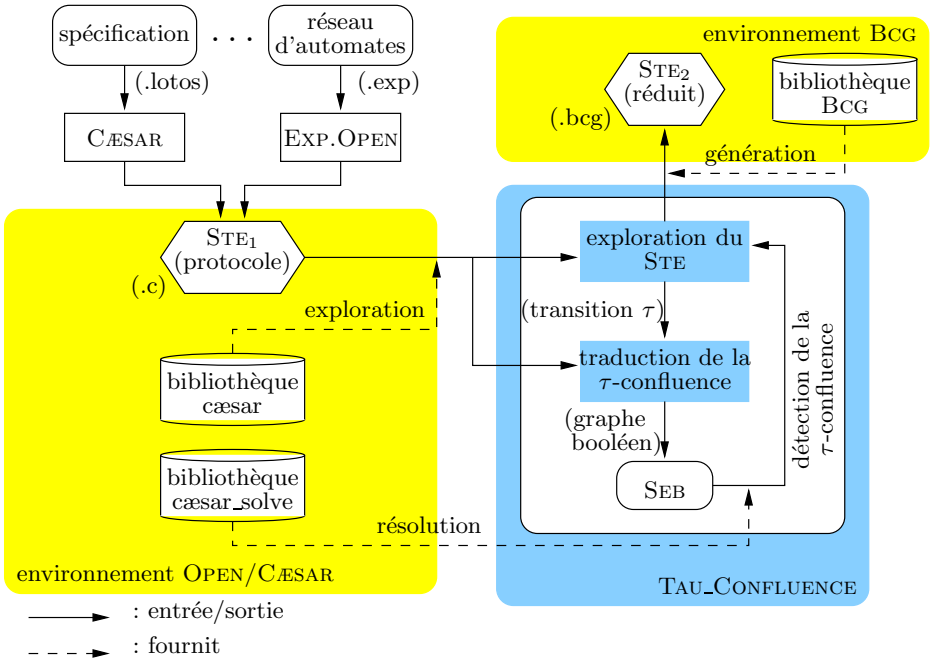


Figure 4.5: Architecture de l'outil TAU_CONFLUENCE

la prioritisation de τ, et d'une partie arrière responsable de la résolution du SEB. Les versions séquentielle [PLM03, Mat05] et distribuée de TAU_CONFLUENCE sont obtenues en utilisant pour partie arrière soit les algorithmes séquentiels de la bibliothèque CÆSAR_SOLVE [Mat03a, Mat06], soit l'algorithme DSOLVE, respectivement. Puisqu'elles partagent la même partie avant, les deux versions de l'outil produisent exactement le même STE réduit, qui est entre un et deux ordres de grandeur plus petit que le STE original dans les expériences sélectionnées.

Lors de la connexion de l'outil TAU_CONFLUENCE au solveur distribué DSOLVE, nous avons été confronté à plusieurs problèmes à la fois architecturaux au niveau du logiciel, mais également plus généraux, comme sur le contexte d'application de notre méthode distribuée :

- Contrairement à l'outil BISIMULATOR (voir la figure 3.3) qui n'a pour résultat qu'une valeur booléenne (donc peu coûteuse à construire), TAU_CONFLUENCE construit au fur et à mesure de son exécution un STE (dénoté STE_2 dans la figure 4.5) sous format BCG. Etant donné que notre architecture est totalement symétrique, les processus exécutés sur chacun des nœuds réalisent exactement la même progression dans la partie avant de l'outil, afin de résoudre la même variable booléenne à chaque instance de calcul distribué, et générer *in fine* le même STE réduit sous format BCG. Pour ce faire, nous synchronisons les fins de résolution distribuée, et les différents processus distribués peuvent ainsi avancer localement d'un même pas séquentiel dans la partie avant. Comme nous pouvons l'observer, cette progression cadencée engendre du temps supplémentaire pour synchroniser tous les processus à chaque appel de résolution distribuée, et des calculs inutiles d'accès aux entrées/sorties pour écrire le STE réduit au format BCG sur le disque local de chacun des nœuds, alors que seul le processus superviseur devrait construire le STE réduit.

  Nous avons expérimenté plusieurs versions de partie avant de l'outil afin de réduire le nombre d'appels de résolution de SEB. Néanmoins, le problème du temps d'exécution prohibitif passé

pour synchroniser les différentes résolutions distribuées a persisté. La raison principale est la compression de τ-transitions (afin de travailler sur des STEs non divergents) et autres optimisations de la partie avant qui génèrent des SEBs beaucoup trop petits pour nécessiter une distribution de leur résolution.

- Un phénomène observé lors de la résolution distribuée et non présent dans la résolution séquentielle a été le problème du choix d'un état *représentant* pour une CFC de τ-transitions. En effet, lors de la résolution distribuée de la détection de τ-confluence, le parcours du SEB correspondant engendre un parcours à la volée du STE en cours de réduction. Ce parcours de STE fait appel à la partie avant de l'outil, qui contient entre autres opérations, une phase de τ-compression [Mat05] permettant la fusion des CFCs de τ-transitions. Chaque nœud établissant ces parcours localement, une même CFC pourra être associée à plusieurs représentants différents sur l'ensemble des nœuds du réseau. Une conséquence directe en est alors une exploration d'un plus grand nombre d'états dans le STE original et de variables dans les SEBs avec la solution distribuée, et donc des résolutions de SEB plus coûteuses, lors de l'utilisation de DSOLVE par l'outil TAU_CONFLUENCE.
- Lors de la résolution distribuée, des informations sont échangées entre les nœuds, notamment des variables booléennes à explorer ou à stabiliser. Ces variables étant liées aux états explorés du STE en cours de réduction, il est nécessaire que deux nœuds différents puissent interpréter les mêmes informations contenues dans la variable échangée. Pour ce faire, les variables ont besoin d'être canoniques. La solution que nous avons pour l'instant adopté a été d'inclure le contenu de l'état, lui-même étant supposé canonique (c'est-à-dire ne contenant pas de pointeurs), directement dans la variable, afin d'avoir une représentation canonique de la variable. Cette solution ne permet par contre aucune amélioration concernant le problème des représentants de variables, ni le traitement d'états contenant des pointeurs.
- Un des objectifs principaux de l'architecture logicielle proposée, est d'avoir une bibliothèque de résolution séquentielle et distribuée de SEBs qui puisse être utilisée indifféremment par les applications utilisateurs, sans que ces dernières n'aient à modifier leur partie avant, en ajoutant par exemple des comportements distincts selon le processus qui exécute le calcul distribué. Cette transparence des mécanismes distribués vis-à-vis de l'utilisateur implique une symétrie des comportements distants de chacun des nœuds concernant l'exécution de leur partie avant. Nous avons vu que dans le cas de la τ-confluence, cette symétrie résultait en une synchronisation artificielle des parties avant, et des coûts élevés en temps d'exécution.

  Il semblerait que la solution la plus appropriée au niveau architectural logiciel soit d'encapsuler et d'exporter les informations liées au réseau (comme l'identification des nœuds travailleurs et superviseur), pour empêcher au niveau applicatif, la génération du STE réduit sous format BCG sur les nœuds travailleurs, et seulement la permettre sur le nœud superviseur.

Nous proposerons par la suite une analyse des performances de la version distribuée de l'outil TAU_CONFLUENCE tel que décrit précédemment, ainsi que des améliorations possibles de notre architecture pour pallier ses limites actuelles.

4.4.2 Analyse de performances

Dans le but de comparer la version distribuée de TAU_CONFLUENCE et sa version séquentielle (basée sur un algorithme de parcours en largeur d'abord de CÆSAR_SOLVE), nous appliquons chacun d'eux sur plusieurs STEs correspondant à des exemples variés de systèmes concurrents.

Réduction Le tableau 4.1 décrit chacun des STEs utilisés pour l'expérimentation de la réduction distribuée par ordre partiel grâce à l'outil TAU_CONFLUENCE connecté à DSOLVE (et utilisation de la grappe de PCS XEON mentionnée au chapitre 2). Pour chacun d'entre eux, nous donnons le nombre d'états, de transitions totales ainsi que le nombre de transitions invisibles, et enfin l'espace disque utilisé pour stocker le STE.

	ORIGINAL			
EXEMPLE	états	transitions	τ-transitions	taille (Ko)
leader_11	2 314 275	16 008 044	16 008 043	13 939
sieve_14	5 152 472	27 595 913	27 589 255	24 674
des	5 189 956	29 359 344	28 315 448	31 816
sum_net_5	533 007	3 169 061	3 169 060	2 752
cwi_566_3984	566 640	3 984 157	3 666 614	4 857
sum_wheel_6	156 957	767 211	767 210	695
rel_rel	150 911	1 249 375	1 147 831	3907

Table 4.1: Tailles des STEs originaux avant la réduction par τ-confluence

Les exemples ont des tailles variant de 5 189 956 états et 29 359 344 transitions (*des*) à 150 911 états et 1 249 375 transitions (*rel_rel*), et décrivent les protocoles suivants :

leader_11 est un protocole d'élection de représentant sur un anneau à jetons composé de 11 processus. Cet exemple, extrait de la plate-forme *Concurrency Factory* [CPS89], a lui-même été inspiré initialement par une version implémentée dans SPIN [Hol97].

Les exemples *leader_11* et *sieve_14* (décrit ci-après) ont été utilisés dans [RS97] comme jeu de tests pour expérimenter une autre approche de réduction d'ordre partiel basée sur l'inertie de τ.

Nous obtenons les différentes versions de leader_* en faisant varier l'entier * correspondant au nombre de nœuds dans l'anneau.

sieve_14 est une implémentation distribuée du crible d'Erathostène utilisant 14 unités dans le tube, extraite de l'étude de cas *demo_36* de la boîte à outils CADP. Sa spécification est basée sur une description écrite dans le langage XL par Y.S. Ramakrishna et C.R. Ramakrishnan, qui elle-même est inspirée d'une spécification en Promela due à D. Peled.

Le système du crible consiste en un tube de $N + 1$ processus :

- le premier processus, appelé générateur, envoie au prochain processus une séquence bornée de nombres consécutifs en commençant par 2, et
- les N autres processus , appelés unités, sont N instances du même processus ; le premier nombre reçu par une unité, qui est premier par construction, est stocké dans une variable P ; ensuite, chaque nombre reçu par la même unité est soit envoyé au suivant s'il n'est pas divisible par P, soit ignoré.

Les processus dans le tube communiquent par tampons de communication asynchrones. Le système contient N tampons, chacun d'eux étant une instance d'un processus intermédiaire. Les N premiers nombres premiers (2, 3, 5, etc.) sont ainsi calculés.

Dans cet exemple, le graphe original d'une instance du crible impliquant 14 unités est généré avec des techniques de vérification compositionnelle.

des est la description d'un processeur totalement asynchrone implémentant le Chiffrement Standard de Donnée (*Data Encryption Standard*).

La version utilisée ici est celle non réduite (c'est-à-dire contenant 5 189 956 états et 29 359 344 transitions) par équivalence de branchement (qui elle contient 67 états et 102 transitions) décrite et analysée en utilisant CADP dans [BBM+03] et [SS05].

sum_* est la spécification d'un algorithme de somme distribuée sur un réseau à jeton composé de N nœuds.

Le système consiste en N processus séquentiels, appelés P_0 à P_N. Chaque processus a deux paramètres, notamment un identifiant unique de processus, représenté par un entier naturel, et un poids, représenté par un entier naturel dans l'intervalle [0..7]. Chaque processus peut communiquer avec un sous-ensemble des autres processus, appelés ses voisins. Le voisinage est une relation symétrique et antiréflexive entre processus.

Le système calcule de façon distribuée la somme (modulo 8) des poids des processus P_0 à P_N : chaque processus collectionne les sommes locales de quelques uns des ses voisins, et le processus P_0 retourne le résultat final une fois calculé.

Nous allons traité deux exemples tirés de ce protocole de somme distribuée :

- *sum_wheel_6* correspond à une topologie réseau avec un processus central connecté aux six autres processus et les six processus connectés en anneau. Cet exemple est également décrit dans l'étude de cas *demo_35* de la boîte à outils CADP.
- *sum_net_5* correspond à une topologie réseau où les cinq processus sont connectés deux a deux.

L'exemple *sum_∗* a été décrit et analysé dans [GMS05]. Ce dernier exemple est un challenge très connu pour les techniques de réduction d'ordre partiel.

cwi_566_3984 est un exemple extrait de la suite de tests VLTS [VLT03], qui a été conçue pour être un critère de référence pour les réalisations scientifiques d'algorithmes et d'outils opérant sur des STEs de grande taille. Le choix s'est porté sur ce graphe car il est un bon candidat potentiel à la réduction par τ-confluence, puisqu'il ne contient presque uniquement des τ-transitions. Par contre rien n'indique que le comportement correspondant est le résultat d'entrelacement de plusieurs processus, condition nécessaire pour obtenir de larges losanges de τ-transitions réductibles par τ-confluence.

rel_rel décrit un protocole multicast atomique et fiable, dénommé rel/REL.

Cet exemple, extrait de l'étude de cas *demo_20* de la boîte à outil CADP, a été généré compositionnellement à partir d'une interface utilisateur pour un réseau composé d'un transmetteur et de 3 processus récepteurs. Il a été résumé et utilisé dans [FGM+92].

EXEMPLE	RÉDUIT							
	états	%	trans.	%	τ-trans.	%	taille (Ko)	%
leader_11	68	$3 \cdot 10^{-3}$	67	$4 \cdot 10^{-4}$	66	$1 \cdot 10^{-4}$	2	$2 \cdot 10^{-2}$
sieve_14	452	$9 \cdot 10^{-3}$	451	$2 \cdot 10^{-3}$	446	$2 \cdot 10^{-3}$	3	$1 \cdot 10^{-2}$
des	3 362	$7 \cdot 10^{-2}$	3 429	$1 \cdot 10^{-2}$	3 230	$1 \cdot 10^{-2}$	6	$2 \cdot 10^{-2}$
sum_net_5	2 548	$5 \cdot 10^{-1}$	2 852	$9 \cdot 10^{-2}$	2 851	$9 \cdot 10^{-2}$	5	$2 \cdot 10^{-1}$
cwi_566_3984	7 125	1.25	13 433	$3 \cdot 10^{-2}$	12 303	$3 \cdot 10^{-2}$	23	$5 \cdot 10^{-2}$
sum_wheel_6	2 881	1.8	3 384	$4 \cdot 10^{-1}$	3 383	$4 \cdot 10^{-1}$	6	$8 \cdot 10^{-1}$
rel_rel	121 978	81	603 600	48	566 277	49	1 308	34

Table 4.2: Tailles des STEs après la réduction par τ-confluence, et ratios par rapport aux STEs originaux

Le tableau 4.2 décrit chacun des STEs réduits par τ-confluence. Pour chacun d'entre eux, nous donnons le nombre d'états du STE réduit ainsi que le ratio entre le nombre d'états réduits et le nombre d'états originaux (c'est-à-dire le pourcentage du nombre d'états du STE original qui reste après réduction). De même, nous donnons ce type d'information pour les transitions totales, les transitions invisibles, et enfin l'espace disque utilisé pour stocker le STE réduit.

Nous vérifions à l'aide de l'outil BISIMULATOR de CADP que chaque STE réduit est équivalent par branchement avec le STE original. Pour bon nombre d'exemples, nous avons dû utiliser la version

distribuée de BISIMULATOR (présentée au chapitre 3) pour résoudre l'équivalence, car la version séquentielle nécessitait plus de mémoire que la quantité disponible localement sur les machines XEON utilisées pour les expériences.

Les plus fortes réductions — de facteur compris entre 10^2 et 10^5 — ont été réalisées pour les exemples *leader_11*, *sieve_14*, *des* et *sum_net_5*.

Les quatre STEs correspondants montrent un haut degré de parallélisme entrelacé et un haut pourcentage de transitions τ, étant ainsi adaptés à la réduction par τ-confluence.

Les plus basses réductions — de facteur inférieur à 10^2 — sont réalisées pour les exemples *cwi_566_3984*, *sum_wheel_6* et *rel_rel*.

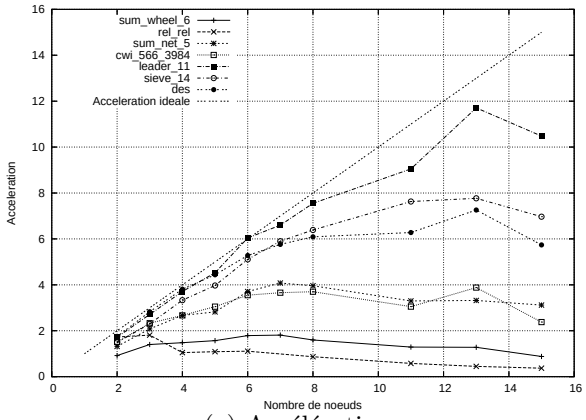
Malgré un haut pourcentage de transitions τ, les trois STEs correspondants n'exposent pas un haut degré de parallélisme entrelacé. De là, seules quelques transitions τ-confluentes peuvent être trouvées et utilisées pour la réduction. On peut également remarquer que la configuration du réseau pour l'exemple *sum_∗* joue un rôle particulièrement important sur la réduction par τ-confluence. En effet, en optant pour une topologie complète, comme dans l'exemple *sum_net_5*, l'entrelacement des processus est favorisé, créant ainsi de larges losanges de τ-transitions, qui seront plus facilement réduits par τ-confluence que dans le cadre d'une topologie en anneau, comme cela est le cas pour *sum_wheel_6*.

Remarque 4.4
Seuls des exemples pouvant être réduits avec l'outil séquentiel TAU_CONFLUENCE sont donnés dans le tableau. En effet, nous souhaitons pouvoir comparer les performances en terme de vitesse de traitement et de consommation mémoire entre la version séquentielle et la version parallèle de TAU_CONFLUENCE sur une même série d'exemples. ■

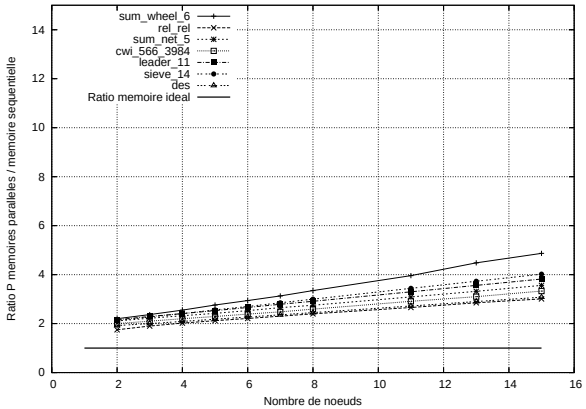
Remarque 4.5
A titre indicatif, les exemples *leader_12*, *sieve_15* et *sum_wheel_7*, obtenus à partir des exemples ci-dessus, ne sont pas réductibles par l'outil séquentiel TAU_CONFLUENCE (par manque de mémoire), mais le sont avec la version parallèle de TAU_CONFLUENCE. Ainsi, nous pouvons réduire l'expérimentation *leader_12* (8 636 956 états et 65 218 244 transitions) à seulement 74 états et 73 transitions, avec 3 nœuds en 1836 secondes, l'expérimentation *sieve_15* (13 628 673 états et 77 448 587 transitions) à 506 états et 505 transitions, avec 5 nœuds en 1500 secondes, et l'expérimentation *sum_wheel_7* (1 737 125 états et 10 189 249 transitions) à 9 844 états et 11 658 transitions, avec 2 nœuds en 1405 secondes, achevant des réductions allant jusqu'à un facteur 10^5 par rapport aux STEs originaux. Les SEBs traduisant les problèmes *leader_12*, *sieve_15* et *sum_wheel_7* ont respectivement 65 218 243 variables et 113 162 578 opérateurs, 77 448 582 variables et 127 639 830 opérateurs, et 34 429 830 variables et 56 994 463 opérateurs, et n'ont pu être résolus qu'à partir de 3 (*resp.* 5 et 2) nœuds utilisés en parallèle car nécessitant environ 4 (*resp.* 6 et 1,6) Go de mémoire pour le calcul. ■

Accélération et surcoût en mémoire La figure 4.6(a) montre l'accélération de la version distribuée de TAU_CONFLUENCE par rapport à la version séquentielle sur les STEs indiqués dans le tableau 4.1 ordonnés par la taille en nombre de transitions, du plus petit STE (*sum_wheel_6*) au plus grand STE (*des*). Chaque nœud du réseau exécute une instance de TAU_CONFLUENCE distribué, et le nombre de nœuds XEON varie de 2 à 15.

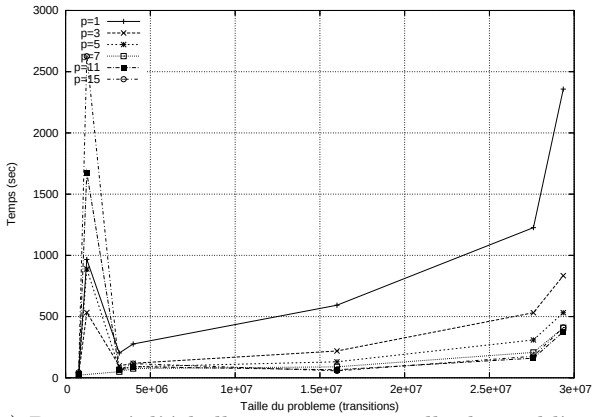
Nous pouvons observer que l'accélération est proportionnelle au pourcentage de réduction obtenu sur les STEs. Plus les STEs peuvent être réduits, meilleure est la rapidité du calcul distribué. Pour certains exemples (*leader_11*), l'accélération obtenue est très proche du linéaire, et nous obtenons une accélération de 11.71 avec 13 nœuds. Ces très bonnes performances sont obtenues grâce à la bonne propriété du problème *leader_11* d'avoir un SEB sous-jacent très large (16 008 043 variables et 27 387 540 opérateurs) et un nombre d'invocations de résolution très petit (66 résolutions initiées



(a) Accélération



(b) Surcoût mémoire



(c) Passage à l'échelle en temps et taille du problème

Figure 4.6: Accélération (a), surcoût mémoire (b) et passage à l'échelle (c) de la réduction à la volée distribuée par τ-confluence par rapport à l'outil Tau_Confluence séquentiel

dont aucune redondante), soit en moyenne un SEB contenant 242 546 variables et 414 962 opérateurs à résoudre en distribué à chaque invocation.

Dans le pire des cas (*rel_rel*), nous n'observons presque aucune amélioration en vitesse, même en augmentant le nombre de nœuds. La raison de ce comportement repose sur l'algorithme de prioritisation de τ, qui réalise un parcours en largeur d'abord du STE et invoque l'algorithme de résolution de SEB pour chaque transition τ rencontrée dans le but de décider si elle est confluente ou non. L'algorithme de résolution séquentielle de SEB n'est pas pénalisé par un aussi grand nombre d'invocations ; cependant, chaque appel à DSOLVE invoque une détection distribuée de terminaison, qui agit comme une barrière de synchronisation en forçant tous les nœuds travailleurs à se synchroniser. DSOLVE pourrait être encore plus efficace dans cette situation, si seulement la quantité de travail requise par chaque résolution était importante ; malheureusement, ce n'est pas le cas en pratique. Par exemple, pour le problème *rel_rel*, un SEB faisant 8 446 921 variables et 15 308 352 transitions doit être résolu. Seulement, la résolution ne se fera pas sur une seule invocation de calcul distribuée : en tout, 754 454 appels de résolution distribuée seront réalisés, dont 708 930 (93%) redondants. La taille moyenne des SEBs ainsi résolus à chaque invocation est de seulement 11 variables et 20 opérateurs booléens, ce qui rend l'utilisation d'une résolution distribuée totalement inadaptée à ce type de problème (c'est-a-dire que les différents surcoûts liés à la distribution des données et du calcul sont trop importants vis-à-vis de la taille du problème).

L'accélération de TAU_CONFLUENCE distribué pourrait être améliorée en changeant l'algorithme d'exploration du STE de la manière suivante : au lieu d'invoquer la résolution de SEB pour chaque transition τ rencontrée (par exemple, sortante de l'état en tête de la file de recherche en largeur d'abord), l'outil TAU_CONFLUENCE invoquerait la résolution sur un ensemble de transitions τ (par exemple, chaque transition τ sortante d'un autre état de la file de recherche en largeur d'abord). En utilisant ce schéma, le nombre de résolutions distribuées (de là le surcoût de détections distribuées de terminaison) sera réduit et la quantité de travail effectuée par chaque résolution sera augmentée ; ces deux facteurs rendront très probablement l'utilisation de DSOLVE plus efficace. Cela requerrait aussi l'extension de DSOLVE pour maîtriser des ensembles de variables booléennes au lieu de simples variables, ce qui peut être fait directement en changeant les phases d'initialisation de l'algorithme (voir la section 2.3). Nous laissons ce sujet de recherche pour travail et expérimentation futurs.

Par ailleurs, les bonnes propriétés d'accélération sont complétées par une bonne consommation mémoire de TAU_CONFLUENCE distribué par rapport à la version séquentielle, tel que le montre la figure 4.6(b). Nous n'observons presque aucune augmentation du surcoût mémoire, même en augmentant le nombre de nœuds. Contrairement aux problèmes de comparaisons par équivalences, que les STEs à vérifier soient petits (par exemple, *sum_wheel_6* et *rel_rel*), ou bien grands (par exemple, *sieve_14*, *leader_11* ou encore *des*), les surcoûts mémoire ont la même augmentation avec l'ajout de nœuds supplémentaires. Ce comportement suggère que la distribution de la détection de τ-confluence passe bien à l'échelle avec le nombre de nœuds travailleurs, et nous permet ainsi de maîtriser des STEs de grande taille.

Enfin, les bonnes performances de TAU_CONFLUENCE distribué sont confirmés par la figure 4.6(c) qui donne une vision plus concrète du passage à l'échelle de notre méthode en taille du problème à résoudre par rapport au temps d'exécution. Plus grand (en termes de transitions) est le STE, meilleur sera le gain d'une solution distribuée sur une solution séquentielle, comme le montre l'écart entre la courbe séquentielle (c'est-à-dire $p = 1$) et, par exemple, la courbe avec 15 nœuds (c'est-à-dire $p = 15$). A part le premier pic observé sur la figure, qui représente l'exemple *rel_rel* pénalisant la version parallèle par des invocations répétitives et redondantes de résolution, l'ensemble des expérimentations ont un temps d'exécution proportionnel à leur taille en nombre de transitions.

——— ⋆ ——— ⋆ ——— ⋆ ———

Nous avons présenté une nouvelle méthode pour la réduction à la volée distribuée d'un système de transitions en utilisant plusieurs machines connectées par un réseau. Cette méthode a été expérimentée avec succès sur le réducteur de τ-confluence TAU_CONFLUENCE dont les fonctionnalités ont été étendues pour permettre le calcul distribué.

Cette seconde application à un problème de vérification montre bien la généricité de notre approche, basée sur la résolution distribuée de SEBs. Nous avons montré que la distribution de la détection à la volée de τ-confluence permettait à la fois d'accélérer la réduction par ordre partiel, mais également de traiter des problèmes plus larges, non réductibles par les méthodes séquentielles existantes.

Les indicateurs essentiels de performance pour ce genre de problème sont la consommation mémoire, qui passe très bien à l'échelle avec le nombre de machines, et la réduction finale obtenue sur le STE, qui peut atteindre un facteur 10^5.

Les avancées obtenues dans cette étude sur la réduction à la volée distribuée de systèmes de transitions peuvent être continuées dans plusieurs directions. Premièrement, TAU_CONFLUENCE peut être étendu à d'autres formes de réductions d'ordre partiel, telles que la τ-confluence faible, et l'inertie de τ, où les losanges de τ-transitions sont plus grands et donc plus intéressants à réduire de manière distribuée. Enfin, le modèle de programmation parallèle utilisé actuellement pourrait être modifié en un modèle client/serveur, où les nœuds clients pourraient avoir des comportements asymétriques entre eux, et ne travailler que sur requêtes spécifiques du serveur. Ce modèle permettrait ainsi de minimiser le coût de chaque résolution de variable booléenne, qui engendre dans le cas présent une détection distribuée de terminaison sur l'ensemble des nœuds.

Finalement, l'algorithme DSOLVE a été appliqué avec efficacité à deux applications de vérification et des résultats très prometteurs ont été obtenus pour chacune d'entre elles. Néanmoins, DSOLVE montre des limites concernant le type de problème pouvant être résolu de manière distribuée. En effet, seule la résolution de SEBs monoblocs y est traitée, ce qui rend, par exemple, impossible son utilisation pour la vérification de logiques temporelles ou encore pour la génération automatique de cas de tests. Nous nous proposons de considérer une extension de l'algorithme DSOLVE dans le prochain chapitre, et son expérimentation sur des problèmes de vérification nécessitant la résolution de SEBs multiblocs.

Chapitre 5

Résolution distribuée à la volée de SEBs multiblocs

Généralement, les problèmes de vérification formelle se traduisent en des calculs de points fixes. Certains, comme la comparaison par équivalence de deux automates ou la réduction d'ordre partiel d'un automate, ne nécessitent le calcul que d'un seul point fixe, en l'occurrence celui d'un plus grand point fixe. D'autres, comme la vérification de propriétés logiques temporelles sur un système de transitions, peuvent résulter en un calcul d'une succession de points fixes de polarité quelconque. Jusqu'à présent dans cette étude, la solution distribuée proposée ne permet de résoudre que des problèmes ne dépendant que d'un seul point fixe. Afin d'obtenir une solution totalement satisfaisante, sa capacité de traitement doit être étendue à l'ensemble des problèmes de vérification formelle basés sur les systèmes de transitions, impliquant plusieurs calculs de points fixes.

——— ⋆ ——— ⋆ ——— ⋆ ———

Ce chapitre présente une extension conservatrice [JM06] de l'algorithme monobloc DSolve présenté au chapitre 2. Il est le résultat du déroulement logique de ce travail de thèse, qui consiste à donner en premier lieu une solution au problème simplifié de la résolution distribuée de SEBs monoblocs, pour ensuite ajouter des couches de complexité à cette solution permettant à l'algorithme final d'être plus général et de traiter des SEBs contenant un ou plusieurs blocs sans dépendances mutuellement récursives.

La section 5.1 est dédiée à l'étude de solutions existantes ou envisageables pour la résolution distribuée à la volée de SEBs multiblocs. Elle est décomposée en deux parties : la première décrit un mécanisme séquentiel de résolution de SEBs multiblocs (voir section 5.1.1), et la seconde (voir section 5.1.2) explicite trois extensions distribuées possibles de DSolve au problème de résolution de SEBs multiblocs. Une fois la solution distribuée choisie, la section 5.2 présente les différentes extensions nécessaires au niveau des structures de données et des opérations distribuées pour réaliser une telle résolution à partir de DSolve. La section 5.3 synthétise l'extension de DSolve dans un nouvel algorithme, appelé MB-DSolve, qui présente en détail le comportement des nœuds travailleurs. Celui du nœud superviseur est quant à lui traité dans la section 5.4. Finalement, la section 5.5 donne des résultats de performances de cette extension sur un ensemble d'études de cas pour plusieurs problèmes de vérification.

5.1 Limites et propositions d'extension de l'algorithme DSOLVE

La difficulté d'une résolution de SEB multibloc est de gérer la résolution de plusieurs blocs dépendants les uns des autres.

5.1.1 Résolution séquentielle de SEBs multiblocs

La figure 5.1 illustre un SEB multibloc composé de trois blocs, numérotés de 1 à 3. Il a été obtenu à partir de la figure 2.2 du chapitre 2, en regroupant les variables existantes dans des blocs différents. A titre d'illustration, la polarité du bloc 2 a également été changée en un plus petit point fixe (signe μ), et la dépendance $x_6 \rightarrow x_1$ de la figure 2.2 a été supprimée dans la figure 5.1 pour respecter la propriété de non-alternance du SEB multibloc, et empêcher ainsi un cycle de transitions entre le bloc 1 et le bloc 2. Le SEB est décrit en terme de blocs d'équations booléennes en partie gauche de la figure, et sous forme de graphe booléen correspondant en partie droite de la figure. Les mêmes conventions que pour la figure 2.2 sont utilisées, à savoir un cercle plein (*resp.* vide) correspond à une variable fausse (*resp.* vraie).

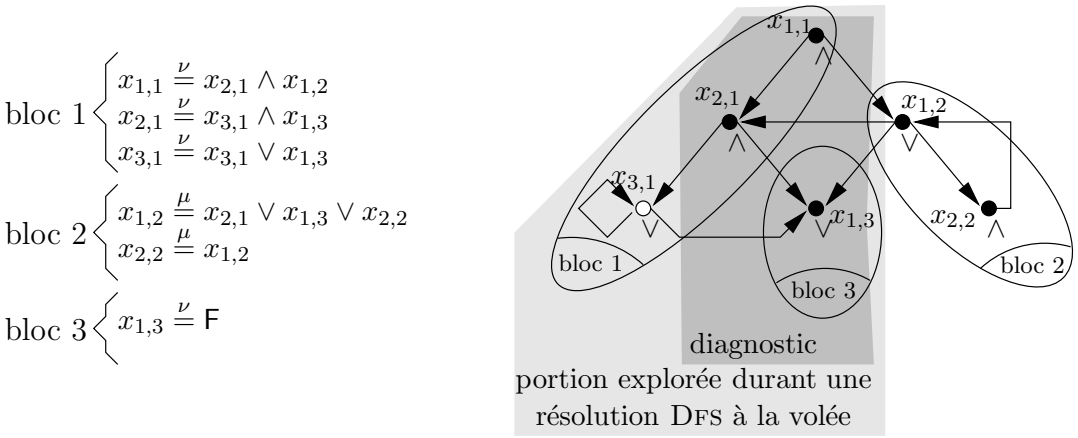


Figure 5.1: Un SEB multibloc, son graphe booléen, et le résultat d'une résolution locale pour $x_{1,1}$

Ce schéma illustre bien les dépendances interblocs (appelées aussi transitions interblocs) existantes entre les différents blocs, comme par exemple la dépendance $x_{2,1} \rightarrow x_{1,3}$ (*resp.* $x_{1,1} \rightarrow x_{1,2}$) entre le bloc 1 et le bloc 3 (*resp.* le bloc 2). La partie grise claire représente une résolution séquentielle possible de ce SEB multibloc, en réalisant un parcours DFS à partir de la variable d'intérêt $x_{1,1}$. La résolution explore d'abord les variables $x_{1,1}$, $x_{2,1}$ et $x_{3,1}$, et ne pouvant stabiliser la variable d'intérêt à partir de l'exploration de cette portion de bloc, elle exécute une nouvelle résolution à partir d'un des successeurs des variables déjà explorées, en l'occurrence $x_{1,3}$. Cette seconde résolution retourne immédiatement la valeur de $x_{1,3}$ qui est stable par définition ($x_{1,3}$ est une constante $false$) à son unique prédécesseur $x_{2,1}$. La propagation arrière de la valeur stable le long des dépendances du bloc 1 aboutit finalement à la stabilisation de la variable d'intérêt $x_{1,1}$.

On peut observer que le bloc 2 n'a pas eu besoin d'être exploré. Dans le cadre de résolutions à la volée, cela se traduit par un gain direct en mémoire et en temps d'exécution, puisque le bloc 2 n'aurait eu besoin ni d'être construit ni d'être résolu.

A partir de la résolution séquentielle précédemment réalisée, il est possible de générer un diagnostic (décrit dans la partie grise foncée de la figure). Ce diagnostic illustre un arbre (ici, un chemin) de

racine $x_{1,1}$ dans le SEB original qui justifie la stabilité de la variable d'intérêt. Il est exprimé en termes de variables du SEB et peut donc recouvrir plusieurs blocs (dans le cas présent, le bloc 1 et 3).

Le mécanisme global de la solution séquentielle utilisée dans la figure 5.1 et décrite dans [Mat03a], consiste à invoquer des résolutions de bloc en cascade le long des dépendances interblocs. Globalement et à chaque instant, l'algorithme séquentiel n'explore qu'un seul bloc à partir d'une variable d'intérêt qui n'est autre que la variable destinataire de la transition interbloc en cours d'exploration. Si ce bloc dépend d'un autre bloc, l'algorithme s'invoquera récursivement sur cet autre bloc, jusqu'à ce que le bloc courant ne dépende plus d'aucun autre bloc, et qu'il soit résolu à son tour. Le SEB multibloc donné en entrée étant d'alternance 1, la pile d'appels récursifs de résolution est bornée par la taille du plus long chemin dans le graphe de dépendances entre les blocs.

5.1.2 Propositions de résolutions distribuées de SEBs multiblocs

Trois solutions possibles d'extension de l'algorithme DSOLVE pour la résolution distribuée de SEBs multiblocs sont envisageables.

Solution distribuée à gros grain La première solution directe, à gros grain, est celle de l'approche séquentielle. A chaque dépendance vers un bloc différent du bloc de la résolution courante, tous les nœuds travailleurs se synchronisent pour résoudre ce nouveau bloc. Dans ce contexte, chaque nœud travailleur impliqué dans le calcul effectue la résolution de la même variable d'intérêt au même instant pour le même bloc. Cette solution présente l'avantage de la place mémoire puisque les propagations se font en priorité par rapport aux explorations, et un nouveau bloc ne sera pas résolu si un bloc précédent peut stabiliser la variable d'intérêt. La propagation y est donc efficace, que la variable destinataire de la dépendance interbloc soit stabilisée *activement* (par la propagation de la valeur stable d'un de ses successeurs) ou *passivement* (par la détection de terminaison de la résolution de la portion de bloc contenant cette variable). Seulement, elle présente de nombreux inconvénients. Tout d'abord elle nécessite une synchronisation forte des nœuds travailleurs pour débuter et terminer chaque nouvel appel de résolution. Cette synchronisation est un frein réel au parallélisme du calcul, et empêche par exemple la résolution de plusieurs blocs en même temps. Pour réaliser une telle solution, la primitive DSOLVE devrait être remodelée pour être réentrante et permettre des appels récursifs de résolution, ce qui ne peut être fait directement avec la version actuelle de DSOLVE. Cette limitation est principalement due à notre architecture logicielle qui a été conçue pour être totalement symétrique à chaque appel de résolution. La synchronisation qui résulterait d'une telle extension laisse présager de faibles accélérations en temps d'exécution par rapport à une version séquentielle. Néanmoins, cette solution a été choisie par d'autres travaux de vérification distribuée comme ceux de [BLW02].

Solution distribuée à grain fin Une autre solution, à grain plus fin, est celle dans laquelle les variables appartenant à des blocs différents ne sont pas différenciées. Une fois que toutes les variables possibles du SEB multibloc ont été explorées et stabilisées, l'algorithme de détection distribuée de terminaison présent dans DSOLVE est utilisé pour permettre la propagation arrière des variables stabilisées passivement, notamment les variables destinataires de transitions interblocs. Cette opération est répétée tant qu'il reste des variables stabilisées passivement qui n'ont pas été propagées à tous leurs prédécesseurs ou bien que la variable d'intérêt n'a pas encore été stabilisée. L'avantage d'une telle solution est qu'elle est très simple à mettre en place, puisque le seul ajout fait à DSOLVE est l'opération d'itération entre détection de terminaison et propagation de variables stabilisées passivement. L'inconvénient majeur est celui de la place mémoire que prend cette solution, car elle favorise l'expansion du SEB et ne permet pas, par exemple, de stabiliser un bloc sans avoir à explorer tous les

autres blocs accessibles à partir de la variable d'intérêt. Le problème principal de la vérification formelle basée sur les modèles étant l'espace mémoire nécessaire pour stocker et parcourir ces modèles, cette solution ne peut nous satisfaire complètement.

Solution distribuée à grain moyen Enfin, une troisième solution, à grain moyen, consiste à affiner la détection de terminaison utilisée par DSolve, pour détecter le fait qu'un bloc (ou une portion explorée de ce bloc) a été résolu entièrement, parallèlement à la résolution d'autres blocs. L'avantage direct d'une telle approche est une bonne parallélisation des résolutions sur plusieurs blocs, ainsi qu'une propagation efficace des variables stabilisées activement et passivement. L'inconvénient principal est la complexité algorithmique due à la multiplication des structures de données proportionnellement au nombre de blocs à résoudre, et donc une consommation mémoire un peu plus élevée que dans le cas d'une solution inspirée de celle séquentielle, où à tout instant, un et un seul bloc est en cours de résolution.

Finalement, nous avons opté pour cette troisième solution et décidé d'entrelacer les résolutions des différents blocs avec un seul appel global de résolution traitant tous les blocs du Seb. Afin de réaliser cette solution, plusieurs aspects de DSolve doivent être étendus :

- les transitions interblocs ont un rôle primordial dans la résolution de Sebs multiblocs. La variable destinataire d'une telle transition peut être soit stabilisée activement par propagation de valeurs stables au sein de son bloc, soit stabilisée passivement après que la portion du bloc accessible à partir d'elle ait été complètement explorée et stabilisée. L'extension de DSolve doit permettre la propagation de valeurs stables le long de ces transitions interblocs indépendamment du type de stabilité des variables destinataires.
- les blocs composant le Seb peuvent avoir des polarités différentes. Dans un bloc de plus petit (*resp.* plus grand) point fixe μ (*resp.* ν), les variables sont initialisées à F (*resp.* T), et l'on propage les variables stabilisées à T (*resp.* F). L'extension de DSolve doit pouvoir traiter indistinctement les deux types de bloc au sein d'une même résolution.
- les blocs ont des niveaux d'imbrication différents influençant le parcours global de la résolution, et l'efficacité de la stabilisation. Chaque bloc étant numéroté, il est possible d'établir un arbre couvrant les dépendances entre les blocs, les nœuds de l'arbre étant les blocs. Lorsque plusieurs blocs sont en cours d'exploration (*resp.* de stabilisation), il importe de donner des priorités à certains blocs selon leur numéro (et donc leur position vis-à-vis du bloc de la variable d'intérêt). Il n'est pas nécessaire de trier topologiquement les blocs, avec un numéro correspondant à la profondeur du bloc dans l'arbre, car cela est déjà réalisé au niveau de la fonction successeur du Seb donnée en paramètre d'entrée. A partir de cet ordre sur les blocs, il est possible de réaliser diverses stratégies d'exploration. Par exemple, il est souhaitable de propager les variables stabilisées de plus haut bloc (c'est-à-dire de bloc le plus près du bloc de la variable d'intérêt) en priorité, car elles ont plus de chances de stabiliser la variable d'intérêt et donc de terminer la résolution plus rapidement. Par contre, pour ce qui est de l'exploration, il est préférable d'explorer les blocs les plus bas dans l'arbre, car ils ont plus de chance de contenir des constantes booléennes, ou des cycles, tous deux stables par définition. Ainsi le parcours global de l'extension de DSolve devra être Bfs localement à un bloc, et Dfs sur l'exploration globale à l'ensemble des blocs.
- enfin, la détection distribuée globale de terminaison doit être étendue pour prendre en compte les stabilités partielles des blocs parcourus. Le nouvel algorithme doit permettre la détection de terminaison de la résolution de chacun des blocs parallèlement à l'exploration et à la stabilisation d'autres blocs.

La solution choisie ne changera en rien le mécanisme, déjà présent dans DSolve, de distribution des tâches d'exploration et de stabilisation, ainsi que la distribution des variables multiblocs. La figure 5.2

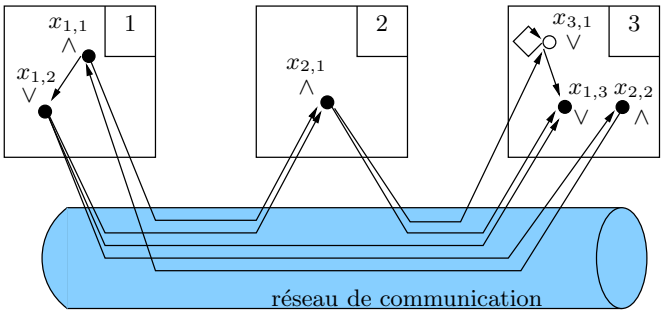


Figure 5.2: Le résultat de la résolution distribuée à la volée de $x_{1,1}$ avec 3 nœuds

illustre ce mécanisme de distribution pour le même SEB multibloc, que celui décrit précédemment, sur trois nœuds. La seule différence notable avec la distribution du SEB monobloc de la figure 2.3 du chapitre 2 est la réindexation des variables par le numéro de bloc auquel elles sont rattachées, ainsi que les légères modifications apportées au SEB multibloc pris ici pour exemple (comme la suppression d'une transition et le changement de polarité d'un bloc) par rapport au SEB monobloc de la figure 2.2.

5.2 Extension des tâches et des données distribuées

La figure 5.3 présente l'ensemble des structures de données utilisées dans l'extension multibloc de l'algorithme DSOLVE appelée MB-DSOLVE, ainsi que dans l'extension de l'algorithme SUPERVISOR.

La première modification qui a été apportée aux données par rapport à celles utilisées dans DSOLVE est la démultiplication des structures de données par le nombre de blocs du SEB à résoudre. Chaque structure est maintenant indexée par un numéro de bloc k et utilisée dans la résolution correspondante à ce bloc. Elles gardent aussi leur indice i indiquant le numéro de nœud sur lequel elles sont définies et locales. Sauf spécifié explicitement autrement, $k \in [1..N]$ est un numéro de bloc compris entre 1 et le nombre maximal de blocs (N) donné en paramètre d'entrée, et $i \in [1..P]$ est un numéro de nœud travailleur. Une hypothèse raisonnable est que l'on connaisse globalement à l'avance le nombre de blocs N et de nœuds P impliqués dans la résolution du SEB multibloc, sans pour autant avoir besoin de le construire.

Une conséquence directe de cette démultiplication est la modification des invariants liant, par exemple, une variable et sa valeur. La valeur des variables booléennes non stabilisées respecte maintenant un invariant dépendant du bloc dans lequel sont définies les variables, qui est : $\forall k \in [1..N], i \in [1..P] \mid v(x_{ki}) = true \Leftrightarrow [(c(x_{ki}) = 0 \wedge \sigma(k) = \mu) \vee (c(x_{ki}) \neq 0 \wedge \sigma(k) = \nu)]$. Ainsi, pour qu'une variable x_{ki} soit *vraie*, elle doit appartenir soit à un bloc d'équations booléennes de plus petit point fixe, et ne plus avoir de successeurs instables, soit à un bloc de plus grand point fixe, et posséder au moins un successeur instable.

5.2.1 Requêtes d'expansion

A part l'extension des structures précédemment utilisées dans DSOLVE à un usage multibloc (c'est-à-dire en leur ajoutant l'indice du bloc auquel elles sont rattachées), de nouvelles structures de données ont vu leur apparition, telles que la liste de propagations résiduelles (R_{ki}), et la liste de requêtes d'expansion (Q_{ki}), qui toutes deux servent à la gestion de la résolution au niveau des dépendances interblocs.

var : /* nœuds travailleurs */
- $c : V \rightarrow \mathbb{N}$ /* nombre de successeurs instables pour chacune des variables de V */
- $d : V \rightarrow 2^V$ /* dépendances en arrière entre variables booléennes de V */
- $v : V \rightarrow$ **Bool** /* valeur booléenne pour chacune des variables de V */
- $b : V \rightarrow \mathbb{N}$ /* numéro de bloc pour chaque variable de V */
- $\sigma : \mathbb{N} \rightarrow \{\mu, \nu\}$ /* type de point fixe pour chaque numéro de bloc */
- $initiator_i$: **Bool** /* *true* si i est le nœud initiateur de la résolution */
- $S_{ki} : 2^V$ /* (Search space) variables du bloc k visitées par le nœud i */
- $W_{ki} : 2^V$ /* (Working space) file BFS de variables du bloc k visitées mais */ /* non explorées par le noeud i */
- $B_{ki} : 2^V$ /* (Backward resolution space) variables du bloc k à stabiliser */ /* par le noeud i */
- $R_{ki} : 2^V$ /* (Residual propagation space) variables instables de bloc k */ /* visitées par noeud i, ayant un prédécesseur de bloc $l \neq k$ */
- $Q_{ki} : 2^{V \times V}$ /* (expansion reQuests space) transitions interblocs (x_{li}, y_{ki}), bloc */ /* destinataire k $(l \neq k)$, en attente d'exploration par le nœud i */
- $exp\_req_{ki} : \mathbb{N}$ /* nombre de transitions interblocs partant du bloc k et locales */ /* au nœud i nécessitant une stabilisation */
- $s : V \rightarrow V$ /* successeur (s'il existe) stabilisant activement la variable de V */
- $stable : V \rightarrow$ **Bool** /* *true* pour toute variable $x_{ki} \in V$ stabilisée */
- $term_{ki}$: **Bool** /* *true* si la terminaison du bloc k a été détectée sur le noeud i */
- $inactif_{ki}$: **Bool** /* *true* si le nœud i n'a plus d'opérations à faire sur le bloc k */
- $sent_{ki} : int$ /* nombre de messages, rattachés au bloc k, envoyés par le nœud i */
- $recv_{ki} : int$ /* nombre de messages, rattachés au bloc k, reçus par le nœud i */
- $sender_i : [0..P]$ /* identifiant du noeud émettant un message au noeud i */
- $x_{ki} : V$ /* variable booléenne non-explorée du bloc k locale au noeud i */
- $y_{ki} : V$ /* variable booléenne d'itération du bloc k locale au noeud i */
- $w_{ki} : V$ /* variable booléenne du bloc k locale au noeud i dépendante de y_{ki} */
- $msg_i : [Evl(x_{ki}, y_{lj}), Exp(x_{ki}, y_{lj}), Trm(k), Sol, Idl(k,t), Act(k), Ack(k,t)]$ /* messages échangés entre nœuds, $x_{ki}, y_{ki} \in V$, $k, l \in [1..N]$ et $t \in \mathbb{N}$ */

var : /* nœud superviseur */
- $term\_statut_k \in [DETECT, CONFIRM, STOP, TERM]$ /* statut de terminaison de la résolution du bloc k */
- $nb\_msg_k : i \in [0..P] \rightarrow int$ /* nombre de messages émis moins ceux reçus par le nœud $i \in [0..P]$ */ /* concernant la résolution du bloc k */
- $total\_msg_k : int$ /* nombre de messages émis moins ceux reçus par l'ensemble des */ /* nœuds du système, concernant le bloc k */
- $stamp_k : \mathbb{N}$ /* estampillage des messages d'accusé de réception lors de la DDT */ /* de la résolution du bloc k */
- $nb\_inactif_k : [0..P]$ /* nombre de noeuds inactifs sur la résolution du bloc k */
- $nb\_ack_k : [0..P]$ /* nombre d'accusés de réception confirmant l'inactivité des nœuds */ /* sur la résolution du bloc k */
- $bcast\_node_k : [1..P]$ /* identifiant du noeud lors de diffusions de messages dans le */ /* contexte de la résolution du bloc k */
- $solution$: **Bool** /* valeur de la variable d'intérêt (appelée x au niveau des nœuds */ /* travailleurs) à la fin de la résolution */
- $sender : [1..P]$ /* identifiant du noeud émettant un message au superviseur */
- $msg : [Trm(k), Sol, Idl(k,t), Act(k), Ack(k,t)]$ /* messages échangés entre nœuds et superviseur, $k \in [1..N]$ et $t \in \mathbb{N}$ */

const :
- $superviseur : 0$

Figure 5.3: Définition des structures de données distribuées (sur nœuds travailleurs et superviseur) pour la résolution de SEBs multiblocs

La structure Q_{ki} permet d'empêcher l'exploration de nouvelles portions de blocs alors qu'une partie de ce même bloc est déjà en cours de résolution. Q_{ki} sauvegarde les transitions interblocs en attente d'exploration sur le bloc destinataire k. Une conséquence directe de son utilisation est de permettre de n'explorer et de ne stabiliser qu'une seule portion de bloc à la fois pour chaque bloc, et ainsi de privilégier la propagation arrière éventuelle de valeurs stables de ce bloc. Un second avantage de Q_{ki} est qu'elle assure l'atomicité de la résolution et de la détection de terminaison de la portion de bloc en cours d'exploration. En effet, pour stabiliser dès que possible les portions de bloc entièrement explorées, il peut être nécessaire de détecter la terminaison de la résolution de cette portion de bloc. Si plusieurs résolutions étaient permises par bloc, il faudrait complexifier grandement l'algorithme de détection de terminaison, afin d'identifier non seulement le bloc, mais aussi quelle portion de bloc est en cours de détection de terminaison, et quelles autres sont simplement en cours de résolution. Afin d'éviter cette complexité coûteuse et peu utile, l'expansion de nouvelles portions de blocs pour un bloc déjà en cours de résolution, est tout simplement mise en attente dans Q_{ki}, et on évite ainsi l'expansion d'une variable d'un bloc pour lequel une détection de terminaison est en cours.

Remarque 5.1
En utilisant de l'information sur la structure du SEB multibloc, il serait possible d'effectuer une optimisation du mécanisme de mise en attente des requêtes d'expansion, en empêchant cette attente pour les requêtes d'expansion de variables appartenant à des blocs *terminaux* (c'est-à-dire sans dépendance vers d'autres blocs). Cette optimisation serait particulièrement utile lors de la vérification de formules logiques, telles que la propriété de famine, pour laquelle le SEB multibloc correspondant contient deux blocs dont l'un est terminal, avec pour chaque variable du bloc de plus haut niveau, une dépendance avec une variable du bloc terminal. En effet, le nombre de détection distribuée de stabilité passive de la résolution sur le bloc terminal serait diminué d'un facteur proportionnel à la taille du bloc terminal, diminuant d'un même facteur le temps d'exécution de la résolution du SEB. ■

5.2.2 Propagations résiduelles

La structure R_{ki} permet d'identifier les transitions qui sont interblocs, et pour lesquelles la valeur de la variable destinataire n'a pas encore été propagée. Afin de prendre en compte les dépendances entre les blocs, et permettre éventuellement la détection de terminaison de la résolution d'un bloc, les transitions interblocs doivent obligatoirement être parcourues par la propagation arrière de valeurs stabilisées activement ou passivement.

Il existe deux sortes de stabilité au niveau des variables booléennes :

- la stabilisation *active* est une stabilisation directe d'une variable x_{ki} de plus grand point fixe (*resp.* de plus petit point fixe) qui peut être obtenue de deux manières différentes : soit lorsque la variable x_{ki} est une $\wedge$-variable (*resp.* $\vee$-variable) et lors de la propagation de la valeur $fausse$ (*resp.* $vraie$) de l'un de ses successeurs, soit lors de la stabilisation de son dernier successeur instable (c'est-à-dire décrémentation du compteur $c(x_{ki})$ à la valeur 0). Dans les deux cas, la variable x_{ki} prend la valeur stable de ce successeur, et devient stable à son tour.
- la stabilisation *passive* d'une variable x_{ki} est quant à elle obtenue par détection de terminaison du bloc k auquel elle appartient. La résolution d'un sous-ensemble d'un bloc peut amener à effectuer toutes les opérations d'exploration et de propagation possibles sans pour autant stabiliser activement toutes les variables de ce sous-ensemble. Il est possible de détecter par un algorithme de DDT la stabilité passive de cet ensemble de variables explorées d'un même bloc. La détection de terminaison permet ainsi de s'assurer de la stabilité de chacune des variables de cet ensemble, et de connaître leur valeur stable, qui est leur valeur courante.

De manière générale, les relations entre ensembles de variables utilisés pour le calcul gardent les mêmes propriétés d'inclusion que les ensembles utilisés pour la résolution monobloc, à savoir : $W_{ki} \subseteq S_{ki}$, $B_{ki} \subseteq S_{ki} \backslash W_{ki}$, $R_{ki} \subseteq S_{ki} \backslash B_{ki}$ *et* $Q_{ki} \subset E$. Les nouveaux ensembles R_{ki} et Q_{ki} apportent de l'information supplémentaire sur les dépendances interblocs, et notamment sur la nécessité de propagation arrière de valeurs stables à travers certaines transitions pour poursuivre la résolution de blocs.

5.2.3 Communication interbloc

Les messages échangés peuvent maintenant se référer à un bloc en particulier, et à ce titre, ils contiennent tous l'information du numéro du bloc auquel ils sont rattachés, et peuvent également contenir de l'information sur le bloc source et le bloc destinataire du message (comme avec $Exp(x_{ki}, y_{lj})$). Cette extension d'information est nécessaire pour pouvoir détecter la terminaison des résolutions de chaque bloc, et des champs supplémentaires sont ainsi rajoutés à la structure de bloc (comme $nb\_sent_k, nb\_recv_k, \ldots$) afin de refléter le statut de la résolution au niveau du bloc k.

5.2.4 Génération de diagnostic multibloc

Dans le chapitre 2 présentant la résolution distribuée monobloc, il a été décrit que la construction du diagnostic (exemple ou contre-exemple) nécessite un second parcours de plusieurs parties du SEB résolu. Pour une variable booléenne donnée, on cherche à connaître le chemin de variables successeurs qui a mené jusqu'à la situation de contre-exemple. Ce parcours n'est permis que si de l'information est retenue lors de la résolution du SEB. Seules les variables stabilisées activement détiennent de l'information pertinente pour la construction d'un tel diagnostic. En effet, en sauvegardant le successeur ayant stabilisé activement une autre variable (utilisation de la structure de donnée s), un sous-graphe du graphe booléen correspondant au SEB et détenant toute l'information pour la construction du diagnostic peut être construit. L'ensemble des variables du diagnostic résultant est alors inclus dans S_{ki}. De plus, et contrairement à DSOLVE, il est nécessaire de garder l'information sur les blocs lors de la sauvegarde de variables pour la génération du diagnostic multibloc.

5.3 Algorithme de résolution MB-DSOLVE

La primitive MB-DSOLVE dont l'algorithme résumé est donné dans la figure 5.4, décrit le comportement d'un nœud travailleur i pour la résolution distribuée sur P nœuds de la variable d'intérêt x du SEB multibloc contenant N blocs défini par le graphe booléen (V, E, L). La distribution des données se fait toujours par une fonction de hachage h connue statiquement par chacun des nœuds.

Le but de cette primitive est de calculer la valeur de vérité (obtenue par stabilisation active ou passive) de la variable d'intérêt x, et d'envoyer ce résultat au nœud superviseur en charge de renvoyer la réponse à l'(application) utilisateur.

Le calcul est amorcé par la définition des données distribuées (ligne 12), telles que décrites dans la section 5.2, et par leur initialisation (ligne 14, et description dans la section 5.3.1). Une itération a ensuite lieu jusqu'à la détection de terminaison de la résolution (lignes 16–21), et qui consiste en quatre opérations :

- la réception de messages éventuellement en attente sur les tampons de réception (lignes 23–26),
- la propagation de variables booléennes stabilisées (lignes 28–32),

```
1  function MB-DSolve (
2        x : V,            /* variable booléenne d'intérêt du Seb */
3        E : V → 2^V,      /* relation successeur */
4        L : V → {∨, ∧}, /* fonction d'extraction du label */
5        N : ℕ,            /* nombre de blocs d'équations booléennes >= 1 */
6        P : ℕ,            /* nombre de nœuds >= 1 */
7        h : V → [1..P],   /* fonction de hachage */
8        i : [1..P]        /* identifiant nœud */
9        )                 /* → envoie au superviseur la valeur Bool de la */
10                         /* variable x du Seb défini par (V,E,L) */
11
12   Distributed data definition /* (Figure 5.3) */
13
14   Initialization /* (Figure 5.6) */
15
16   while ¬term_{b(x)i} do
17   /* conditions de terminaison = */
18   /* (initiator_i ∧ stable(x)) : */
19   /* détection anticipée de terminaison par stabilisation directe de x */
20   /* ∨ ((∀k ∈ [1..N], i ∈ [1..P] ((W_ki = ∅) ∧ (B_ki = ∅))) ∧ aucun_message_en_transit) : */
21   /* détection distribuée de la terminaison (ddt) de la résolution du Seb multibloc */
22
23     /* lecture non-bloquante de messages en attente sur les canaux de communication */
24     if IReceive(msg_i, sender_i) then
25       /* (Figure 5.10) */
26       ReadWorker(msg_i, sender_i)
27
28     /* ensembles de variables à stabiliser non vides ou détection de blocs stables */
29     /* (Figure 5.5) */
30     elsif (l := PendingStabilization) ≤ N then
31       /* (Figure 5.9) */
32       Stabilization
33
34     /* files BFS de travail non vides */
35     /* (Figure 5.5) */
36     elsif (k := PendingExpansion) ≥ 1 then
37       /* (Figure 5.7) */
38       Expansion
39
40     /* files Bfs de travail vides et ensembles de variables à stabiliser vides */
41     else
42       /* lecture bloquante de messages en attente sur les canaux de communication */
43       Receive(msg_i, sender_i);
44       /* (Figure 5.10) */
45       ReadWorker(msg_i, sender_i)
46
47     endif
48   endwhile
49
50   /* envoi du résultat final de la résolution du Seb au nœud superviseur */
51   if initiator_i then
52     Send(Sol(v(x)), superviseur)
53   endif
54
55 end
```

Figure 5.4: Résolution distribuée à la volée d'un Seb multibloc en utilisant son graphe booléen

```
 1 function PendingStabilization : ℕ
 2   /* retourne l'index minimal du */
 3   /* bloc devant être stabilisé */
 4   /* activement ou passivement */
 5   k := 1;
 6   while (k ≤ N) ∧ B_ki = ∅
 7         ∧(¬term_ki ∨ R_ki = ∅) do
 8     if W_ki = ∅ ∧ exp_req_ki = 0
 9       ∧¬inactif_ki then
10       return k
11     else
12       k := k + 1
13     endif
14   endwhile;
15   return k
16 end
```

```
17 function PendingExpansion : ℕ
18   /* retourne l'index maximal du bloc ayant */
19   /* une file Bfs non vide ou bien */
20   /* l'index minimal du bloc ayant */
21   /* des requêtes d'expansion en attente */
22   k := N;
23   while (k ≥ 1) ∧ (W_ki = ∅) do
24     if exp_req_ki = 0 ∧ B_ki = ∅ ∧ R_ki = ∅
25       ∧term_ki ∧ Q_ki ≠ ∅ then
26       return k
27     else
28       k := k − 1
29     endif
30   endwhile;
31   return k
32 end
```

Figure 5.5: Primitives de tests d'existence d'opérations de stabilisation (colonne gauche) et de propagation (colonne droite)

- l'exploration de portions de blocs d'équations booléennes (lignes 34–38), et
- la réception de données supplémentaires ou la détection de terminaison (lignes 40–45).

Deux primitives (présentées dans la figure 5.5) sont utilisées pour le calcul des expressions conditionnelles déterminant quelle branche de l'expression **if − then − else** en cascade va être suivie dans la fonction MB-DSolve :

- la primitive PendingStabilization dont l'invariant d'appel est : $\exists\ k \in [1..N]\ \mid\ (B_{ki} \neq \emptyset \vee (term_{ki} \wedge R_{ki} \neq \emptyset)) \vee (W_{ki} = \emptyset \wedge exp\_req_{ki} = 0 \wedge \neg inactif_{ki})$ renvoie le plus petit numéro l de bloc compris dans l'intervalle $[1..N]$ tel que le bloc l est un bloc à stabiliser, ou $N+1$ sinon, et
- la primitive PendingExpansion dont l'invariant d'appel est : $\exists\ k \in [1..N]\ \mid\ (W_{ki} \neq \emptyset) \vee (W_{ki} = \emptyset \wedge exp\_req_{ki} = 0 \wedge B_{ki} = \emptyset \wedge R_{ki} = \emptyset \wedge term_{ki} = true \wedge Q_{ki} \neq \emptyset)$ renvoie le plus grand numéro k de bloc compris dans l'intervalle $[1..N]$ tel que le bloc k est un bloc à explorer, ou 0 sinon.

La solution finalement renvoyée au nœud superviseur (lignes 50–53) dépendra de la stabilité active ou passive de la variable d'intérêt, ainsi que du point fixe qui lui est rattaché. Ainsi une variable d'intérêt x de plus petit point fixe stabilisée activement ($c(x) = 0$) aura pour valeur finale stable *vraie*.

Remarque 5.2
Bien que le nœud initiateur détienne toute l'information nécessaire pour retourner lui-même la valeur de la variable d'intérêt du Seb multibloc résolu, et afin que les autres nœuds apprennent le même résultat, une phase de synchronisation avec le superviseur tient lieu après la détection de terminaison de la résolution du Seb. Le nœud superviseur envoie la valeur finale de la variable x à tous les nœuds travailleurs, attend qu'ils aient confirmé la réception de cette valeur, pour ensuite leur ordonner de terminer la primitive MB-DSolve. Ce mécanisme est très utile pour certaines applications, comme la réduction d'ordres partiels, où une succession de résolution de Sebs est calculée. Pour cela, il est intéressant de garder actif les nœuds travailleurs et superviseur, et de leur soumettre les différents Sebs à tour de rôle une fois le précédent Seb résolu. La phase de synchronisation de la valeur du Seb résolu et de la terminaison de MB-DSolve devient alors obligatoire pour effectuer la succession de résolutions à partir des mêmes résultats pour chacune des résolutions intermédiaires. ■

```
init : /* nœuds travailleurs */
 1 forall k ∈ [1..N] do
 2    S_ki, W_ki, B_ki, R_ki, Q_ki := ∅;
 3    exp_req_ki := 0;  x_ki, y_ki, w_ki := ⊥;
 4    term_ki, inactif_ki := true;  sent_ki, recv_ki := 0
 5 endfor;
 6 if h(x) = i then /* nœud initiateur */
 7    initiator_i := true;  S_b(x)i, W_b(x)i := {x};  d(x) := ∅;
 8    c(x) := if (σ(b(x)) = μ ∧ L(x) = ∧)
 9    ∨ (σ(b(x)) = ν ∧ L(x) = ∨) then |E(x)| else 1 endif;
10    v(x) := if σ(b(x)) = μ then false else true endif
11 else
12    initiator_i := false
13 endif;
14 term_b(x)i, inactif_b(x)i := false;
15 sender_i, msg_i := ⊥
```

```
init : /* nœud superviseur */
16 forall k ∈ [1..N] do
17    forall i ∈ [0..P] do
18       nb_msg_k(i) := 0
19    endfor;
20    total_msg_k := 0;
21    stamp_k := 0;
22    nb_ack_k := 0;
23    bcast_node_k := ⊥;
24    term_statut_k := TERM;
25    nb_inactif_k := P;
26 endfor;
27 solution := ⊥;
28 term_statut_b := DETECT;
29 nb_inactif_b := 0;
30 sender, msg := ⊥
```

Figure 5.6: Initialisation de la résolution parallèle (sur nœuds travailleurs et superviseur)

5.3.1 Initialisation

Les différentes structures de données utilisées pour la résolution de SEBs multiblocs, sont initialisées dans la figure 5.6. La partie gauche (*resp.* droite) de la figure décrit l'initialisation des structures utilisées dans MB-DSOLVE (*resp.* SUPERVISOR). Celles démultiplexées par le nombre de blocs présents dans le SEB le sont dans les lignes 1–5 (*resp.* lignes 16–26).

On peut regrouper les diverses initialisations en trois catégories : celles se référant à la DDT, celles concernant les variables booléennes, et enfin celles définies pour la communication.

- L'initialisation de la terminaison consiste à déclarer, sur tous les nœuds, que toutes les résolutions de blocs sont terminées (lignes 4 et 16–26) sauf celle résolvant le bloc contenant la variable d'intérêt (lignes 14 et 28–29). Elle est composée de structures de données telles qu'un indicateur d'inactivité des blocs localement à chacun des nœuds, un de terminaison de résolution de bloc, ainsi que des compteurs de messages reflétant la présence ou non de messages en transit sur le réseau. Sur le nœud superviseur, dont le seul rôle décrit ici est la détection de terminaison, une multitude d'autres structures sont utilisées et seront décrites dans la suite de cette section.
- Les variables booléennes sont quant à elles initialisées en fonction du point fixe sur lequel elles sont définies : dans le cadre d'un plus petit point fixe μ, les variables sont mises à $false$, sinon à $true$.
- L'initialisation de la communication inter-nœuds, que cela soit entre nœuds travailleurs seulement, ou entre nœuds travailleurs et nœud superviseur, consiste à définir les tampons de communication pour chaque nœud, ainsi que quelques structures de données, telles que $sender_i$ et msg_i (lignes 15 et 30), afin de permettre l'échange de données.

5.3.2 Expansion

L'exploration du graphe booléen, correspondant au SEB multibloc donné en paramètre d'entrée de MB-DSOLVE, est décrite par la primitive EXPANSION (voir la figure 5.7).

L'ensemble $W_{b(x)i}$ étant la seule file d'exploration BFS non vide à l'initialisation de la résolution, l'exploration commence par la génération des successeurs (ou *expansion*) de la variable d'intérêt x

```
 1 procedure EXPANSION is
 2   if W_ki = ∅ then
 3     /* expansion interbloc */
 4     forall (x_lj, y_ki) ∈ (Q_ki)
 5           ∧¬term_b(x)i do
 6       if j ≠ i ∨ ¬stable(x_lj) then
 7         /* (Figure 5.8) */
 8         BLOCKACTIVITYTEST (k);
 9         EXPAND(x_lj, y_ki)
10       elsif l ≠ k then
11         exp_req_li := exp_req_li − 1
12       endif
13     endfor
14   else
15     /* expansion intrabloc */
16     x_ki := choose(W_ki); W_ki := W_ki \ {x_ki};
17     forall y_lj ∈ E(x_ki) ∧ ¬term_b(x)i
18           ∧¬stable(x_ki) do
19       if k ≠ l then
20         exp_req_ki := exp_req_ki + 1
21       endif;
22       if h(y_lj) = i then
23         if k ≠ l then
24           Q_li := Q_li ∪ {(x_ki, y_lj)}
25         else
26           /* expansion locale du */
27           /* sommet successeur y_lj */
28           EXPAND(x_ki, y_lj)
29         endif
30       else
31         /* expansion distante du */
32         /* sommet successeur y_lj */
33         /* (Figure 5.10) */
34         SENDING(Exp(x_ki, y_lj), h(y_lj))
35       endif
36     endfor
37   endif
38 end
```

```
39 procedure EXPAND(x_kj, y_li) is
40   if y_li ∉ S_li then
41     S_li := S_li ∪ {y_li}; d(y_li) := ∅;
42     c(y_li) := if (σ(l) = μ ∧ L(y_li) = ∧)∨
43                (σ(l) = ν ∧ L(y_li) = ∨)
44                then |E(y_li)| else 1 endif;
45     v(y_li) := if σ(l) = μ then false
46                else true endif;
47     if c(y_li) ≠ 0 then W_li := W_li ∪ {y_li}
48     else stable(y_li) := true
49     endif
50   endif;
51   if k ≠ l ∧ y_li ∉ R_li then
52     R_li := R_li ∪ {y_li}
53   endif;
54   if stable(y_li) then
55     if y_li ∈ R_li then
56       R_li := R_li\{y_li}
57     endif;
58     if h(x_kj) = i then
59       if k ≠ l then
60         exp_req_li := exp_req_li − 1
61       endif
62       /* stabilisation locale du */
63       /* sommet prédécesseur x_kj */
64       /* (Figure 5.9) */
65       STABILIZE(x_kj, y_li)
66     else
67       /* stabilisation distante du */
68       /* sommet prédécesseur x_kj */
69       B_li := B_li ∪ {y_li};
70       d(y_li) := d(y_li) ∪ {x_kj}
71     endif
72   else
73     d(y_li) := d(y_li) ∪ {x_kj}
74   endif
75 end
```

Figure 5.7: Expansion BFS du SEB multibloc à partir de sommets non-stables du bloc k

sur le nœud initiateur (lignes 14–36). Les successeurs de la variable sont ensuite parcourus en largeur d'abord (parcours BFS) et chaque nouvelle variable successeur visitée, peut être explorée à sont tour soit localement soit à distance par un autre nœud.

Par la suite, EXPANSION distingue deux ensembles de variables nouvelles à explorer localement à un nœud i et à un bloc k (k étant la valeur retournée par la primitive PENDINGEXPANSION) :

- les variables destinataires de transitions locales (c'est-à-dire intrabloc) au bloc k (ensemble W_{ki}), et
- les variables destinataires de transitions interblocs de bloc destinataire k (ensemble Q_{ki}).

Exploration interbloc La priorité est donnée à l'exploration de transitions interblocs de bloc destinataire k (lignes 2–13). Lorsque plus aucune transition locale au bloc k n'existe, et que la terminaison de la résolution du bloc k a été détectée (c'est-à-dire $W_{ki} = \emptyset \wedge exp\_req_{ki} = 0 \wedge B_{ki} = \emptyset \wedge R_{ki} = \emptyset \wedge term_{ki} = true \wedge Q_{ki} \neq \emptyset$, voir appel de PENDINGEXPANSION), plusieurs heuristiques d'exploration

des transitions interblocs sont possibles :

- soit traiter une seule transition à la fois de l'ensemble des transitions interblocs, afin de favoriser l'opération de stabilisation en explorant un minimum de dépendances interblocs, ou
- soit traiter l'ensemble des transitions interblocs en attente d'exploration (c'est-à-dire vider complètement Q_{ki}), afin de minimiser le nombre de détections de terminaison partielle d'un bloc, et ainsi limiter le nombre de synchronisations internœuds coûteuses pour un calcul distribué.

Après avoir choisi dans un premier temps la première solution, en suivant la logique de la priorité globale de l'algorithme d'effectuer d'abord les opérations de stabilisation, nous avons finalement opté pour la seconde solution qui présentait de bien meilleurs résultats de temps d'exécution en pratique à cause du nombre de synchronisations coûteuses de la première solution.

L'exploration d'une transition interbloc (x_{lj}, y_{ki}) non visitée est une tâche de travail qui réactive la résolution du bloc k sur le nœud i. L'utilisation de la primitive BlockActivityTest (ligne 8 et figure 5.8) permet ainsi de signaler au nœud superviseur la reprise d'activité sur le nœud k. Si le nœud était déjà actif, l'opération est sans effet. La transition est ensuite explorée par la primitive Expand (ligne 9) où les dépendances entre les variables x_{lj} et y_{ki} sont également mises à jour. Dans le cas d'une variable source x_{lj} locale au nœud i et stabilisée, l'expansion de la transition devient inutile. Si la transition de ce dernier cas est interbloc, il est nécessaire d'indiquer que la requête d'expansion a néanmoins été prise en compte (ligne 11).

Exploration intrabloc Dans le cas d'une expansion intrabloc (c'est-à-dire $W_{ki} \neq \emptyset$), une expansion Bfs du Seb est réalisée à partir d'un sommet x_{ki} du bloc k. Après avoir extrait une variable visitée x_{ki} de la file Bfs W_{ki} (ligne 16, dont l'invariant est : $\forall x_{ki} \in W_{ki},\ c(x_{ki}) \neq \bot \wedge x_{ki} \in S_{ki} \wedge h(x_{ki}) = i$), l'exploration consiste à examiner chacun de ses successeurs y_{lj}, et soit d'envoyer des requêtes d'expansion à des nœuds distants responsables de la variable successeur, soit de mettre à jour localement la file Bfs (W_{li}) de variables du bloc l à explorer, ou bien l'ensemble de requêtes d'expansion interbloc à stabiliser ultérieurement et mises en attente dans Q_{li}.

Remarque 5.3
Une variante possible de parcours des successeurs consisterait à ne parcourir qu'un seul successeur à la fois, et laisser ainsi la possibilité de tester l'existence de variables stabilisées à propager, ou la réception de messages de propagation arrière de valeurs stables. Cette variante de parcours pourrait constituer une optimisation de la résolution de Seb où la variable d'intérêt est stabilisée activement. Par souci d'efficacité, et afin que chaque nœud ait une quantité de travail non négligeable sans avoir besoin d'échanger de nombreux messages avec les autres nœuds, nous avons opté pour l'exploration de tous les successeurs à la fois d'une variable de W_{ki}. ■

Remarque 5.4
La primitive Sending utilisée à la ligne 34 et définie sur la figure 5.10 (colonne droite), est identique en tout point à celle de la figure 2.4, à ceci près que la première prend en considération l'arrêt anticipé de la résolution sur stabilisation de la variable d'intérêt (lignes 35–37) contrairement à la seconde. Sinon, les tâches effectuées sont les mêmes, à savoir, la réception de messages en attente sur les canaux de communication pour résoudre l'interblocage, et l'attente passive, le cas échéant d'évènements de communication à la fois sur les tampons de réception (c'est-à-dire attente bloquante d'une réception de message venant d'un nœud de $\{0..P\}$) et sur les tampons d'émission (c'est-à-dire réussite de l'envoi de message au nœud $sender_i$). ■

L'opération de mise à jour locale de la file Bfs avec une nouvelle variable booléenne est décrite dans la primitive Expand (voir la figure 5.7 (colonne droite)). L'invariant d'appel de cette procédure

```
1 procedure BlockActivityTest (b(x_ki)) is
2   if inactif_ki then
3     inactif_ki := false;
4     term_ki := false;
5     Send(Act(k), superviseur)
6   endif
7 end
```

Figure 5.8: Détection de reprise d'activité et envoi du statut courant au nœud superviseur

est : $\neg stable(x_{kj})$, $x_{kj} \in V$, $y_{li} \in V$, *et* $h(y_{li}) = i$. Un autre invariant qui vient le compléter est : $k \leq l$. Ce dernier invariant permet d'assurer, en ordonnant numériquement les blocs, que la résolution de blocs récursifs (c'est-à-dire des SEBs contenant des cycles de blocs) ne soit pas permise, ce qui correspond à la définition de SEBs alternance 1. Outre l'initialisation de plusieurs structures de données (telles que l'ensemble de prédécesseurs $d(y_{li})$, le compteur de successeurs stables $c(y_{li})$ selon le signe du bloc l et le type de y_{li}, ou encore la valeur de vérité $v(y_{li})$ selon le signe du bloc l) rattachées à la résolution de la nouvelle variable y_{li} (lignes 41–48), EXPAND permet de traiter le cas des constantes booléennes (lignes 54–71) qui sont par définition stables, et peuvent donc être ajoutées à l'ensemble B_{li} de variables stabilisées du bloc l du SEB multibloc en cours de résolution. Si cette variable était également variable destinataire d'une transition interbloc, elle n'a plus besoin de faire partie de l'ensemble R_{li}, car son insertion dans B_{li} permettra de propager sa valeur stable à tous ses prédécesseurs y compris ceux interblocs. Pour la même raison, le compteur de transitions interblocs non stabilisées exp_req_{li} est décrémenté d'une transition si la transition $x_{ki} \rightarrow y_{li}$ en cours d'expansion est interbloc et locale (ligne 60).

L'appel à STABILIZE vérifie l'invariant : $d(y_{li}) = \emptyset$, car, à ce niveau de l'algorithme, y_{li} ne peut être qu'une constante booléenne, donc le seul prédécesseur est le prédécesseur courant x_{kj}.

Afin de propager ultérieurement la valeur de y_{li} à ses prédécesseurs distants, l'ensemble B_{li} est mis à jour avec la variable y_{li}.

Si y_{li} n'est pas une variable stabilisée (c'est-à-dire $\neg stable(y_{li})$), seule sa dépendance avec le prédécesseur x_{kj} est mise à jour (ligne 73).

5.3.3 Stabilisation

Les opérations de stabilisation peuvent être séparées en deux parties : stabilisation active et stabilisation passive, toutes deux décrites dans la primitive STABILIZATION de la figure 5.9. Cette primitive ayant une priorité plus élevée que EXPANSION, l'effet global de l'algorithme MB-DSOLVE sera de stabiliser tous les blocs retournés par la primitive PENDINGSTABILIZATION avant d'effectuer des explorations de bloc, et ainsi de privilégier la propagation arrière de valeurs stables jusqu'à la variable d'intérêt.

Stabilisation passive Comme énoncé dans la section 5.2, une variable peut être stabilisée directement par un de ses successeurs stables, ou bien l'être par détection de stabilité du bloc auquel elle appartient.

La stabilisation passive du bloc l (l étant la valeur retournée par la primitive PENDINGSTABILIZATION) est décrite aux lignes 35–38 de la figure 5.9. Elle respecte l'invariant suivant : $W_{li} = \emptyset \wedge exp\_req_{li} = 0 \wedge \neg inactif_{li}$, c'est-à-dire la file BFS W_{li} est vide, il n'y a plus aucune requête d'expansion exp_req_{li} ni de stabilisation R_{li}, et toutes les variables du bloc l stabilisées activement ont été propagées et enlevées de B_{li}. La résolution du bloc l est donc dans un état d'inactivité qui doit être transmis au superviseur (lignes 37–38) pour mettre à jour sa vision globale de l'inactivité des nœuds travailleurs

```
 1 procedure STABILIZATION is
 2   if B_li ≠ ∅ ∨ (term_li ∧ R_li ≠ ∅) then
 3     /* stabilisation active */
 4     while B_li ≠ ∅ ∨ (term_li ∧ R_li ≠ ∅) do
 5       if B_li ≠ ∅ then
 6         y_li := choose(B_li);  B_li := B_li \ {y_li}
 8       else
 9         y_li := choose(R_li);  R_li := R_li \ {y_li}
11       endif;
12       forall w_kj ∈ d(y_li) ∧ (B_li ≠ ∅ ∨ k ≠ l)
13         ∧¬term_b(x)i ∧ ¬stable(w_kj) do
14         if k ≠ l then
15           /* (Figure 5.8) */
16           BLOCKACTIVITYTEST (k)
17         endif;
18         if h(w_kj) = i then
19           /* stabilisation locale du */
20           /* sommet prédécesseur w_kj */
21           if k ≠ l then
22             exp_req_ki := exp_req_ki − 1
23           endif;
24           STABILIZE(w_kj, y_li)
25         else
26           /* stabilisation distante du */
27           /* sommet prédécesseur w_kj */
28           /* (Figure 5.10) */
29           SENDING(Evl(w_kj, y_li), h(w_kj))
30         endif
31       endfor;
32       d(y_li) := ∅
33     endwhile
34   else
35     /* stabilisation passive */
36     inactif_li := true;
37     SEND(Idl(l, sent_li − recv_li),
38          superviseur)
39   endif
40 end

41 procedure STABILIZE(w_ki, y_lj) is
42   if ((L(w_ki) = ∨) ∧ v(y_lj))∨
43     ((L(w_ki) = ∧) ∧ ¬v(y_lj)) then
44     s(w_ki) := y_lj;
45     c(w_ki) := 0;  stable(w_ki) := true
46   else
47     c(w_ki) := c(w_ki) − 1;
48   endif;
49   if stable(w_ki) then
50     B_ki := B_ki ∪ {w_ki};
51     if w_ki ∈ R_ki then
52       R_ki := R_ki \ {w_ki}
53     endif;
54     term_b(x)i := initiator_i∧
55                   stable(x)
56   endif
57 end
```

Figure 5.9: Propagation des valeurs des variables stables (activement ou passivement) du bloc courant l

sur le bloc l, et éventuellement déclencher une tentative de détection distribuée de terminaison du bloc l. La stabilisation passive est donc la détection de portions de blocs non stabilisés activement (c'est-à-dire dont toutes les variables n'ont pas été stabilisées activement) mais ne dépendant plus de la résolution d'autres blocs pour avoir une valeur finale stable.

Remarque 5.5
Les lignes 35–38 de la figure 5.9 ont pour invariant d'appel : $\neg inactif_{li} = true$, car le nœud travailleur courant i doit être actif sur la résolution du bloc l avant de pouvoir appeler cette partie de la primitive STABILIZATION, qui transmettra au nœud superviseur le statut d'inactivité du nœud i, pour détection éventuelle de la terminaison du bloc l sur l'ensemble des nœuds. ■

Stabilisation active Dans la figure 5.9, la stabilisation active est décrite entre les lignes 2–33. Elle consiste en une propagation arrière des variables stabilisées du bloc l. Une variable est stabilisée activement par propagation de la valeur d'un de ses successeurs soit au sein du même bloc ($B_{li} \neq \emptyset$) soit d'un autre bloc ($R_{li} \neq \emptyset$).

Remarque 5.6
Les ensembles B_{li} et R_{li} sont des ensembles disjoints quelques soient les valeurs de $l \in [1..N]$ et $i \in [1..P]$. L'invariant suivant est donc préservé lors de la résolution, et notamment lors d'opérations de stabilisation et de propagation de valeurs stables : $\forall\ k, l \in [1..N],\ i, j \in [1..P] \mid B_{ki} \cap R_{lj} = \emptyset$. ■

Après avoir extrait une variable stabilisée à propager de l'ensemble de variables stabilisées B_{li} ou R_{li} (invariant : $\forall y_{li} \in B_{li} \cup R_{li},\ c(y_{li}) = 0 \wedge y_{li} \in S_{li} \wedge h(y_{li}) = i$), l'algorithme itère sur les prédécesseurs de cette variable afin de leur transmettre la valeur stable de la variable propagée. Cette stabilisation arrière des variables dépendantes de y_{li} est soumise à quelques restrictions : elle est restreinte aux prédécesseurs qui ne sont pas eux-mêmes stables ($\neg stable(w_{kj})$) et qui sont soit d'un bloc différent de celui de la variable stabilisée, soit stabilisés activement ($B_{li} \neq \emptyset \vee k \neq l$). Ces restrictions sont faites afin d'éviter de réaliser des propagations inutiles, respectivement vers des variables déjà stabilisées ou appartenant au même bloc déjà stabilisé passivement. L'opération de propagation arrière peut engendrer une réactivation de la résolution du bloc k (ligne 16), lorsque la variable propagée appartient à un bloc $l \neq k$, et, qui plus est, a été stabilisée passivement. Pour ce faire, une détection de reprise d'activité est alors envoyée au nœud superviseur, qui détient ainsi une vision à jour de l'activité du système. La propagation de la valeur stable de la variable propagée peut se faire localement (ligne 24) ou à distance par l'intermédiaire de message *Evl* (ligne 29).

L'opération de mise à jour de la valeur de w_{ki} lors de la propagation de la valeur de y_{lj} est faite par la primitive STABILIZE dont l'invariant d'invocation est : $\neg stable(w_{ki}),\ w_{ki} \in S_{ki},\ stable(y_{lj}),\ et\ y_{lj} \in E(w_{ki})$, c'est-à-dire, la propagation de la valeur de la variable stable y_{lj}, successeur de w_{ki}, n'est faite que si la variable w_{ki} appartenant à l'ensemble des variables visitées S_{ki}, n'a pas déjà été stabilisée.

Bien que la propagation d'une valeur *vraie* (*resp.* $fausse$) sur une variable de plus grand fixe (*resp.* plus petit) point fixe n'ait aucun effet sur sa valeur finale qui est déjà initialisée à *vraie* (*resp.* $fausse$), propager toutes les variables stabilisées permet d'accélérer la stabilisation globale de la résolution (lignes 42–48).

Lors de la détection de stabilité d'une variable (lignes 42–43), il est convenant de sauvegarder l'information nécessaire pour la génération future du diagnostic, à savoir :

- quelque soit le point fixe μ ou ν, les variables $\vee$ (*resp.* $\wedge$) peuvent être stabilisées directement par un seul successeur de valeur *vraie* (*resp.* $fausse$), (ligne 44), et
- dans les autres cas, tous les successeurs sont obligatoirement parcourus, et font partie du diagnostic.

Si la propagation de la valeur de y_{lj} stabilise activement (c'est-à-dire directement) w_{ki}, alors l'ensemble de variables stabilisées non propagées (B_{ki}) rattaché au bloc k est mis à jour (ligne 50) avec la variable w_{ki} pour une propagation future de sa valeur, ainsi que l'ensemble de dépendances interblocs non stabilisées (R_{ki}) de bloc destinataire k (lignes 51–53).

Cette opération STABILIZE permet également de déclencher la terminaison anticipée de la résolution du SEB multibloc lorsque la variable d'intérêt a été stabilisée (lignes 54–55). Seul le nœud initiateur peut détecter la terminaison anticipée.

5.3.4 Réception

La réception de messages en attente sur les canaux de communication est l'opération de plus haute priorité dans l'algorithme MB-DSOLVE. Ce choix a été fait pour minimiser les temps d'inactivité localement à chaque nœud en favorisant la réception de données locales et tâches d'exécution. Deux heuristiques au niveau de la réception des messages sont possibles :

- lecture d'au plus un message à chaque appel de réception, et traitement des ensembles de stabilisation et d'expansion dès que possible, ou
- lecture et traitement de tous les messages reçus quelque soit le nœud émetteur à chaque appel de réception, afin de mettre à jour en bloc les différents ensembles de variables utiles pour la résolution du SEB.

```
 1 procedure ReadWorker(msg_i, sender_i) is
 2   case msg_i is
 3     Exp(x_{ksender_i}, y_{li}) →
 4         recv_{ki} := recv_{ki} + 1;
 5         if k ≠ l then
 6           Q_{li} := Q_{li} ∪ {(x_{ksender_i}, y_{li})}
 7         else
 8           /* (Figure 5.8) */
 9           BlockActivityTest(l);
10           /* (Figure 5.7) */
11           Expand(x_{ksender_i}, y_{li})
12         endif
13     Evl(x_{ki}, y_{lsender_i}) →
14         recv_{ki} := recv_{ki} + 1;
15         /* (Figure 5.8) */
16         BlockActivityTest (k);
17         if k ≠ l then
18           exp_req_{ki} := exp_req_{ki} − 1
19         endif
20         if ¬stable(x_{ki}) then
21           /* (Figure 5.9) */
22           Stabilize(x_{ki}, y_{lsender_i})
23         endif
```

```
24     Ack(k, stamp) →
25         if inactif_{ki} then
26           term_{ki} := false;
27           Send(Ack(k, stamp),
28                superviseur)
29         endif
30     Trm(k) →
30         term_{ki} := true;
31         forall x_{ki} ∈ S_{ki} do
32           stable(x_{ki}) := true
33         endfor
34   endcase
35 end

36 procedure Sending(msg_j, node_j) is
37   while ¬ ISend(msg_j, node_j))
38         ∧ ¬term_{b(x)i} do
39     if IReceive(msg_i, sender_i) then
40       /* (Figure 5.10) */
41       ReadWorker(msg_i, sender_i)
42     else
43       WaitEvent({0..P}, , sender_i)
44     endif
45   endwhile
46   if ¬term_{b(x)i} then
47     sent_{ki} := sent_{ki} + 1
48   endif
49 end
```

Figure 5.10: Interprétation des messages reçus (colonne gauche) et émission non-bloquante de messages (colonne droite)

Ces heuristiques n'apparaissent pas dans l'algorithme, mais la dernière est celle utilisée en pratique dans l'algorithme MB-DSolve au niveau de la primitive ReadWorker (voir figure 5.10).

Cette primitive décrit le traitement d'un message reçu msg_i.

Les messages Exp et Evl sont envoyés entre les nœuds travailleurs afin de réaliser respectivement les parcours de :

- expansion en avant des variables nouvelles (c'est-à-dire l'expansion Bfs à partir d'un des successeurs y_{li} distants du nœud $x_{ksender_i}$) du bloc l dont l'invariant d'appel est : $x_{ksender_i} \in S_{ksender_i} \wedge y_{li} \in E_{ksender_i} \wedge h(y_{li}) = i \wedge sender_i \neq i \wedge sender_i \neq superviseur$. Si la transition à explorer est interbloc, elle est mise en attente dans Q_{li} (ligne 6), et
- propagation en arrière de valeurs stables (c'est-à-dire la stabilisation du nœud x_{ki}, prédécesseur du nœud distant $y_{lsender_i}$) sur le bloc k dont l'invariant d'appel est : $x_{ki} \in S_{ki} \wedge y_{lsender_i} \in E(x_{ki}) \wedge stable(y_{lsender_i}) \wedge sender_i \neq i \wedge sender_i \neq superviseur$. Cette propagation n'est réalisée que si x_{ki} n'a pas déjà été stabilisée entre-temps, sinon elle devient inutile. Le test de reprise d'activité (ligne 16) permet de relancer la résolution d'un bloc auparavant inactive, ainsi que de tenir compte d'un changement possible de la valeur de vérité de x_{ki} après propagation de la valeur de $y_{lsender_i}$.

Les messages Ack et Trm sont échangés entre le nœud superviseur et les nœuds travailleurs dans le cadre de la détection distribuée de terminaison d'un bloc k. Le message Ack permet de confirmer par chaque nœud travailleur au nœud superviseur de son état inactif sur la résolution du bloc k. Un

estampillage du message Ack permet d'identifier les différentes tentatives de détection de terminaison réalisées par le nœud superviseur. Si le nœud local a repris son activité entre-temps, aucune communication n'est réalisée avec le nœud superviseur car un message de remise en activité (Act) a déjà été envoyé. Le message Trm, quant à lui, permet d'indiquer la terminaison effective de la résolution courante du bloc k, ainsi que la stabilisation passive des variables explorées mais non stabilisées activement dans le bloc k. Le message de détection de terminaison est envoyé selon une topologie en étoile par le nœud superviseur vers les nœuds travailleurs.

Le protocole de détection distribuée de terminaison d'un bloc est le même que celui décrit dans la section 2.4.2 et la figure 2.7 du chapitre 2. La distinction vient de la multitude de blocs à résoudre pour un même SEB et le fait que plusieurs parties d'un bloc peuvent être amenées à être résolues indépendamment, en impliquant pour chacune d'entre elles une détection distribuée de terminaison. Pour cela, chacun des messages de détection de terminaison est indexé par le numéro du bloc auquel il se réfère, ainsi que les compteurs de messages $recv_{ki}$ et $send_{ki}$ nécessaires à la détection de terminaison. Les compteurs n'énumèrent plus que les messages utilisés pour la résolution, à savoir Exp et Evl. En effet, notre algorithme de détection de terminaison n'a pas besoin de connaître le nombre de messages de contrôle échangés, car il vérifie la stabilité de l'inactivité détectée à deux reprises, ce qui permet à des messages de contrôle éventuels, comme Act et Idl, d'arriver sur le nœud superviseur avant que l'inactivité soit confirmée. De plus, les blocs ayant des dépendances interblocs, ne doivent plus dépendre des autres blocs avant de pouvoir être détectés stables (c'est-à-dire avant de détecter la terminaison de la résolution du bloc en question). Ces blocs nécessitent alors une stabilisation des variables sources des transitions interblocs. Pour cela, un compteur (exp_req) est utilisé pour indiquer le nombre de dépendances interblocs restant à stabiliser.

5.3.5 Estimation de la complexité

La complexité de la résolution d'un SEB dépend de la taille du graphe booléen correspondant devant être parcouru. MB-DSOLVE étant une extension de DSOLVE gardant le mécanisme de parcours (exploration avant et propagation arrière) du graphe booléen, les bornes de complexité de l'algorithme DSOLVE s'appliquent donc à l'algorithme MB-DSOLVE. La complexité en temps d'exécution de MB-DSOLVE reste linéaire en la taille du graphe booléen exploré et stabilisé ($O(|V|+|E|)$), et la complexité en espace mémoire de la résolution multibloc est également inchangée, égale à $O(|V|+|E|)$, à cause des dépendances $d(y_{li})$ sauvegardées pour la propagation des valeurs finales de variables stabilisées.

Pour conclure cette section, l'exécution de MB-DSOLVE sur un bloc d'équations booléennes est identique à l'exécution de DSOLVE sur ce même bloc (et présentée sur la figure 2.5 du chapitre 2). L'extension présente dans MB-DSOLVE consiste principalement en la gestion des dépendances entre plusieurs blocs d'un SEB multibloc, et en sa capacité à détecter différents niveaux de terminaison lors de la résolution de ce SEB.

5.4 Extension de l'agent central de coordination et de terminaison

La fonction principale du nœud superviseur reste inchangée par rapport à la version de SUPERVISOR utilisée avec la résolution monobloc.

Ce nœud cherche à détecter la terminaison de la résolution du SEB. Dans le cas des SEBs multiblocs, sa fonction doit être étendue à la détection de terminaison de chaque résolution (partielle) de bloc réalisée au cours du calcul de la variable d'intérêt du SEB multibloc donné en paramètre.

5.4.1 Tâches de coordination

La primitive décrivant le comportement du nœud superviseur est nommée SUPERVISOR (voir la figure 5.11). Cette primitive prend trois paramètres en entrée :

- b : **Bool** est le numéro du bloc de la variable booléenne d'intérêt,
- $N : \mathbb{N}$ est le nombre de blocs d'équations booléennes du SEB multibloc devant être résolu, et
- $P : \mathbb{N}$ est le nombre de nœuds travailleurs effectuant la résolution distribuée.

Cette primitive se décompose en deux parties :

- recherche du plus petit index de bloc devant être stabilisé passivement (lignes 8–13) et détection de terminaison de ce bloc (lignes 14–41). Cette dernière opération effectue le traitement de la terminaison distribuée, une fois l'inactivité d'un bloc k détectée. Elle est elle-même décomposée en deux sous-parties, qui correspondent aux deux phases du protocole de détection distribuée de terminaison : la diffusion de la confirmation d'inactivité de la résolution sur le bloc k (lignes 18–23) et la diffusion de la transmission de la détection de terminaison (lignes 24–31), toutes deux aux nœuds travailleurs. L'état $STOP$ correspond à la terminaison de la résolution du bloc, quant à l'état $CONFIRM$, il sert à confirmer l'inactivité de la résolution distribuée sur le bloc considéré.
- détection d'inactivité d'un bloc du SEB en cours de résolution (lignes 37–40). Cette partie n'est réalisée que lorsque tous les blocs ont un statut $DETECT$ ou $TERM$. Seule une réception bloquante de message est alors nécessaire pour éventuellement relancer une détection de terminaison sur un bloc particulier.

Remarque 5.7
Lors de la diffusion d'informations concernant la terminaison d'un bloc, le nœud superviseur peut être amené à ne plus pouvoir envoyer de messages sur le nœud travailleur destinataire. Notre algorithme étant asynchrone, le nœud superviseur peut réaliser des réceptions de messages en attendant que les tampons de réception du nœud travailleur distant obtiennent de l'espace libre, ce qui correspond à de l'attente active sur l'émission ou la réception de messages. Ceci est permis grâce au statut du nœud superviseur qui maintient l'information nécessaire à la continuation de la diffusion préalablement interrompue. Par exemple, lors de la diffusion de la confirmation de détection de terminaison (lignes 19–20), le nœud superviseur reste dans l'état $CONFIRM$ tant que tous les messages de confirmation n'ont pas été diffusés (lignes 19–23). ■

La primitive READSUPERVISOR (voir figure 5.11 (colonne droite)) est utilisée pour traiter les différents messages reçus sur le nœud superviseur. La primitive BLOCKINACTIVITYTEST (voir figure 5.11 (colonne droite)) sert quant à elle pour la détection d'inactivité par le nœud superviseur au vu de l'état global de la résolution d'un bloc donné à un instant donné (en l'occurrence, sur réception d'un message Idl (ligne 55) ou d'un message Ack (ligne 63)). Si l'inactivité est détectée, une phase de confirmation de cette inactivité sera nécessaire pour pouvoir conclure à la terminaison de la résolution du bloc en question. Sinon, l'état $term\_statut_k$ du nœud superviseur reste inchangé jusqu'à la réception de nouveaux messages Idl ou Ack.

Une fois la détection de terminaison du SEB réalisée, si le nœud superviseur n'a pas encore reçu de message contenant la solution, alors le seul message qui peut lui parvenir la contient et provient du nœud initiateur (lignes 43–46).

5.4.2 Détection de terminaison distribuée

Le protocole de DDT décrit dans la section 2.4.2 et la figure 2.7 du chapitre 2, et utilisé dans l'algorithme DSOLVE, peut fonctionner avec très peu de changements dans le cadre d'une résolution

```
 1 function SUPERVISOR(b, N, P) : ℕ is
 2   /* (Figure 5.3) */
 3   DISTRIBUTED DATA DEFINITION
 4   /* (Figure 5.6) */
 5   INITIALIZATION
 6
 7   while term_statut_b ≠ TERM do
 8     k := 1;
 9     while (k ≤ N)∧
10           (term_statut_k ≠ CONFIRM)∧
11           (term_statut_k ≠ STOP) do
12       k := k + 1
13     endwhile;
14     if k ≤ N then
15       /* stabilisation passive */
16       /* du bloc k */
17       case term_statut_k is
18         CONFIRM →
19           while (bcast_node_k ≤ P) ∧
20                 ISEND(Ack(k, stamp_k),
21                 bcast_node_k) do
22             bcast_node_k := bcast_node_k + 1;
23           endwhile;
24         STOP →
25           while (bcast_node_k ≤ P) ∧
26           ISEND(Trm(k), bcast_node_k) do
27             bcast_node_k := bcast_node_k + 1
28           endwhile;
29           if (bcast_node_k > P) then
30             term_statut_k := TERM
31           endif
32       endcase;
33       if IRECEIVE(msg, sender) then
34         READSUPERVISOR(msg, sender)
35       endif
36     else
37     /* détection d'inactivité */
38     /* d'un bloc du SEB */
39       RECEIVE(msg, sender);
40       READSUPERVISOR(msg, sender)
41     endif
42   endwhile
43   if solution = ⊥ then
44     RECEIVE(msg, sender)
45   endif;
46   return (solution)
47 end
```

```
48 procedure READSUPERVISOR(msg, sender) is
49   case msg is
50     Act(k) →
51       nb_inactif_k := nb_inactif_k − 1;
52       total_msg_k := total_msg_k−
53                      nb_msg_k(sender);
54       term_statut_k := DETECT
55     Idl(k, t) →
56       nb_msg_k(sender) := t;
57       total_msg_k := total_msg_k+
58                      nb_msg_k(sender);
59       nb_inactif_k := nb_inactif_k + 1;
60       if term_statut_k = DETECT then
61         BLOCKINACTIVITYTEST(k)
62       endif
63     Ack(k, t) →
64       if t = stamp_k then
65         if term_statut_k = DETECT then
66           BLOCKINACTIVITYTEST(k)
67         elsif term_statut_k =
68                  CONFIRM then
69           nb_ack_k := nb_ack_k + 1;
70           if (nb_ack_k = P) ∧ (total_msg_k +
71             nb_msg_k(superviseur) = 0) then
72             term_statut_k := STOP;
73             bcast_node_k := 1;
74             nb_ack_k := 0
75         endif
76       endif
77     Sol(b) →
78       solution := b;
79       if (term_statut_k ≠ STOP)∧
80           (term_statut_k ≠ TERM) then
81         term_statut_k := STOP;
82         bcast_node_k := 1
83       endif
84   endcase
85 end

86 procedure BLOCKINACTIVITYTEST(k)
87   if (nb_inactif_k = P) ∧ (total_msg_k+
88          nb_msg_k(superviseur) = 0) then
89     term_statut_k := CONFIRM;
90     bcast_node_k := 1;
91     nb_ack_k := 0;
92     stamp_k := stamp_k + 1
93   endif
94 end
```

Figure 5.11: Supervision et détection de la terminaison multibloc du SEB

multibloc.

Chaque fois que le superviseur détecte l'inactivité d'un bloc (lignes 87–93), il initie une vague de confirmation de détection d'inactivité de ce bloc. Si tous les nœuds confirment l'inactivité de la résolution sur le bloc donné, le nœud superviseur peut alors conclure et diffuser que la résolution du bloc considéré est terminée. Cette DDT est utilisée pour chaque bloc, et peut finalement détecter la terminaison du bloc de la variable d'intérêt si celle-ci n'est pas stabilisée activement entre-temps.

Chaque phase de confirmation d'inactivité de la résolution sur un bloc donné k, est indexée par un estampillage $stamp_k$ (voir ligne 20) qui est incrémenté (ligne 92) chaque fois que la résolution d'un bloc passe de l'état inactif à l'état actif. Lors de la réception d'un message *Ack* de confirmation d'inactivité, il est alors utile de comparer l'estampillage véhiculé par le message avec l'estampillage courant rattaché au bloc considéré (ligne 64). Si l'estampillage est dépassé, le message est ignoré car des messages de reprise d'activité *Act* ou d'autres tentatives de détection de terminaison (*Ack*) ont du être envoyés entre-temps. S'il ne reçoit pas de message, le nœud superviseur repasse alors dans un état de test de l'inactivité du système (lignes 39–40).

5.5 Implémentation

5.5.1 Algorithme MB-DSOLVE

MB-DSolve a été implémenté (15 000 lignes de code C) à partir de DSolve dans l'environnement Open/Cæsar. L'implémentation du comportement des nœuds travailleurs (algorithme MB-DSolve, 9 000 lignes de code C) est deux fois plus large que celle utilisée dans le cas monobloc (algorithme DSolve, 4 700 lignes de code C). Il en est de même pour le comportement du nœud superviseur qui double (on passe de 1 200 lignes à 2 000 lignes de code C) dans la version multibloc. L'interface d'utilisation de MB-DSolve, quant à elle, est conforme en tout point avec l'interface que propose Cæsar_Solve, la bibliothèque de résolution séquentielle de Sebs [Mat03a, Mat06] qui est intégrée à la boîte à outils Cadp. Ainsi, MB-DSolve peut être utilisé immédiatement par tous les outils de vérification basés sur DSolve (comme par exemple Bisimulator et Tau_Confluence). Enfin, notre implémentation repose sur la bibliothèque de communication Cæsar_Network, qui est également utilisée par DSolve.

5.5.2 Expérimentations de MB-DSOLVE

Nous avons testé notre implémentation de MB-DSolve en l'expérimentant au travers des applications existantes connectées à DSolve. Ces tests ont consisté à effectuer de nouveau toutes les expériences réalisées pour analyser la performance de DSolve, notamment dans le cadre des applications Bisimulator et Tau_Confluence. Les conditions de test ont été respectées puisque l'architecture matérielle est restée la même que pour l'analyse de DSolve, à savoir une grappe de Pcs Xeon.

Bien que n'ayant pas effectué de mesures de performances brutes (en réduisant au maximum le rôle de l'application, comme par exemple en connectant le générateur aléatoire de Sebs avec MB-DSolve), les résultats obtenus avec les outils de vérification sont très encourageants puisqu'ils donnent des performances identiques à celles obtenues avec DSolve, ce dernier étant optimisé pour les problèmes de vérification se traduisant en Sebs monoblocs. Ainsi, nous obtenons une accélération et une efficacité très bonne, de même qu'un passage à l'échelle performant, et des coûts en mémoire et en communication quasi constants avec l'augmentation du nombre de nœuds travailleurs. Les surcoûts liés à l'extension de DSolve à la résolution multibloc ne sont donc pas visibles lors de l'exécution

de MB-DSolve sur des problèmes monoblocs, ce qui laisse présager de bonnes performances sur des Sebs contenant deux blocs ou plus.

——— ⋆ ——— ⋆ ——— ⋆ ———

En résumé, nous avons développé une extension conservatrice de DSolve, appelée MB-DSolve, qui permet de résoudre efficacement à la volée et en distribué aussi bien des Sebs monoblocs que des Sebs multiblocs. Les résultats présentés dans ce chapitre, bien que très prometteurs, ne suffisent pas à eux-mêmes pour donner un bon aperçu de la capacité et de l'efficacité de MB-DSolve à résoudre des Sebs multiblocs. A cette fin, nous considérons dans la suite de cette étude deux problèmes de vérification pouvant être traduits en Sebs multiblocs : la vérification par évaluation de formules logiques et la génération automatique de tests de conformité.

Chapitre 6

Application 3 : Evaluation de formules logiques

Dans l'approche de vérification sur les modèles traitée jusqu'à présent, seules les *spécifications comportementales* d'un programme à vérifier ont été considérées, notamment dans le cadre de la comparaison par équivalence de deux automates (voir le chapitre 3). Cette approche consiste à exprimer les propriétés comportementales attendues du programme P_1 sous forme d'un autre programme P_2 (dans le domaine des protocoles de communication, P_1 et P_2 sont souvent appelés *protocole* et *service*, respectivement), et de comparer les STEs M_1 et M_2 correspondants, selon des relations d'équivalence ou de préordre.

Cependant, il existe aussi d'autres approches pour exprimer les propriétés à vérifier, notamment les *spécifications logiques*. Les spécifications logiques bénéficient des avantages de la modularité et de l'abstraction [MP90], et permettent à ce titre une meilleure caractérisation des propriétés globales des programmes (par exemple, l'exclusion mutuelle entre plusieurs processus). En outre, il est souvent plus aisé de traduire les propriétés attendues (exprimées informellement en langage naturel) sous forme de formules de logique temporelle que sous forme de programmes.

——— ⋆ ——— ⋆ ——— ⋆ ———

Ce chapitre se propose d'appliquer le nouvel algorithme MB-DSOLVE au problème d'évaluation à la volée de formules de logiques temporelles exprimées en μ-calcul modal d'alternance 1 afin de distribuer ce problème de vérification sur un ensemble de machines interconnectées [JM06].

La section 6.1 introduit les différents formalismes utilisés dans la littérature pour les spécifications logiques temporelles, en détaillant plus finement le formalisme adopté dans cette étude, à savoir le μ-calcul modal d'alternance 1. Le mécanisme d'évaluation à la volée est ensuite décrit dans la section 6.2 et sa traduction sous forme de SEBs multiblocs définie et illustrée par des exemples dans la section 6.3. La connexion de la bibliothèque générique de résolution de SEBs multiblocs, basée sur l'algorithme MB-DSOLVE, est exposée dans la section 6.4, où l'outil d'évaluation distribuée à la volée qui en résulte est analysé selon plusieurs critères de performance.

6.1 Expression des formules de logiques temporelles

Il existe une multitude de logiques temporelles qui ont été définies et étudiées pour spécifier et vérifier les propriétés des programmes parallèles. Nous présentons ici une synthèse des principaux résultats existants, en mettant l'accent sur l'expressivité comparée des diverses logiques temporelles, sur leur adaptation aux différents types de programmes parallèles, ainsi que sur la complexité de leur évaluation sur des modèles finis, ce qui permet d'avoir une description globale et unifiée de ces formalismes tout en soulignant leurs différences respectives.

6.1.1 Logiques : modales, arborescentes, régulières et de point fixe

Pour un état de l'art sur l'expressivité des logiques temporelles basées sur actions, nous reportons le lecteur intéressé au chapitre 1, section 2 de la thèse de Radu Mateescu [Mat98], ainsi qu'à l'article [Mat03b].

Logiques temporelles basées sur états : Les logiques temporelles permettant d'exprimer des propriétés sur les états des programmes ont été intensivement étudiées dans la littérature. Ces logiques, interprétées sur des structures de Kripke, sont généralement partagées entre deux classes :

- Les logiques du *temps linéaire* comme LTL (*Linear Temporal Logic*) [Lam80] et PTL (*Propositional Temporal Logic*) [MP92], permettant d'exprimer des propriétés portant sur les chemins individuels (issus de l'état initial) du programme et
- Les logiques du *temps arborescent* comme CTL (*Computational Tree Logic*) et CTL* [CES86, EH86], permettant d'exprimer des propriétés portant sur les arbres d'exécution (issus de l'état initial) du programme.

Les différentes combinaisons des opérateurs de ces logiques permettent d'exprimer diverses classes de propriétés, telles que : les propriétés de *sûreté* [Lam83] (état indésirable d'un programme jamais atteint), les propriétés de *vivacité* [AS85] (état désirable *potentiellement* ou *inévitablement* atteignable), et les propriétés d'*équité* [GPSS80, AFK87] (états du programme répétés indéfiniment).

Concernant le pouvoir expressif des logiques temporelles, il varie selon que la logique est linéaire ou arborescente. Les premières sont plus adaptées aux propriétés d'équité, car elles permettent d'imbriquer les modalités de chemins. En revanche, certaines logiques arborescentes (comme CTL) permettent de caractériser des chemins individuels du modèle grâce à la présence explicite des quantificateurs.

Un autre aspect important est l'existence d'algorithmes efficaces pour évaluer les formules temporelles sur les modèles. L'évaluation est plus efficace pour les logiques arborescentes car elle est linéaire en taille du modèle et en taille de la formule, alors qu'elle est exponentielle en taille de la formule pour les logiques linéaires (comme LTL).

Enfin, dans le domaine des algèbres de processus, les logiques temporelles utilisées avec prédilection sont les logiques arborescentes, car elles permettent de distinguer plus finement la structure de branchement du comportement des programmes parallèles.

Logiques temporelles basées sur actions : Certaines classes de langages parallèles, comme les algèbres de processus CCS (*Calculus of Communicating Systems*), CSP (*Communicating Sequential Processes*) et ACP (*Algebra of Communicating Processes*) ou les langages de description LOTOS et μCRL, ont une sémantique opérationnelle définie en termes d'*actions* effectuées par le programme. Les propriétés des programmes décrits dans ces langages doivent donc naturellement porter sur l'ordonnancement des actions dans le temps. Les logiques arborescentes sur actions en sont une

représentation. L'une d'entre elles, appelée ACTL (*Action CTL*) [NFGR91], présente un intérêt pratique puisqu'il s'agit d'une des premières logiques arborescentes basées sur actions qui autorise l'expression de propriétés de sûreté et de vivacité relativement élaborées, tout en ayant un fragment significatif adéquat avec une équivalence faible (celle de branchement) entre STEs. Une extension naturelle d'ACTL est ACTL* (*Action CTL**) [NV90] qui constitue l'équivalent de CTL* en termes d'actions. D'autres logiques sur actions ont été définies afin de caractériser les propriétés des programmes décrits dans les algèbres de processus : il s'agit de logiques *modales*, munies d'opérateurs permettant d'exprimer la *possibilité* et la *nécessité*. La logique HML (*Hennessy-Milner*) [HM85] peut être considérée comme le représentant standard de cette classe de logiques.

Il existe également différentes logiques permettant de combiner les deux types de propriétés interprétées sur les états et sur les actions. Une méthode pour spécifier des propriétés sur états et sur actions est d'étendre les logiques basées sur états, comme CTL ou CTL*, avec des formules sur actions. Cette approche a été utilisée dans la logique LTAC [QS83] qui contient, outre des opérateurs similaires à ceux de CTL, des prédicats sur actions. Une autre classe de logique permettant d'exprimer des propriétés sur états et sur actions, sont les logiques *dynamiques* (introduites pour l'étude des programmes à commandes gardées), qui permettent d'exprimer les propriétés sous forme d'expressions régulières construites sur un vocabulaire d'actions élémentaires. PDL (*Propositional Dynamic Logic*) [FL79] figure parmi les logiques dynamiques les plus utilisées.

Logiques temporelles avec opérateurs de point fixe : Certaines modalités temporelles peuvent être caractérisées comme plus petits et plus grands points fixes de fonctionnelles monotones [EC80]. Ainsi sont apparues des logiques temporelles contenant des *opérateurs de point fixe*, capables d'exprimer une large gamme de modalités en utilisant un ensemble restreint d'opérateurs primitifs.

Le *μ-calcul modal* [Koz83] est une logique très expressive, étudiée intensivement dans la littérature. Ce formalisme peut être vu comme une extension de la logique modale HML (qui est donc une sous-logique du μ-calcul) avec des opérateurs de point fixe.

Enfin, nous pouvons mentionner d'autres logiques avec des opérateurs de point fixe qui ont été proposées dans la littérature, comme la logique de Dicky [Dic86], la logique STL (*Synchronization Tree Logic*) [GS86] ou la logique HML avec récursion [Lar88].

Logiques temporelles étendues avec des valeurs : Les logiques modales et temporelles mentionnées jusqu'ici sont interprétées sur des modèles contenant des actions atomiques et/ou des constantes propositionnelles associées aux états. En particulier, les logiques interprétées sur actions sont adaptées pour spécifier les propriétés des programmes décrits dans des algèbres de processus "pures" (*pure* CCS, *basic* LOTOS, etc.), c'est-à-dire ne contenant pas de communication avec passage de valeurs. En revanche, ces logiques ne sont pas adaptées pour les langages de description basées sur valeurs (*full* CCS, *full* LOTOS, μCRL, etc.). Ainsi, de nouvelles logiques temporelles ont été introduites, et sont capables d'exprimer des propriétés sur les valeurs manipulées dans les programmes à vérifier. Parmi ces logiques, on peut citer la *logique modale de μCRL* [GvV94]. Il s'agit d'une logique arborescente basée sur actions, dédiée à la description des propriétés temporelles des programmes μCRL. Cette logique peut être vue comme une extension d'ACTL* avec des opérateurs sur le passé. Les propriétés sur valeurs sont exprimées au moyen de variables typées, de prédicats d'égalité sur les termes et de quantification du premier ordre. Cette logique est interprétée sur les modèles des programmes μCRL, qui sont des systèmes de transitions ayant des actions structurées (noms de canaux de communication et listes de valeurs échangées) mais ne contenant pas d'informations sur les états.

Parmi les logiques temporelles décrites jusqu'à présent, les logiques de point fixe, telles que le μ-calcul modal ou la logique de Dicky, sont les plus expressives, mais également les plus complexes à utiliser,

pouvant être considérées comme des langages de bas niveau, tels des langages d'assemblage, pour la description des propriétés temporelles. En revanche, les logiques arborescentes, comme ACTL, ou régulières, comme PDL, bien que moins expressives que les logiques de point fixe, fournissent une gamme d'opérateurs plus intuitifs, qui rendent la description de propriétés plus aisée. L'expérience pratique tend à montrer que la meilleure solution consiste à utiliser une logique temporelle suffisamment expressive – donc comportant des opérateurs de point fixe – et offrant également des opérateurs arborescents et réguliers (des exemples de telles logiques sont μ-ACTL, le μ-calcul régulier). Nous nous concentrons par la suite sur une version restreinte du μ-calcul modal, le μ-calcul modal d'alternance 1.

6.1.2 μ-calcul d'alternance 1

Le μ-calcul modal [Koz83] est une logique très expressive, qui peut remplacer la plupart des logiques temporelles définies dans la littérature. Dans sa version originelle [Koz83], le μ-calcul est interprété sur des structures de Kripke $\mathcal{K} = (S, P, L, T)$. Les formules modales sont définies sur un vocabulaire d'actions atomiques, chaque action a induisant une relation binaire sur S qui relie les états de début et de fin de l'exécution de a. Néanmoins, de façon similaire à d'autres logiques temporelles, comme PDL, il est possible de définir la sémantique des formules du μ-calcul modal sur des STEs $M = (Q, A, T, q_0)$.

La syntaxe et la sémantique de la variante du μ-calcul modal que nous considérons ici sont définies dans le tableau 6.1.

Type	Syntaxe		Sémantique		
Formules sur actions	α	$::= a$	$[\![a]\!]$	$=$	$\{a\}$
		$\mid \mathsf{F}$	$[\![\mathsf{F}]\!]$	$=$	$\emptyset$
		$\mid \mathsf{T}$	$[\![\mathsf{T}]\!]$	$=$	A
		$\mid \neg\alpha_1$	$[\![\neg\alpha_1]\!]$	$=$	$A \setminus [\![\alpha_1]\!]$
		$\mid \alpha_1 \vee \alpha_2$	$[\![\alpha_1 \vee \alpha_2]\!]$	$=$	$[\![\alpha_1]\!] \cup [\![\alpha_2]\!]$
		$\mid \alpha_1 \wedge \alpha_2$	$[\![\alpha_1 \wedge \alpha_2]\!]$	$=$	$[\![\alpha_1]\!] \cap [\![\alpha_2]\!]$
Formules sur états	φ	$::= \mathsf{F}$	$[\![\mathsf{F}]\!]\rho$	$=$	$\emptyset$
		$\mid \mathsf{T}$	$[\![\mathsf{T}]\!]\rho$	$=$	Q
		$\mid \neg\varphi_1$	$[\![\neg\varphi_1]\!]\rho$	$=$	$Q \setminus [\![\varphi_1]\!]\rho$
		$\mid \varphi_1 \vee \varphi_2$	$[\![\varphi_1 \vee \varphi_2]\!]\rho$	$=$	$[\![\varphi_1]\!]\rho \cup [\![\varphi_2]\!]\rho$
		$\mid \varphi_1 \wedge \varphi_2$	$[\![\varphi_1 \wedge \varphi_2]\!]\rho$	$=$	$[\![\varphi_1]\!]\rho \cap [\![\varphi_2]\!]\rho$
		$\mid \langle\alpha\rangle\,\varphi_1$	$[\![\langle\alpha\rangle\,\varphi_1]\!]\rho$	$=$	$\{q \in Q \mid \exists q \xrightarrow{a} q' \in T.\, a \in [\![\alpha]\!] \wedge q' \in [\![\varphi_1]\!]\rho\}$
		$\mid [\alpha]\,\varphi_1$	$[\![[\alpha]\,\varphi_1]\!]\rho$	$=$	$\{q \in Q \mid \forall q \xrightarrow{a} q' \in T.\, a \in [\![\alpha]\!] \Rightarrow q' \in [\![\varphi_1]\!]\rho\}$
		$\mid X$	$[\![X]\!]\rho$	$=$	$\rho(X)$
		$\mid \mu X.\,\varphi_1$	$[\![\mu X.\,\varphi_1]\!]\rho$	$=$	$\bigcap\{U \subseteq Q \mid [\![\varphi_1]\!]\rho[U/X] \subseteq U\}$
		$\mid \nu X.\,\varphi_1$	$[\![\nu X.\,\varphi_1]\!]\rho$	$=$	$\bigcup\{U \subseteq Q \mid U \subseteq [\![\varphi_1]\!]\rho[U/X]\}$

Table 6.1: Syntaxe et sémantique du μ-calcul modal

La logique contient deux types d'entités : les formules sur actions (notées α) et les formules sur états (notées φ), qui permettent respectivement de caractériser des sous-ensembles d'actions et d'états du STE M.

- Les formules sur actions sont identiques à celles de la logique HML. Elles sont construites au-dessus du vocabulaire des actions $a \in A$ au moyen des opérateurs booléens standard. Des opérateurs booléens dérivés sont définis de la manière habituelle :

$$\begin{aligned} \alpha_1 \Rightarrow \alpha_2 &= \neg\alpha_1 \vee \alpha_2 \\ \alpha_1 \Leftrightarrow \alpha_2 &= (\alpha_1 \Rightarrow \alpha_2) \wedge (\alpha_2 \Rightarrow \alpha_1) \end{aligned}$$

La sémantique d'une formule α (partie droite du tableau 6.1) sur le STE M est définie par l'interprétation $[\![\alpha]\!] \subseteq A$, qui dénote le sous-ensemble d'actions du STE satisfaisant α.

- Les formules sur états sont construites sur un ensemble $\mathcal{X}$ de variables propositionnelles (notées X, X_1, X_2, etc.) – dénotant des sous-ensembles d'états du STE – au moyen d'opérateurs booléens, modaux de *possibilité* ($\langle\alpha\rangle\,\varphi$) et de *nécessité* ($[\alpha]\,\varphi$), de plus petit ($\mu X.\,\varphi$) et de plus grand point fixe ($\nu X.\,\varphi$).
  La sémantique d'une formule φ (partie droite du tableau 6.1) est définie par l'interprétation $[\![\varphi]\!]\rho \subseteq Q$ (où $\rho : \mathcal{X} \to 2^Q$ est un environnement associant des sous-ensembles d'états à toutes les variables propositionnelles libres dans φ) qui dénote le sous-ensemble d'états du STE satisfaisant φ dans l'environnement ρ.
  La notation $\rho[U/X]$ représente un environnement identique à ρ, excepté pour la variable X, à laquelle est associé l'ensemble d'états U.
  Les opérateurs booléens ont la sémantique habituelle définie en termes d'opérations ensemblistes. La modalité de nécessité $[\alpha]\,\varphi$ dénote les états dont toutes les transitions successeurs étiquetées par des actions satisfaisant α mènent à des états satisfaisant φ. Les deux opérateurs modaux sont duaux ($[\alpha]\,\varphi = \neg\,\langle\alpha\rangle\,\neg\varphi$) et monotones (si $\varphi_1 \Rightarrow \varphi_2$ alors $\langle\alpha\rangle\,\varphi_1 \Rightarrow \langle\alpha\rangle\,\varphi_2$ et $[\alpha]\,\varphi_1 \Rightarrow [\alpha]\,\varphi_2$). Les opérateurs $\mu X.\,\varphi$ et $\nu X.\,\varphi$ dénotent respectivement la plus petite et la plus grande solution de l'équation $X = \varphi$ dans un environnement ρ, c'est-à-dire les points fixes correspondants de la fonctionnelle $\Phi_\rho : 2^Q \to 2^Q$ définie par $\Phi_\rho(U) = [\![\varphi]\!]\rho[U/X]$. La correction de leur interprétation définie dans le tableau 6.1 est assurée par la monotonie syntaxique des formules φ (ce qui implique la monotonie des fonctionnelles Φ_ρ, c'est-à-dire $U_1 \subseteq U_2 \Rightarrow \Phi_\rho(U_1) \subseteq \Phi_\rho(U_2)$ pour tout environnement ρ et ensembles d'états U_1, $U_2 \subseteq Q$) et par le théorème de Tarski [Tar55].

Exemple 6.1
Nous pouvons illustrer le pouvoir expressif du μ-calcul modal par quelques exemples de propriétés :

"absence de blocage sur l'état courant"	:	$\langle\mathsf{T}\rangle\,\mathsf{T}$
"accessibilité potentielle d'une action a"	:	$\mu X.\,\langle a\rangle\,\mathsf{T} \vee \langle\mathsf{T}\rangle\,X$
"pas d'action RECV avant une action SEND"	:	$\nu X.\,[RECV]\,\mathsf{F} \wedge [\neg SEND]\,X$

■

L'expressivité des formules φ du μ-calcul modal augmente avec leur *alternance* [EL86], qui mesure le degré de récursion mutuelle entre les opérateurs de plus petit et de plus grand point fixe à l'intérieur de φ. Intuitivement, les formules d'alternance 1 n'admettent pas de récursion mutuelle entre les opérateurs μ et ν, celles d'alternance 2 admettent deux opérateurs μ et ν mutuellement récursifs, etc. Il est important de noter que la notion d'alternance est indépendante du degré d'imbrication des opérateurs de point fixe.

Exemple 6.2
Considérons la propriété suivante :

"après chaque action SEND, il y aura potentiellement une action RECV"

Cette propriété exprimée en langage naturel se traduit par la formule de μ-calcul suivante :

$$\nu X.\,[SEND]\,(\mu Y.\,\langle RECV\rangle\,\mathsf{T} \vee \langle\mathsf{T}\rangle\,Y) \wedge [\mathsf{T}]\,X$$

Cette formule est d'alternance 1, puisque la variable de plus grand point fixe X n'apparaît pas dans la sous-formule $\mu Y.\,\langle RECV\rangle\,T \vee \langle T\rangle\,Y$ définissant la variable de plus petit point fixe Y ("Y n'appelle pas X").

En revanche, la formule :

$$\nu X.\, \mu Y.\, (\langle SEND \rangle\, X \vee \langle \neg RECV \rangle\, Y)$$

exprimant l'existence d'une séquence infinie d'actions SEND au détriment d'actions RECV, bien qu'elle comporte également deux opérateurs de point fixe imbriqués, est d'alternance 2, puisque la variable de plus grand point fixe X et celle de plus petit point fixe Y s'appellent mutuellement. ■

La notion d'alternance induit une hiérarchie de fragments du μ-calcul modal, notés Lμ_1, Lμ_2, ..., chaque fragment Lμ_i contenant les formules d'alternance i. Le fragment Lμ_1 (également appelé *sans alternance*) est le moins expressif de la hiérarchie, dont l'autre extrême est le μ-calcul modal complet Lμ, qui réunit tous les fragments Lμ_i pour $i \geq 0$. Cependant, les fragments Lμ_1 et Lμ_2 sont suffisamment expressifs pour permettre la traduction de logiques arborescentes comme ACTL ou régulières comme PDL et PDL-Δ.

Lors du choix d'une logique temporelle en tant que formalisme de spécification de propriétés, un aspect important à prendre en compte est la complexité de la vérification (évaluation d'une formule sur un modèle STE), mesurée habituellement par rapport à la taille de la formule (nombre d'opérateurs) et du STE (nombre d'états et de transitions). En effet, même si la plupart des formules ne dépassent pas une dizaine d'opérateurs, les STEs des applications réelles vont souvent au-delà de la dizaine de millions d'états, ce qui impose l'utilisation de logiques ayant une complexité de vérification polynomiale afin d'obtenir des temps de réponse utiles en pratique. Les fragments Lμ_1 et Lμ_2 du μ-calcul (formules d'alternance 1 et 2) constituent de bons compromis entre expressivité de la logique et complexité de leur évaluation, car ils permettent la description de propriétés relativement élaborées et ont une complexité d'évaluation linéaire, respectivement quadratique, en taille du STE et de la formule. Pour ces raisons, nous choisissons les formules du μ-calcul d'alternance 1 comme formalisme de description de propriétés logiques dans le cadre de cette étude où nous souhaitons développer un évaluateur distribué à la volée de propriétés temporelles.

6.2 Evaluation locale des formules de logiques temporelles

La spécification logique d'un programme parallèle consiste habituellement en une liste de formules de logique temporelle, chacune exprimant une propriété de bon fonctionnement du programme. Le problème de vérification basée sur les modèles est de déterminer si ces formules sont satisfaites par le modèle du programme, ce qui revient à *évaluer* la valeur de vérité de chaque formule sur le modèle. On distingue généralement deux classes de méthodes d'évaluation des formules temporelles sur un modèle :

Les méthodes globales utilisent une représentation explicite du modèle (le modèle devant être généré complètement à l'aide d'un compilateur *avant* de commencer l'évaluation de la formule). Ces méthodes fonctionnent selon le schéma indiqué dans la figure 6.1.

Les formules de logique temporelle constituant la spécification du programme sont vérifiées grâce à un évaluateur, qui fournit en sortie leur valeur de vérité, éventuellement accompagnée d'un diagnostic.

Considérons un STE $M = (Q, A, T, q_0)$ et une formule sur états φ. Conformément à la définition donnée en section 6.1.2, la vérification de φ sur M revient à évaluer la valeur de vérité de φ sur q_0, c'est-à-dire à déterminer si $q_0 \in [\![\varphi]\!]$. La méthode globale consiste à calculer d'abord l'interprétation $[\![\varphi]\!]$ et à tester ensuite si l'état initial q_0 lui appartient.

Remarque 6.1
L'interprétation des opérateurs booléens et des modalités $\langle \alpha \rangle\, \varphi$ et $[\alpha]\, \varphi$ peut être calculée directement

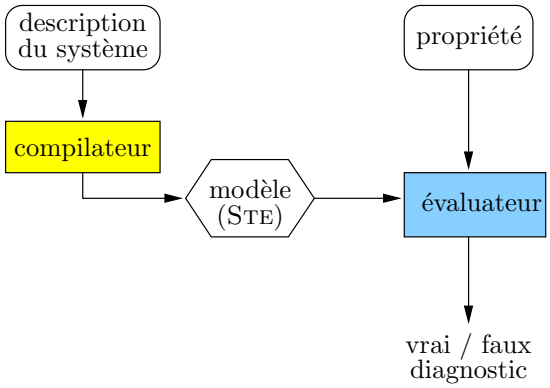


Figure 6.1: Principe de l'évaluation globale

à partir de leur définition donnée dans le tableau 6.1. L'interprétation des opérateurs de point fixe $\mu X.\varphi$ et $\nu X.\varphi$ dans un environnement ρ peut être calculée en utilisant la variante constructive du théorème de Tarski. Dans le cas des STEs finis, les fonctionnelles $\Phi_\rho : 2^Q \to 2^Q$ sont non seulement monotones, mais également *continues* :

$$\Phi_\rho(\bigcup_{k\geq 0} U_k) = \bigcup_{k\geq 0} \Phi_\rho(U_k) \text{ pour toute suite croissante } U_k \subseteq U_{k+1}$$
$$\Phi_\rho(\bigcap_{k\geq 0} W_k) = \bigcap_{k\geq 0} \Phi_\rho(W_k) \text{ pour toute suite décroissante } W_{k+1} \subseteq W_k$$

Cela permet de calculer le plus petit et le plus grand point fixe Φ_ρ de manière itérative, comme $\bigcup_{k\geq 0} \Phi_\rho^k(\emptyset)$ et respectivement $\bigcap_{k\geq 0} \Phi_\rho^k(Q)$. ■

L'avantage des méthodes globales est de factoriser l'effort de construction du modèle, qui n'est généré qu'une seule fois même si plusieurs formules de logique temporelle doivent être évaluées (ce qui en pratique est souvent le cas). En outre, du fait que le modèle est entièrement connu (nombre d'états, de transitions, etc.), les méthodes globales conduisent à des algorithmes ayant une bonne complexité dans le pire des cas.

Cependant, étant donné que les méthodes de vérification globales nécessitent la construction du STE dans sa totalité *avant* de commencer l'évaluation d'une formule, cette approche peut s'avérer prohibitive en termes de mémoire pour les STEs de grande taille. Dans certains cas – notamment, lorsqu'une propriété de vivacité (*resp.* de sûreté) est (*resp.* n'est pas) satisfaite par le modèle – il n'est pas nécessaire de générer entièrement le modèle afin de déterminer la valeur de vérité d'une formule. Cette observation est à la base du développement des méthodes d'évaluation locales.

Les méthodes locales aussi appelées à la volée (*on-the-fly*), utilisent une représentation implicite du modèle (le modèle étant généré *pendant* l'évaluation de la formule). De ce fait, les méthodes locales peuvent être considérées plus sophistiquées que celles globales, puisqu'elles permettent de calculer le même résultat sans avoir une connaissance *a priori* de la totalité du STE. Leur fonctionnement peut être décrit selon le schéma indiqué dans la figure 6.2.

A la différence des méthodes globales, l'évaluation des formules a lieu simultanément avec la génération du modèle : l'environnement de vérification contient un compilateur pour le langage des programmes à vérifier et un évaluateur pour les formules temporelles, ces deux outils fonctionnant conjointement.

Les méthodes locales présentent l'avantage de ne pas générer complètement le modèle lorsqu'une propriété à vérifier peut être validée sur une portion seulement du modèle. De ce fait, elles sont

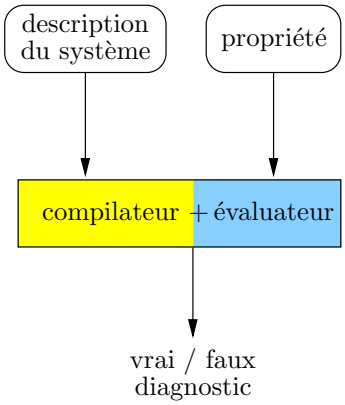


Figure 6.2: Principe de l'évaluation locale

utiles dans les premières phases du processus de conception, quand les erreurs sont susceptibles d'être les plus fréquentes. En revanche, comme le modèle n'est pas entièrement connu au moment de l'évaluation des formules, les méthodes locales conduisent à des algorithmes ayant une complexité légèrement supérieure (d'un facteur logarithmique en taille du modèle) par rapport aux algorithmes globaux correspondants.

Deux manières efficaces de traiter le problème de la vérification par évaluation locale de formules logiques sont les graphes de jeux [Sti95] et les systèmes d'équations booléennes [And94]. Bien que ces deux formalismes aient une traduction directe l'un vers l'autre dans le contexte de la vérification de formules de logiques temporelles, les seuls résultats présentés pour d'autres problèmes de vérification, tels que la comparaison par équivalences, sont théoriques, et très peu de réalisations pratiques ont été effectuées. Ainsi, il est difficile de comparer ces deux formalismes sur une large gamme de problèmes de vérification. Des algorithmes locaux séquentiels pour le μ-calcul modal basés sur la théorie des jeux ont été proposés en [SS98]. D'autres basés sur les systèmes d'équations booléennes ont été proposés dans [VL94, And94, Mad97, MS03]. Le premier algorithme d'évaluation distribuée à la volée de formules logiques en Lμ_1, basé sur les graphes de jeux et supportant également la génération interactive de diagnostic, a été récemment proposé en [BLW02]. Leur méthode consiste à utiliser la représentation canonique en arbre de la formule de μ-calcul modal dans le but de calculer la représentation en graphe de cette formule, sur laquelle ils construisent le graphe de jeux en ne considérant que les parties intéressantes du STE donné en entrée. Cette méthode, bien qu'efficace en pratique et pouvant traiter des STEs de très grande taille (non vérifiable par les méthodes séquentielles classiques) [HLL04], s'adapte mal, de part son mécanisme, à d'autres problèmes de vérification, comme la comparaison par équivalences.

L'approche que nous adoptons, et que nous considérons plus naturelle pour la vérification locale des formules du μ-calcul, consiste à reformuler le problème en une résolution d'un système d'équations booléennes. Cette approche a été prouvée efficace à la fois en séquentiel et en distribué sur les problèmes de comparaison par équivalences (voir le chapitre 3) et de réduction d'ordre partiel (voir le chapitre 4).

Notre travail de thèse concerne ici la vérification par évaluation distribué e à la volée de propriétés temporelles exprimées en μ-calcul modal d'alternance 1 des programmes décrits dans des langages parallèles (tels que LOTOS). Nous allons d'une part détailler la traduction du problème en termes de SEBs (section 6.3) et, d'autre part, développer des algorithmes d'évaluation à la volée distribuée associés à partir desquels nous montrerons l'efficacité de cette approche sur un ensemble d'applications

réelles (section 6.4).

6.3 Traduction de l'évaluation de formules logiques en termes de SEBs

Soit un STE $M = (Q, A, T, q_0)$ et une formule de point fixe $\sigma X.\,\varphi$ avec ($\sigma \in \{\mu, \nu\}$). Une méthode naturelle de vérification locale [CS91b, And94] consiste à reformuler le problème $q_0 \in [\![\sigma X.\,\varphi]\!]$ comme la résolution de la variable X_{q_0} d'un système d'équations booléennes $\{X_q \stackrel{\sigma}{=} (\varphi)_q\}_{q \in Q}$ où chaque variable booléenne X_q indique si l'état q satisfait la variable propositionnelle X. Cette reformulation sous forme de SEBs est définie dans le tableau 6.2.

Ainsi, le principe de l'évaluation locale de formule logique en μ-calcul modal sur un STE M peut se résumer à : $q_0 \models X$ si et seulement si X_{q_0} est vraie.

Afin de réaliser un tel calcul, nous utilisons les SEBs d'alternance 1 qui permettent de traduire le μ-calcul modal d'alternance 1 [CS91a, And94, Mad97].

φ	$(\varphi)_q$	φ	$(\varphi)_q$
F	F	T	T
$\varphi_1 \vee \varphi_2$	$(\varphi_1)_q \vee (\varphi_2)_q$	$\varphi_1 \wedge \varphi_2$	$(\varphi_1)_q \wedge (\varphi_2)_q$
$\langle a \rangle \varphi_1$	$\bigvee_{q \xrightarrow{a} q'} (\varphi_1)_{q'}$	$[a]\,\varphi_1$	$\bigwedge_{q \xrightarrow{a} q'} (\varphi_1)_{q'}$
X	X_q	$\sigma X.\,\varphi_1$	X_q

Table 6.2: Traduction des sous-formules $\sigma X.\,\varphi'$ de φ sous forme de SEB

La traduction standard sous forme de SEB de la formule φ d'alternance 1, étant donné le STE M, se déroule comme suit [CS91b, And94, MS03] :

1. Premièrement, des variables propositionnelles supplémentaires sont introduites aux places appropriées de φ pour assurer que chaque sous-formule $\sigma X.\,\varphi'$ (où $\sigma \in \{\mu, \nu\}$) de φ, φ' contient un seul opérateur booléen ou modal (ceci est nécessaire pour obtenir des formules purement disjonctives ou conjonctives en partie droite du SEB résultant).
2. Ensuite, le SEB est construit de manière ascendante, en créant un bloc d'équations de signe σ pour chaque sous-formule close $\sigma X.\,\varphi'$ de φ. La condition d'alternance 1 assure qu'une fois les sous-formules de point fixe de $\sigma X.\,\varphi'$ ont été traduites, toutes les variables restantes de $\sigma X.\,\varphi'$ sont de signe σ.
3. Enfin, chaque sous-formule close $\sigma X.\,\varphi'$ est traduite dans un bloc $\{X_q \stackrel{\sigma}{=} (\varphi')_q\}_{q \in Q}$, où les variables X_q expriment qu'un état q satisfait X et les parties droites des formules booléennes $(\varphi')_q$ sont obtenues en utilisant les traductions montrées dans le tableau 6.2.

La résolution de la variable X_{q_0} est effectuée en évaluant la formule $(\varphi)_{q_0}$ sur la partie droite de l'équation, ce qui peut nécessiter les valeurs d'autres variables X_q, obtenues par évaluation des formules $(\varphi)_q$ sur la partie droite de leurs équations, et ainsi de suite ; le calcul s'arrête lorsque la valeur de vérité de la variable d'intérêt X_{q_0} a pu être déterminée. Il est important de noter que l'évaluation des formules en partie droite des équations (définies dans le tableau 6.2) permet de parcourir les transitions du STE en *avant*, ce qui autorise la construction du STE à la volée pendant la résolution.

Exemple 6.3
Nous illustrons la traduction de l'évaluation de formules logiques en termes de SEBs avec un exemple de STE (montré en figure 6.3) sur lequel nous souhaitons vérifier la propriété de μ-calcul modal d'alternance 1 suivante (décrite précédemment dans l'exemple 6.2) :

$$\nu X.\,[SEND]\,(\mu Y.\,\langle RECV\rangle\,\mathsf{T} \vee \langle \mathsf{T}\rangle\,Y) \wedge [\mathsf{T}]\,X$$

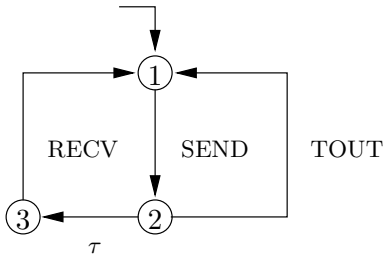


Figure 6.3: Un exemple de STE avec 3 états et 4 transitions étiquetées

La propriété logique est premièrement traduite en une spécification en logique HML avec récursion [Lar88] qui contient deux blocs d'équations modales :

$$\begin{array}{rcl}\{X & \stackrel{\nu}{=} & [SEND]\,Y \wedge [\mathsf{T}]\,X\} \\ \{Y & \stackrel{\mu}{=} & \langle RECV\rangle\,\mathsf{T} \vee \langle \mathsf{T}\rangle\,Y\}\end{array}$$

Nous pouvons ainsi observer que cette description est bien d'alternance 1 car elle ne contient pas de dépendances cycliques entre ses deux blocs d'équations X et Y.

Ensuite, chaque bloc d'équation modale est convertit en un bloc d'équation booléenne en le projetant sur chacun des états du STE (et en utilisant le tableau 6.2) :

$$\begin{array}{rcl}\{X_q & \stackrel{\nu}{=} & (\bigwedge_{q \stackrel{SEND}{\rightarrow} q'} Y_{q'}) \wedge (\bigwedge_{q \stackrel{a}{\rightarrow} q'} X_{q'})\},\ \text{avec } a \in A \\ \{Y_q & \stackrel{\mu}{=} & (\bigvee_{q \stackrel{RECV}{\rightarrow} q'} \mathsf{T}) \vee (\bigvee_{q \stackrel{a}{\rightarrow} q'} Y_{q'})\},\ \text{avec } a \in A\end{array}$$

Ainsi, chaque état $q \in Q$ se retrouve dans deux équations booléennes, qui définissent les variables X_q et Y_q. Une variable booléenne X_q (*resp.* Y_q) est vraie si et seulement si l'état q satisfait la variable propositionnelle X (*resp.* Y).

Bien que les algorithmes de résolution (comme MB-DSOLVE ou encore $A1$ et $A2$) utilisent la description implicite donnée ci-dessus, nous donnons, à titre d'illustration, le système d'équations booléennes explicite et son graphe booléen correspondant dans la figure 6.4.

L'évaluation à la volée de la formule logique en μ-calcul modal initiale revient à calculer la valeur de la variable X_{q_0} en appliquant l'algorithme de résolution locale de SEB. Le calcul de la variable X_{q_0} (dans l'exemple présent, il s'agit de la variable X_1) nécessite l'évaluation de $(\bigwedge_{1 \stackrel{SEND}{\rightarrow} q'} Y_{q'}) \wedge (\bigwedge_{1 \stackrel{a}{\rightarrow} q'} X_{q'}) = Y_2 \wedge X_2$. Il dépend donc des valeurs des variables Y_2 et X_2 qui à leur tour peuvent être calculées ; la variable Y_2 nécessite l'évaluation de $Y_3 \vee Y_1$; et finalement Y_3 conduit à l'évaluation de $(\bigvee_{3 \stackrel{RECV}{\rightarrow} q'} \mathsf{T}) \vee (\bigvee_{3 \stackrel{a}{\rightarrow} q'} Y_{q'}) = \mathsf{T}$. En substituant la variable Y_3 dans la formule $Y_3 \vee Y_1$, la variable Y_2 devient vraie également. Ce mécanisme de propagation arrière des substitutions rend vraies toutes les formules intermédiaires s'il s'agit de disjonctions (comme dans le cas du bloc Y). En conséquence, la variable d'intérêt X_{q_0} ne dépend plus que de la valeur de X_2 : la variable X_2 nécessite l'évaluation de $X_1 \wedge X_3$. Les variables d'un plus grand point fixe étant initialisées à T, la valeur de X_2 ne dépend plus que de la valeur de X_3, car X_1, de valeur initiale T, ne modifie pas la valeur finale de la conjonction définissant X_2. Finalement, X_3 conduit à l'évaluation de X_1. Toutes les variables du bloc X étant sur un même cycle de dépendances, la valeur finale de X_3 est la valeur initiale de X_1, qui est T. En

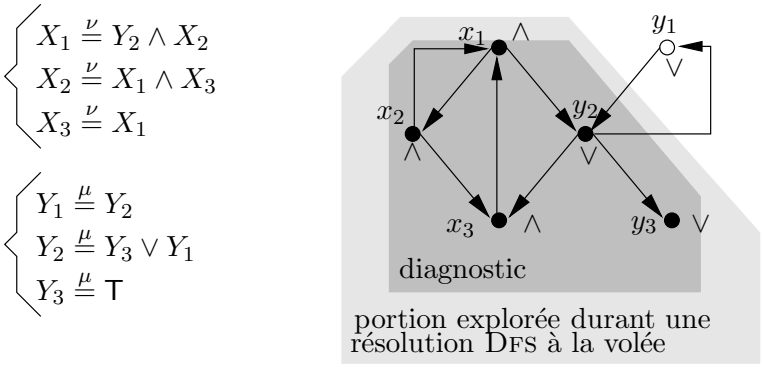


Figure 6.4: Le SEB résultant, son graphe booléen, et une résolution locale pour x_1

propageant les substitutions, la variable d'intérêt X_{q_0} est vraie, signifiant que la formule est satisfaite par le STE. Le calcul a nécessité l'exploration d'une partie seulement du STE puisque, par exemple, la variable Y_1 n'a pas dû être évaluée. Cet exemple de résolution locale DFS à partir de la variable x_1 est illustré dans la partie *grise claire* de la figure 6.4. ■

Ce type de SEB peut être résolu par des algorithmes séquentiels généraux $A1$ et $A2$ donnés en [Mat03a] et distribué MB-DSOLVE (voir le chapitre 5). On peut noter que la procédure de traduction permet de construire le STE à la volée pendant la résolution du SEB. Cependant, lorsque le STE M et/ou la formule φ ont une structure particulière, le SEB peut être simplifié dans le but de rendre applicable des algorithmes séquentiels spécialisés tels que $A3$ et $A4$ [Mat03a] :

- Cas acyclique : Lorsque M est acyclique et que φ est *gardée* (c'est-à-dire tout appel récursif de variable propositionnelle dans φ tombe dans la portée d'un opérateur modal), la formule peut être simplifiée dans le but d'avoir seulement des opérateurs de plus petit point fixe, menant à un SEB acyclique contenant un seul bloc [Mat02]. Si cette condition est rencontrée, alors l'algorithme efficace en mémoire $A3$ peut être utilisé pour effectuer l'évaluation de la formule en μ-calcul modal.
- Cas disjonctif/conjonctif : Lorsque φ est une formule de CTL [CES86], ACTL [NFGR91] ou de PDL [FL79], le SEB résultant après traduction est en forme disjonctive ou conjonctive. Il est alors possible de traduire les opérateurs correspondants en μ-calcul modal d'alternance 1 [Mat03a]. L'algorithme efficace en mémoire $A4$ [Mat03a] peut alors être utilisé pour évaluer des formules en CTL, ACTL et PDL. Cela couvre la plupart des besoins pratiques, puisque de nombreuses propriétés intéressantes peuvent être exprimées en utilisant les opérateurs de ces logiques. A titre d'exemple, la propriété utilisée dans l'exemple 6.2 peut être décrite en PDL par la formule $[\mathsf{T}^*.SEND]\langle\mathsf{T}^*.RECV\rangle\mathsf{T}$.

Nous nous intéressons dans la suite de cette étude aux graphes booléens généraux, pour des STEs quelconques et sur lesquels on souhaite vérifier à la volée une formule de μ-calcul modal d'alternance 1. Pour cela, l'algorithme MB-DSOLVE peut être utilisé dans le cas d'une résolution distribuée du SEB sous-jacent, ainsi que les algorithmes généraux séquentiels $A1$ et $A2$.

6.4 Réalisation et applications

De nouveau, l'architecture modulaire des outils de vérification de CADP est mise à profit. La connexion de l'évaluateur séquentiel existant EVALUATOR, qui permet la vérification à la volée de formules de

logique temporelle en μ-calcul modal d'alternance 1 étendue avec des modalités PDL contenant des expressions régulières sur les séquences de transitions, avec le solveur distribué se fait naturellement. Dans un premier temps, nous présentons brièvement la réalisation de cette connexion, pour ensuite l'évaluer selon plusieurs facteurs (temps, mémoire, et passage à l'échelle) vis-à-vis des versions séquentielles optimisées et de la version distribuée existante dans la littérature.

6.4.1 Développement de la connexion à EVALUATOR 3.5

L'outil EVALUATOR [MS03, Mat03a, Mat06] (voir la figure 6.5) a été développé au sein de CADP [GLM02] en utilisant l'environnement générique OPEN/CÆSAR [Gar98] pour l'exploration à la volée de STEs.

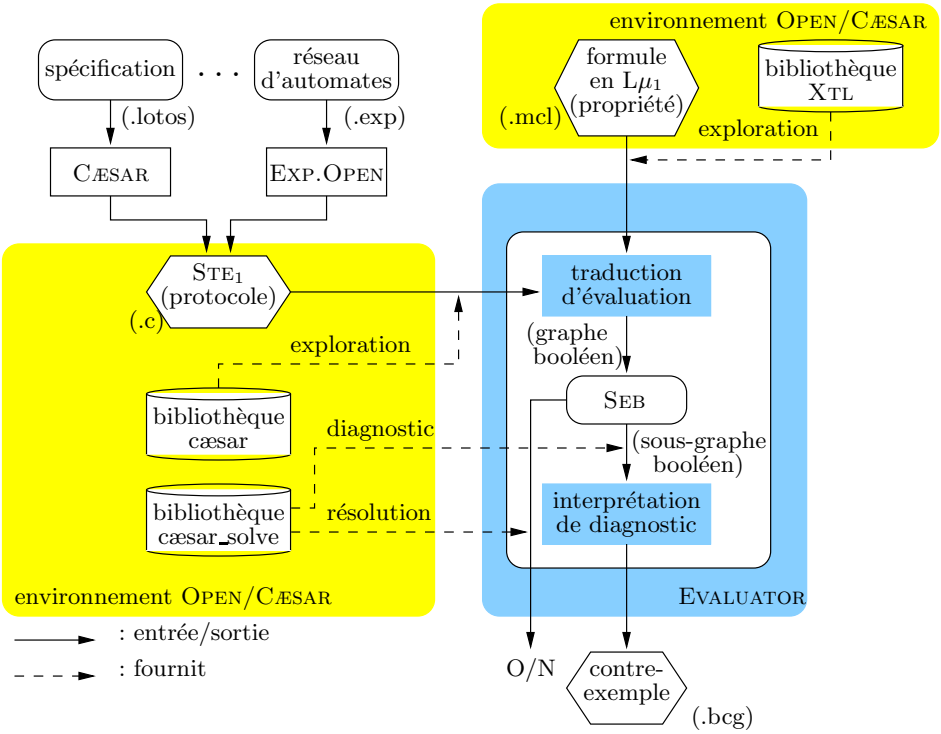


Figure 6.5: Architecture de l'outil EVALUATOR

EVALUATOR (version 3.5) consiste en deux parties : une partie avant, responsable de la traduction du STE et de la formule de μ-calcul modal d'alternance 1 en un SEB (pouvant être multibloc) et de l'interprétation des contre-exemples fournis par la résolution de SEB en termes du STE donné en entrée ; et une partie arrière, qui effectue la résolution du SEB et sert comme moteur de vérification. La version séquentielle [Mat03a, Mat06] et distribuée de l'outil sont obtenues en utilisant comme partie arrière soit l'algorithme séquentiel de la bibliothèque CÆSAR_SOLVE [Mat03a, Mat06], soit l'algorithme MB-DSOLVE, respectivement. Les successeurs d'une variable booléenne X_q dénotant la satisfaisabilité de la formule de μ-calcul modal sur l'état q du STE, sont calculés par la partie avant d'EVALUATOR — qui est appelée séquentiellement et indépendamment sur chaque nœud travailleur — par une exploration en avant du STE en commençant par q, selon la traduction en SEB de la formule (voir le tableau 6.2). Cette architecture modulaire ne pénalise pas les performances. Nous

allons voir dans la suite de ce document que la version séquentielle d'Evaluator [Mat03a, Mat06] se compare favorablement même à des algorithmes spécifiques à la volée distribués implémentés dans UppDMC [HLL04].

Dans le but de comparer les versions distribuée et séquentielle (basée sur l'algorithme $A1$ de parcours en profondeur d'abord (Dfs) de Cæsar_Solve) d'Evaluator, nous avons effectué un ensemble d'expériences étendues sur différents STEs obtenus à partir d'études de cas de Cadp et de la suite de tests Vlts [VLT03]. Seules les dix plus larges expériences du Vlts sont montrées dans cette section. Il est à noter que pour obtenir une image précise des performances, nous excluons les coûts (fixes) des activités dépendantes du système (chargement du code sur les nœuds distants, initialisation des connexions, copie des fichiers de Ste), et nous gardons seulement les coûts (variables) de la résolution distribuée et de la détection de terminaison.

Nous avons effectué chacune des expériences dix fois. Chaque point de chaque courbe correspond à la moyenne des huit valeurs correspondant aux mesures obtenues, en excluant les valeurs maximale et minimale.

6.4.2 Analyse de performances

Dans un premier temps, nous définissons les propriétés logiques ainsi que les STEs que Evaluator distribué va évaluer. Nous donnons ensuite une analyse de performance détaillée de l'outil par rapport à des solutions séquentielle et distribuée existantes. Enfin, nous décrivons quelques améliorations susceptibles de rendre notre solution encore plus performante.

Propriétés La validation de la version distribuée d'Evaluator s'est faite en deux phases : tests de fonctionnalités sur la base de démonstrations de la boîte à outils Cadp, et tests de performance sur les exemples du Vlts.

- La base de démonstrations de Cadp[1] contient actuellement 39 études de cas permettant d'illustrer les différents outils présents dans la plate-forme. Une dizaine d'entre elles font appel à l'outil Evaluator avec lequel des propriétés de μ-calcul modal sont évaluées sur un protocole généralement décrit dans une algèbre de processus, ou sous forme d'automates communicants. Nous avons exécuté avec succès la version distribuée d'Evaluator sur ces dix études de cas afin de comparer les réponses des évaluations de chaque propriété avec les réponses renvoyées par l'outil séquentiel. Ces réponses, traduisant la valeur de vérité du Seb sous-jacent au problème de vérification, sont des critères d'évaluation du fonctionnement comportemental de notre algorithme distribué MB-DSolve. Nous avons ainsi pu tester des propriétés simples d'interblocage et de famine, ainsi que des propriétés beaucoup plus complexes, comme l'inévitabilité et la sûreté.

  Les exemples, tels qu'ils sont présentés dans la base de démonstrations, ne prennent que quelques secondes pour être évalués par les versions séquentielle et distribuée d'Evaluator, et par conséquent ne permettent pas une évaluation de performances très poussée. Néanmoins, avec l'exemple *demo_02*, décrivant le protocole du Bit Alterné en Lotos, il est possible de paramétrer la description du protocole en fixant le nombre de messages différents échangés à 256 au lieu de 5. Le Ste résultant a une taille de 6 067 712 états et 19 505 146 transitions. Comme notre méthode d'évaluation est locale, ce Ste ne nécessite pas d'être construit *a priori* pour être évalué. Seule la partie nécessaire de celui-ci sera explorée dynamiquement. Nous avons ainsi pu évaluer, sur le Ste implicite correspondant, la propriété d'inévitabilité

[1] `http ://www.inrialpes.fr/vasy/capd/demos.html`

suivante, qui vérifie que "après chaque action PUT, il est inévitable d'arriver à des situations de famine (c'est-à-dire des circuits de i-transitions) ou à effectuer une action GET" :

$$
\begin{array}{ll}
1: & \nu X.\,(\\
2: & \quad [PUT.*] \\
3: & \quad \mu Y.\,(\\
4: & \quad\quad (\\
5: & \quad\quad \nu Z.(\langle i \rangle\, Z) \\
6: & \quad\quad \vee \\
7: & \quad\quad ([GET.*]\,\mathsf{T} \wedge [\neg GET.*]\,\mathsf{F} \wedge \langle \mathsf{T} \rangle\, \mathsf{T}) \\
8: & \quad\quad) \\
9: & \quad\quad \vee \\
10: & \quad\quad (\langle \mathsf{T} \rangle\, \mathsf{T} \wedge [\mathsf{T}]\, Y) \\
11: & \quad) \\
12: & \quad \wedge \\
13: & \quad [\mathsf{T}]\, X \\
14: &)
\end{array}
$$

Cette formule de μ-calcul d'alternance 1 contient trois points fixes (deux plus grands et un plus petit). La combinaison du plus petit point fixe μY avec celui du plus grand point fixe νZ permet de définir la condition d'inévitabilité (lignes 3–11). Le plus grand point fixe englobant (νX) vérifie que tout état atteint par une action PUT.* ne contredit pas la propriété logique.

La valeur T de vérité de l'évaluation de cette formule sur le protocole de Bit Alterné avec 256 messages différents, provoque l'exploration de tout le STE, à cause de la formule νX. Elle a été calculée en à peu près 90 secondes avec la version distribuée d'EVALUATOR sur 10 nœuds, alors qu'il a fallu au moins 15 minutes pour les versions DFS et BFS séquentielles pour terminer le calcul, soit 10 fois plus lentement que l'évaluation distribuée. La version séquentielle (BFS) d'EVALUATOR doit stocker beaucoup d'information, comme la version distribuée, pour pouvoir réaliser les parcours en avant et en arrière du graphe booléen sous-jacent au problème d'évaluation. Cette technique peut provoquer un manque d'espace mémoire (limitée à $1,5$ Go sur les machines séquentielles utilisées dans les expérimentations) alors que l'évaluation distribuée, grâce à une répartition à faible surcoût de ses données sur l'espace mémoire d'un ensemble de machines, peut résoudre des problèmes qui seraient trop larges pour la solution BFS. Finalement, deux aspects de notre approche ont été confirmés : la correction du comportement de l'algorithme distribué de résolution de SEBs monobloc et multibloc, et le passage à l'échelle des études de cas pouvant être traitées en pratique (dû aux limites en temps d'exécution et en espace mémoire).

- Seuls les dix plus larges STEs de la base de tests VLTS ont été utilisés pour réaliser les évaluations de performance qui suivent. Ces STEs sont décrits dans le tableau 6.3.

  Pour chaque exemple, nous donnons le nombre d'états du STE, le nombre de transitions, et la présence (X) ou non (-) d'*interblocage* et de situation de *famine*.

  La propriété la plus simple que nous avons considérée concerne l'absence de blocage (ou absence d'interblocage), c'est-à-dire d'états n'ayant aucun successeur (également appelés *états puits*). L'absence de ces états peut être caractérisée en μ-calcul modal par la formule suivante :

$$\nu X.\,(\langle \mathsf{T} \rangle\, \mathsf{T} \wedge [\mathsf{T}]\, X)$$

  La situation de famine reflète, quant à elle, la présence de cycles de séquences d'exécution, qui dénotent la monopolisation indéfinie (dans le cas présent, modélisée par un cycle de i-transitions) d'une ressource par un des processus au détriment des autres processus. Cette

EXEMPLE	états	transitions	interblocage	famine
vasy_2581_11442	2 581 374	11 442 382	X	-
vasy_4220_13944	4 220 790	13 944 372	X	-
vasy_4338_15666	4 338 672	15 666 588	X	-
vasy_6020_19353	6 020 550	19 353 474	-	X
vasy_6120_11031	6 120 718	11 031 292	X	-
cwi_7838_59101	7 838 608	59 101 007	-	X
vasy_8082_42933	8 082 905	42 933 110	X	-
vasy_11026_24660	11 026 932	24 660 513	X	-
vasy_12323_27667	12 323 703	27 667 803	X	-
vasy_33949_165318	33 949 609	165 318 222	-	X

Table 6.3: Description des dix plus larges STEs du VLTS avec la présence (X) ou non (-) d'interblocage ou de famine

situation signifie que, par exemple, le protocole correspondant n'est pas équitable envers l'accessibilité des processus à cette ressource partagée. La présence de famine peut être définie en μ-calcul modal par la formule suivante :

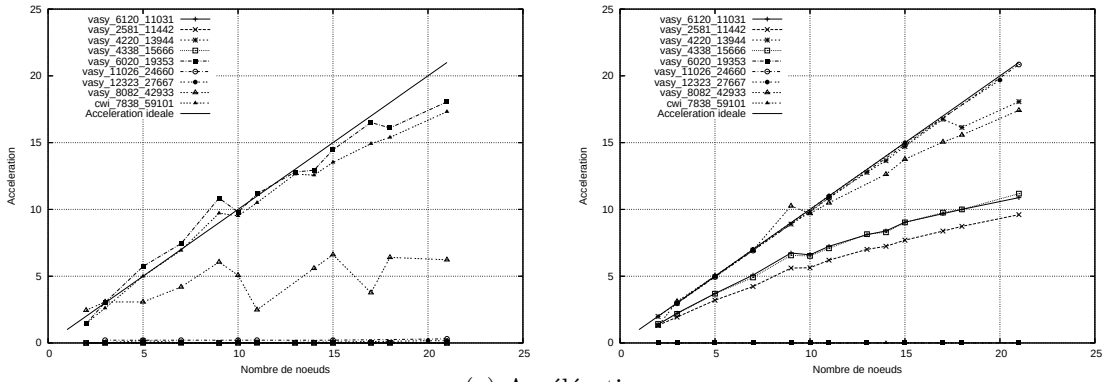
$$\mu X.\,(\nu Y.\,(\langle i \rangle\, Y) \vee \langle \mathsf{T} \rangle\, X)$$

Dans la suite de cette étude, nous comparons d'abord les performances en temps d'exécution, consommation mémoire et passage à l'échelle de la version distribuée avec la version séquentielle (basée sur un parcours en profondeur d'abord) de l'outil EVALUATOR. Dans un deuxième temps, nous reprendrons les résultats obtenus par le premier algorithme d'évaluation à la volée distribuée de formule de μ-calcul (UPPDMC) et présentés dans [HLL04]. Nous les comparerons aux mesures de performance effectuées sur notre méthode distribuée basée sur la résolution de SEBs, afin d'en extraire les avantages et inconvénients de chaque approche.

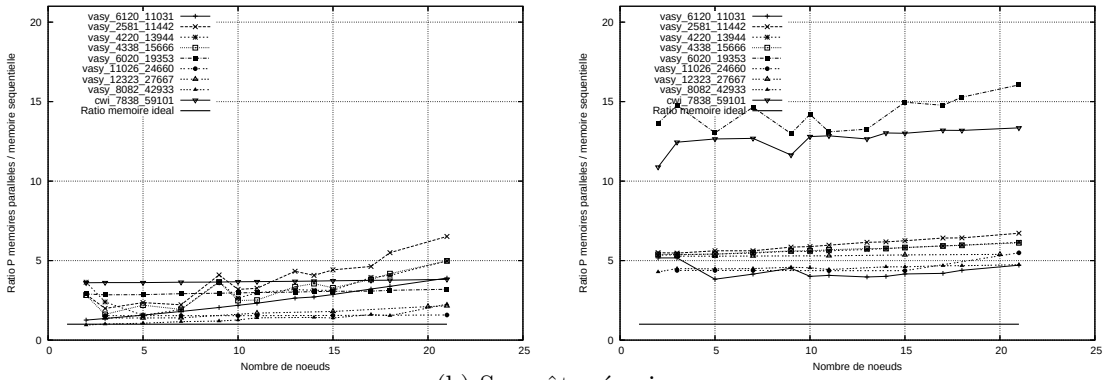
Accélération La version distribuée d'EVALUATOR utilisant l'algorithme MB-DSOLVE permet une très bonne accélération (voir la figure 6.6(a)) par rapport à la version séquentielle basée sur un parcours DFS. Le choix d'une solution séquentielle DFS a été motivé par plusieurs raisons : il s'agit généralement de la solution séquentielle qui réalise les temps d'exécution les plus bas (comparés à des solutions basées sur un parcours BFS), ce qui la rend plus intéressante pour évaluer l'accélération absolue d'une solution distribuée ; la version distribuée d'EVALUATOR a quand même été comparée (avec succès) avec la version BFS d'EVALUATOR et les résultats obtenus ont été des accélérations pour la plupart meilleures (devenant même superlinéaires avec un grand nombre de nœuds) et sur un plus grand ensemble d'expériences que pour la solution DFS. Afin de présenter les avantages réels d'une solution distribuée par rapport à la meilleure solution séquentielle existante, seuls les résultats de la comparaison avec la solution séquentielle DFS sont décrits dans cette étude.

Les STEs *vasy_6020_19353* et *cwi_7838_59101* sont intéressants à double titre puisqu'ils ne contiennent pas d'interblocage mais contiennent des situations de famine (voir le tableau 6.3) :

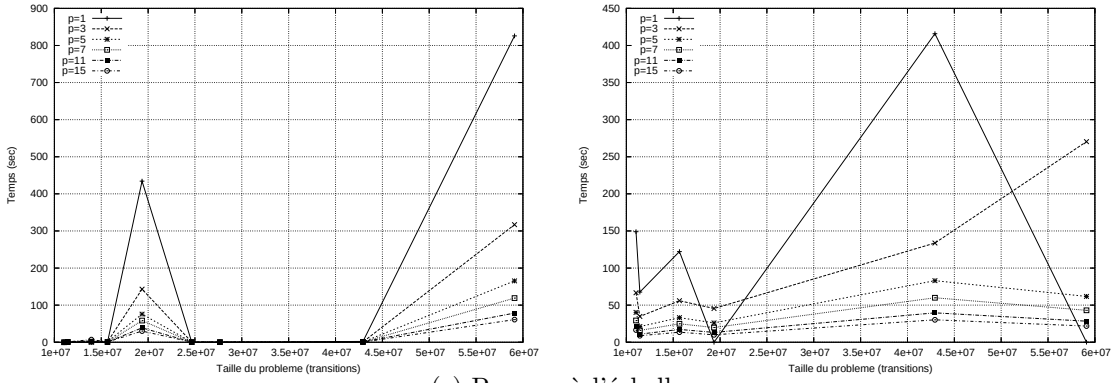
- L'évaluation de la propriété d'absence d'interblocage sur ces STEs va engendrer une exploration et stabilisation totale du SEB correspondant, puisqu'une telle propriété ne pourra jamais être évaluée à vraie, pour aucun des états parcourus. Le SEB résultant étant large (proportionnellement à la taille des STEs et de la formule logique à évaluer), la version distribuée d'EVALUATOR montre un réel intérêt en permettant un gain de temps quasi-linéaire au nombre de nœuds utilisés. Il faut plus de 13 minutes pour vérifier cette propriété d'absence d'interblocage sur le STE *cwi_7838_59101* avec la version séquentielle, alors que seulement 47 secondes



(a) Accélération



(b) Surcoût mémoire



(c) Passage à l'échelle

Figure 6.6: Accélération (a), surcoût mémoire (b) et passage à l'échelle (c) d'EVALUATOR distribué *vs.* EVALUATOR séquentiel (en DFS) pour les propriétés de détection d'interblocage (colonne de gauche) et de famine (colonne de droite)

sont nécessaires avec la version distribuée sur 21 nœuds, réalisant ainsi une évaluation 17 fois plus rapide (voir la colonne gauche de la figure 6.6(a)). On peut observer que l'évaluation de ces deux STEs se démarque bien des autres STEs puisqu'il s'agit des courbes présentant les meilleures accélérations (et donc plus proches de l'accélération idéale théorique).

Les autres exemples de STEs contiennent tous au moins un interblocage, qui est détecté comme étant un contre-exemple par un mécanisme généralement aussi rapide avec la version distribuée qu'avec les versions séquentielles. Les temps d'exécution sont très courts et l'accélération pratiquement nulle (voir les points proches de 0 du graphe de la colonne gauche de la figure 6.6(a)). Cette caractéristique confirme les résultats obtenus avec DSOLVE dans le cadre du problème de vérification basé sur la comparaison par équivalence, où nous décrivions la rapidité de l'approche à trouver et générer des contre-exemples dans le cas où l'équivalence n'était pas respectée entre les deux automates comparés. Cela justifie une nouvelle fois l'extension conservatrice de DSOLVE qui a été réalisée dans l'algorithme MB-DSOLVE, en permettant à ce dernier d'être aussi rapide qu'un algorithme spécialisé pour les propriétés monoblocs.

Remarque 6.2
L'évaluation de la propriété d'absence d'interblocage pour le STE *vasy_8082_42933* dure moins de 0.5 secondes, car un contre-exemple est trouvé immédiatement. Cela explique l'irrégularité de la courbe correspondante d'accélération de la solution distribuée sur la figure 6.6(a), qui pour ce type de mesures de performances ne présente de résultats très significatifs. ■

- L'évaluation de la propriété de présence de famine sur les STEs *vasy_6020_19353* et *cwi_7838_59101* va aboutir, quant à elle, très rapidement à la découverte d'un exemple confirmant la propriété logique.

  Contrairement aux propriétés monoblocs, lors de l'évaluation de propriétés multibloc, comme pour la détection de présence de famine (colonne droite de la figure 6.6(c)), l'asynchronisme du parcours BFS réalisé par l'ensemble des nœuds combiné avec la distribution par message de portions de bloc à explorer ou à stabiliser, font que la version distribuée a tendance à explorer plus de variables booléennes et plus de blocs que les versions séquentielles. Ce comportement résulte en des calculs d'exploration, de stabilisation, de détection de terminaison, de réception et d'émission de messages qui rendent la version distribuée plus lente à découvrir des contre-exemples (voir les courbes d'accélération proche de 0 dans la figure 6.6(a) colonne droite). En revanche, elle présente un avantage majeur sur les versions séquentielles dans le cas d'applications qui nécessiteraient plusieurs résolutions du même SEB multibloc, puisque de plus grandes portions de ce SEB sont explorées et stabilisées à chaque appel de résolution, et peuvent être utilisées directement dans les appels de résolution suivants.

  Les autres exemples de STEs ne contiennent pas de situation de famine, et nécessitent donc une exploration et stabilisation totale du SEB sous-jacent. Pour ce type d'évaluation, une méthode distribuée présente beaucoup d'avantages, et en particulier une accélération allant de faible (mais quand même supérieure au temps de résolution séquentielle) à quasi-linéaire (voir les courbes distinctes de l'axe des abscisses sur la colonne droite de la figure 6.6(a)). Un exemple de très bonne accélération est obtenu avec les STEs *vasy_11026_24660* et *vasy_12323_27667* pour lesquels le temps d'évaluation séquentiel est de 160 045 (*resp.* 195 767) secondes, soit presque 2 (*resp.* 2, 5) jours de calcul, alors qu'il ne faut que 7 641 (*resp.* 9 352) secondes, soit environ 2 (*resp.* 2, 5) heures, pour les évaluer en distribué sur 21 nœuds. Les accélérations ainsi réalisées approchent l'optimal théorique soit 21.

Une piste intéressante, que l'on laissera pour travail futur, serait d'étudier des améliorations possibles pour détecter plus efficacement la terminaison anticipée par stabilisation directe de la variable d'intérêt, ce qui correspond à la détection de contre-exemples. Ces améliorations peuvent porter à la fois sur la propagation arrière de variables stabilisées, sur les priorités entre les différentes opérations

distribuées pour l'ensemble des blocs en cours de traitement, ou encore sur le mécanisme de stabilisation passive pour chacun des blocs, qui représentent autant de filtres et de points de synchronisation dont on cherche à réduire les effets.

Nous avons également confronté notre version distribuée d'EVALUATOR avec la seule autre solution à la volée distribuée existante à notre connaissance, nommée UPPDMC [HLL04].

La figure 6.7 présente, d'une part sous forme de tableau (a) et d'autre part sous forme graphique (b), les temps d'exécution réalisés par UPPDMC avec 25 nœuds bi-processeurs sur les dix plus larges STEs du VLTS (présentés dans le tableau 6.3) et les propriétés logiques d'absence d'interblocage et de présence de famine. Les temps d'exécution de la version distribuée d'EVALUATOR avec 21 nœuds sur les mêmes exemples y sont également reportés.

Remarque 6.3
Seul le temps de la résolution du graphe de jeux ou du SEB correspondant au problème d'évaluation est pris en compte. Les temps relatifs à la copie et à la lecture de la formule logique et du STE sont exclus car considérés comme constants et dépendants de l'architecture parallèle et du réseau utilisés pour les expérimentations. ■

Nous pouvons observer que la détection de contre-exemples se fait beaucoup plus rapidement avec EVALUATOR qu'avec UPPDMC, puisque quelque soit la propriété vérifiée (absence d'interblocage ou présence de famine), les temps d'évaluation sont très inférieurs par rapport à ceux de UPPDMC. Il est à noter que les versions séquentielles donnent également des temps d'exécution bien inférieurs par rapport à ceux de UPPDMC pour la détection de contre-exemples. Une raison possible est le mécanisme efficace de propagation des valeurs stabilisées lors de la résolution du SEB par les algorithmes séquentiels $A1 - A4$ et distribué MB-DSOLVE, car cette opération de stabilisation arrière des dépendances a la plus haute priorité dans le calcul de résolution.

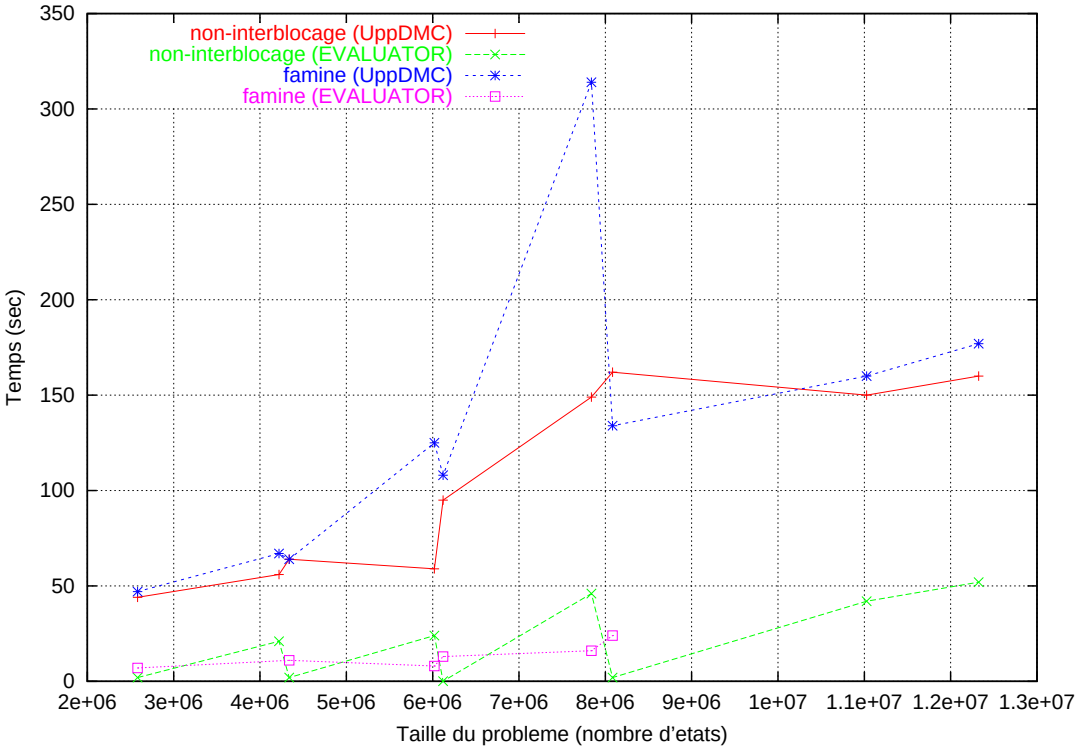
Ces chiffres confirment une nouvelle fois le bon comportement de l'utilisation de MB-DSOLVE pour la résolution de SEBs monoblocs, qui résultent dans le cas présent de l'évaluation de la propriété d'absence d'interblocage.

Remarque 6.4
Quelques singularités sont néanmoins observables dans le tableau (a) de la figure 6.7. Concernant les STEs *vasy_4220_13944*, *vasy_11026_24660* et *vasy_12323_27667* qui prennent beaucoup de temps pour l'évaluation de la propriété de famine, ce comportement étrange est également visible avec les versions séquentielles d'EVALUATOR et avec d'autres outils de CADP (notamment lors de la simple génération du STE explicite correspondant à ces STEs) et ne dépend donc pas de notre algorithme de résolution de SEBs mais plutôt de l'environnement d'expérimentation (en constante évolution, et pouvant donc être instable par période). La raison de la non-présence de résultats pour le STE *vasy_33949_165318* est similaire, puisque le mécanisme de représentation du STE (basé sur le format BCG) consomme plus de mémoire que celle disponible sur les PCs XEON utilisés dans le cadre de cette expérience. Nous supposons que les travaux de [HLL04] ne reposent pas sur la même représentation des STEs et qu'ils transforment à un moment donné le STE du format BCG en un autre format, à notre connaissance, non documenté. Nous avons ainsi préféré ne pas mettre ces résultats non représentatifs dans l'illustration graphique pour une meilleure lisibilité des autres résultats non affectés par ce problème d'environnement de prototypage. Il est à noter que ces singularités sont susceptibles de disparaître dans une prochaine version de CADP. ■

Les courbes présentées dans la figure 6.7(b) illustrent la différence de temps d'exécution entre UPPDMC et EVALUATOR distribué sur l'évaluation des propriétés d'absence d'interblocage et de présence de famine. Alors que les courbes relatives à EVALUATOR augmentent à peine avec la taille

Exemple	interblocage (sec)		famine (sec)	
	(UppDMC)	(Evaluator)	(UppDMC)	(Evaluator)
vasy_2581_11442	44	2	47	7
vasy_4220_13944	56	21	67	622
vasy_4338_15666	64	2	64	11
vasy_6020_19353	59	24	125	8
vasy_6120_11031	95	0.05	108	13
cwi_7838_59101	149	46	314	16
vasy_8082_42933	162	2	134	24
vasy_11026_24660	150	42	160	7 641
vasy_12323_27667	160	52	177	9 352
vasy_33949_165318	560	n.c.	8 715	n.c.

(a) Comparaison des temps d'exécution



(b) Représentation graphique des temps d'exécution

Figure 6.7: Temps d'exécution des évaluateurs distribués UppDMC (25 nœuds) et Evaluator (21 nœuds) pour les propriétés d'absence d'interblocage et de présence de famine sur les dix plus larges STEs du Vlts

du problème, celles de UppDMC augmentent de manière plus marquée lors du traitement de STEs de plus en plus larges, quelques soient les propriétés vérifiées. Toutefois, il ne faut pas perdre de vue la différence de machines utilisées pour les expérimentations (25 pour UppDMC, 21 pour Evaluator).

Coût en mémoire L'utilisation de la version distribuée d'Evaluator n'engendre qu'un faible surcoût en mémoire (par rapport à la version séquentielle) indépendamment de la taille du STE à évaluer et du nombre de nœuds utilisés pour le calcul (voir la figure 6.6(b)).

Les courbes, représentant le rapport entre mémoire distribuée totale et mémoire séquentielle utilisée pour l'évaluation de chaque propriété logique, sont planes et proches de la courbe optimale théorique ($y = 1$), qui représente le cas où la version distribuée consommerait autant de mémoire que la version séquentielle Dfs.

La seule exception est lors de la détection de contre-exemples d'une propriété multibloc où la version distribuée à tendance à explorer une plus grande partie du Seb multibloc sous-jacent par rapport à la version séquentielle, et donc nécessite plus de mémoire pour stocker les informations correspondantes (notamment les transitions du Seb parcourus qui doivent être sauvegardées pour la propagation arrière future des valeurs stables). Ce mécanisme explique la présence des deux courbes plus hautes que les autres dans la colonne droite de la figure 6.6(b).

La figure 6.8 synthétise les consommations mémoires totales relevées sur chacune des évaluations de propriétés et de STEs pour les outils UppDMC (sur 25 nœuds) et Evaluator distribué (sur 21 nœuds).

La rapidité de détection de contre-exemples lors de l'évaluation de formules logiques monoblocs avec Evaluator se répercute sur une consommation mémoire relativement basse par rapport à celle de UppDMC. Par contre, lorsqu'il est nécessaire de parcourir tout le STE (et par suite tout le Seb) pour évaluer la valeur de vérité d'une formule logique, la nécessité de sauvegarder toutes les transitions du Seb, pénalise la version distribuée d'Evaluator car la représentation en mémoire des dépendances sauvegardées utilise des listes, qui ne sont pas des structures de données optimales pour ce genre d'information.

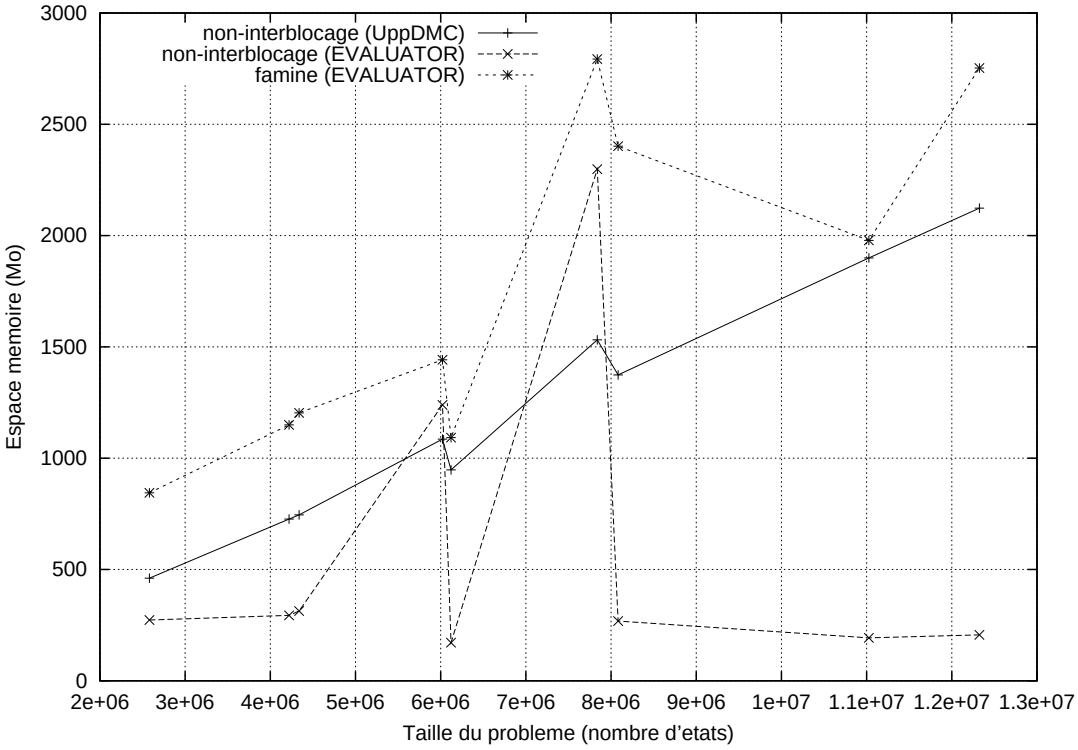
De plus, lorsque la résolution du Seb ne nécessite pas de réaliser des propagations arrières de variables stables, l'information correspondant aux dépendances entre variables est inutile pour la valeur de vérité finale de la variable d'intérêt, et ne devrait donc pas être stockée. La version séquentielle d'Evaluator ne souffre pas de ce surcoût mémoire car elle dispose d'options de parcours optimaux pour ce type d'évaluation (où elle ne sauvegardera pas les dépendances entre variables booléennes du Seb sous-jacent). C'est pourquoi, deux pics sont observables sur la courbe "non-interblocage Evaluator".

Remarque 6.5
Seule la consommation mémoire relative à l'évaluation des formules logiques décrivant l'absence d'interblocage, est présente dans les résultats publiés concernant l'outil UppDMC. En conséquence, nous ne présentons que la courbe "famine Evaluator" dans la figure 6.8, et ne pouvons pas comparer la consommation mémoire de notre approche avec celle de UppDMC dans le cadre de propriétés multiblocs. ■

Passage à l'échelle L'écart grandissant entre le temps de calcul séquentiel ($p = 1$) et distribué ($p = 15$) proportionnellement à la taille des problèmes à évaluer (voir la figure 6.6(c)) justifie l'intérêt d'utiliser une solution distribuée par rapport à une solution séquentielle. Pratiquement, la version distribuée d'Evaluator permet non seulement d'évaluer des STEs et propriétés que les versions séquentielles n'arrivent pas à calculer par manque d'espace mémoire, mais aussi de gagner

Exemple	interblocage (Ko)		famine (Ko)	
	(UppDMC)	(Evaluator)	(UppDMC)	(Evaluator)
vasy_2581_11442	461 082	272 800	n.c.	844 476
vasy_4220_13944	726 659	294 068	n.c.	1 149 432
vasy_4338_15666	745 048	313 252	n.c.	1 203 684
vasy_6020_19353	1 085 694	1 239 396	n.c.	1 442 008
vasy_6120_11031	947 283	170 892	n.c.	1 092 652
cwi_7838_59101	1 531 674	2 298 504	n.c.	2 793 420
vasy_8082_42933	1 374 050	268 292	n.c.	2 401 760
vasy_11026_24660	1 899 274	192 756	n.c.	1 978 668
vasy_12323_27667	2 123 571	206 456	n.c.	2 752 868
vasy_33949_165318	6 562 918	n.c.	n.c.	n.c.

(a) Comparaison de la consommation mémoire



(b) Représentation graphique de la consommation mémoire

Figure 6.8: Consommation mémoire totale des évaluateurs distribués UppDMC (25 nœuds) et Evaluator (21 nœuds) pour les propriétés d'absence d'interblocage et de présence de famine sur les dix plus larges STEs du Vlts

un temps considérable (allant jusqu'à plusieurs jours de calcul dans le cas des STEs *vasy_11026_24660* et *vasy_12323_27667* sur lesquels la présence de famine est évaluée).

Remarque 6.6
Afin de rendre plus lisible le graphe de passage à l'échelle dans le cas de l'évaluation de propriétés de famine (colonne droite de la figure 6.6(c)), nous ne faisons pas figurer les temps d'exécution des expériences *vasy_4220_13944*, *vasy_11026_24660* et *vasy_12323_27667*, qui sont respectivement de 12 705, 160 045 et 195 767 secondes avec la solution séquentielle, car cela empêcherait d'observer les exemples de STEs pour lesquels un contre-exemple est trouvé en moins d'une minute, et pour lesquels des observations importantes peuvent être réalisées. ■

Les courbes donnent les résultats attendus, à savoir un gain de temps linéaire en nombre de nœuds de calcul utilisés. De nombreux points se situent proche de l'axe des abscisses parce qu'ils traduisent des expériences où un contre-exemple a été calculé très rapidement (de l'ordre de la seconde). Les pics (colonne gauche de la figure 6.6(c)), au contraire, décrivent des exemples pour lesquels tout le STE doit être exploré, comme pour les STEs *vasy_6020_19353* et *cwi_7838_59101* avec la propriété d'absence d'interblocage. La rapidité de détection de contre-exemples est bien illustrée par l'exemple *cwi_7838_59101* sur la propriété de famine, où la version séquentielle trouve immédiatement un sous-graphe booléen menant à la situation de contre-exemple, contrairement à la version distribuée qui doit parcourir une plus grande portion du SEB à cause de la distribution de l'information de stabilité des variables du graphe booléen en cours de résolution.

——— ⋆ ——— ⋆ ——— ⋆ ———

Le problème d'évaluation locale de formules logiques temporelles, exprimées en μ-calcul modal d'alternance 1, se prête bien à la distribution sur un ensemble de machines interconnectées au vu des résultats très positifs accomplis par la version distribuée d'EVALUATOR.

Les accélérations proches du linéaire par rapport à des versions séquentielles optimisées, basées sur un parcours en profondeur d'abord, sont complétées par des surcoûts faibles en consommation mémoire, et un passage à l'échelle en taille du problème à évaluer.

Ces bonnes caractéristiques combinées à la généricité de l'approche, basée sur la résolution locale distribuée de SEBs multiblocs, nous permettent d'envisager son application directe et efficace à d'autres problèmes de vérification, comme la génération de cas de tests, qui peuvent également être traduits en terme de SEBs multiblocs.

Chapitre 7

Application 4 : Génération de cas de tests de conformité

Le *test*, dans toutes ses variantes, est la technique de validation de systèmes la plus ancienne et par suite la plus répandue. Nous nous intéressons dans ce chapitre à la génération de cas de test explicite pour tester la conformité d'une implémentation boîte-noire vis-à-vis d'une spécification, dans le contexte des systèmes réactifs. Le problème que nous considérons est la génération séquentielle et distribuée de cas de tests à partir d'un objectif de test, qui est ici un ensemble de scénarios intéressants que l'on veut observer au cours de l'exécution du test. Des solutions efficaces à ce problème ont été proposées dans le cadre de modèles à états finis, tels que les STEs, fondé sur l'utilisation de calculs de point fixe. Nous les étendons par l'utilisation des SEBs pour l'encodage de la relation de conformité et par la possibilité de résoudre le problème de manière distribuée sur un ensemble de machines.

——— ⋆ ——— ⋆ ——— ⋆ ———

Ce chapitre apporte deux contributions majeures au problème de la génération de cas de test de conformité : la traduction du problème sous forme de SEBs, et la génération du graphe complet de test à partir du diagnostic de la résolution du SEB, tous deux pouvant être exécutés de manière distribuée en utilisant l'algorithme MB-DSOLVE [JM06].

La section 7.1 introduit la génération automatique de tests et plus particulièrement celle des cas de test de conformité. La section 7.2 décrit le mécanisme d'extraction à la volée de cas de test. Une traduction de ce mécanisme en termes de SEBs est donnée dans la section 7.3, ainsi qu'une implémentation et diverses expérimentations à la fois séquentielles (utilisant CÆSAR_SOLVE et TGV) et distribuées (utilisant MB-DSOLVE) dans la section 7.4.

7.1 Expression des cas de tests de conformité

Le *test de conformité* vise à établir qu'une implémentation sous test (IUT) d'un système est correcte par rapport à une spécification. Ce test est de type fonctionnel, car seul le comportement décrit par la spécification est vérifié vis-à-vis de l'implémentation. Etant donné que l'implémentation est inconnue lors de la génération de test, ce type de test est appelé test *boîte noire*. Une fois le cas de test obtenu, celui-ci est simulé sur l'IUT à l'aide d'un *testeur* qui déroule l'opération de test selon les sorties émises

par l'implémentation.

Il existe historiquement deux familles de génération formelle de test qui diffèrent selon le modèle utilisé pour représenter le graphe d'états décrivant le comportement de la spécification formelle du logiciel à tester :

- la première, plus ancienne, est fondée sur les *automates de Mealy* [LY94] et présente des algorithmes très coûteux, ne permettant leur application qu'à des graphes d'états de taille très modeste (de l'ordre de la centaine d'états et de transitions),
- la seconde est fondée sur les STEs et est essentiellement issue des algèbres de processus et des problèmes d'équivalences de test, pour lesquels il existe des algorithmes performants.

Nous considérons cette dernière méthode dans cette étude, et plus particulièrement l'approche de test de conformité mise en avant par le travail fondateur sous-jacent à l'outil TGV [JJ05]. Nous donnons seulement un bref résumé de la théorie utilisée par TGV et nous nous concentrons sur les aspects algorithmiques de la sélection de tests, avec l'objectif de les reformuler en termes de résolution de SEB et de génération de diagnostic.

7.1.1 TGV : un générateur automatique de cas de test de conformité

TGV (*Test Generation with Verification technology*) est un outil pour la génération de suite de tests de conformité depuis des spécifications formelles de systèmes réactifs. Les éléments principaux de la théorie de test sous-jacente à TGV sont :

- le modèle de STE distinguant entrées, sorties et actions internes (STEES),
- le concept de relation de conformité, et
- les notions d'exécution et de verdict.

Les algorithmes de génération de test, fondée sur la sélection à la volée de cas de test en fonction d'objectifs de tests, viennent pour la plupart de la technologie de vérification. A l'origine, TGV permet la génération de test focalisée sur des comportements spécifiés au moyen d'objectifs de tests décrits par des STEES explicites.

Les principales opérations (voir la figure 7.1) de TGV sont :

1. un produit synchrone qui identifie les séquences de la spécification acceptées par l'objectif de test,
2. l'abstraction et la déterminisation du calcul des actions tirables visibles, et
3. la sélection de cas de tests par accessibilité et coaccessibilité.

TGV a été développé en collaboration avec VERIMAG à Grenoble et utilise les bibliothèques de la boîte à outils CADP [Gar98]. TGV peut être vu comme une bibliothèque qui peut être liée à différents simulateurs à travers des APIS bien définies. Une version académique de TGV est distribuée dans la boîte à outils CADP et permet la génération de tests depuis des spécifications LOTOS par une connexion à l'interface OPEN/CÆSAR. La même API est utilisée pour une connexion avec la plateforme UMLAUT de validation de modèles UML. Une nouvelle version de TGV étend la précédente avec de nouvelles fonctionnalités pour la génération de tests basée sur un critère de couverture, combiné avec une génération de tests basée sur des objectifs de tests.

7.1.2 Système de transitions étiquetées à entrées/sorties

L'IUT et la spécification que nous considérons dans cette étude sont modélisés par des STE*s à entrées/sorties* (STEES).

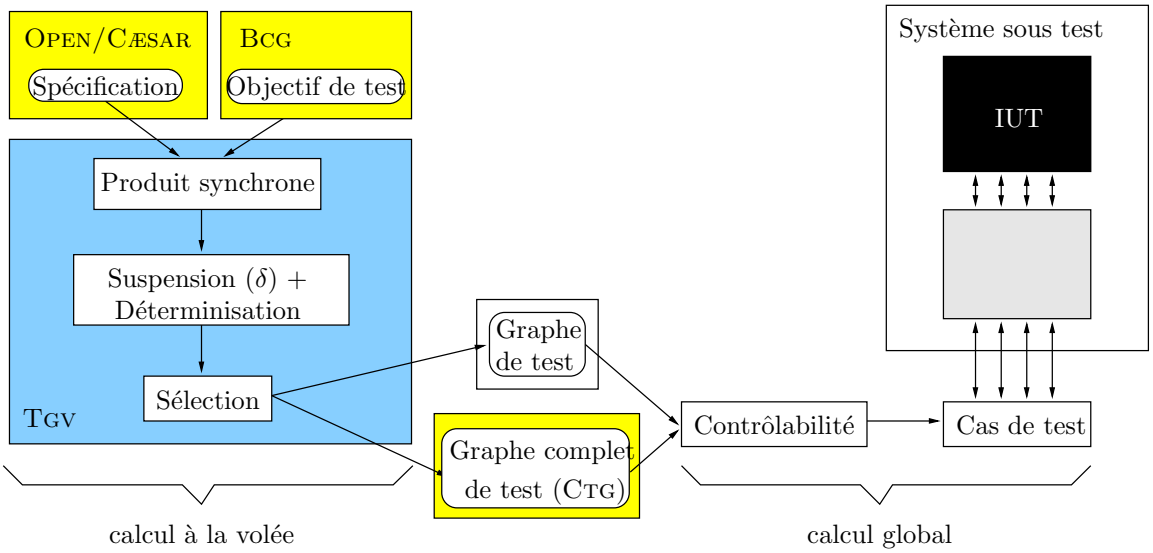


Figure 7.1: Architecture de l'outil Tgv

Définition 7.1 (Système de transitions étiquetées à entrées/sorties)
Un système de transitions étiquetées à entrées/sorties (SteEs) M est un quadruplet (Q^M, A^M, T^M, q_0^M), où :

- Q^M est un ensemble dénombrable non vide d'états,
- A^M est un alphabet dénombrable d'actions partitionnées en trois ensembles disjoints $A^M = A_I^M \cup A_O^M \cup I^M$, avec A_I^M l'ensemble des actions d'entrées, A_O^M l'ensemble des actions de sorties et $I^M = \{\tau\}$ l'ensemble des actions internes (inobservables),
- $T^M \subseteq Q^M \times A^M \times Q^M$ est un ensemble de transitions, et
- $q_0 \in Q^M$ est l'état initial.

■

Remarque 7.1
Par la suite, le SteEs de la spécification formelle du logiciel à tester sera noté $M = (S^M, A^M, T^M, s_0^M)$. ■

Intuitivement, les actions d'entrée de l'Iut sont contrôlées par l'environnement, alors que les actions de sortie sont seulement observables. En pratique, les tests observent les traces d'exécution composées d'actions observables de l'Iut, mais peuvent aussi détecter la *quiescence*, laquelle peut être de trois sortes :

- *interblocage* (états sans successeur),
- *famine* (circuit d'actions internes, qui fait partie d'une composante fortement connexe (Cfc) non triviale d'actions internes) et
- *blocage en sortie* (états sans transition successeur étiquetée par une action de sortie).

Pour un SteEs M, la quiescence est modélisée par un *automate de suspension* $\Delta(M)$.

Définition 7.2 (STEES de suspension)
Soit un STEES $M = (Q^M, A^M, T^M, q_0^M)$. L'automate de suspension de M est le STEES $\Delta(M) = (Q^M, A^{\Delta(M)}, T^{\Delta(M)}, q_0^{\Delta(M)})$, où :

- $A^{\Delta(M)} = A^M \cup \{\delta\}$, avec $\delta \in A_O^{\Delta(M)}$ (c'est-à-dire δ est une sortie) et
- $T^{\Delta(M)} = T^M \cup \{q \xrightarrow{\delta} q \mid q \in quiescent(M)\}$ (c'est-à-dire que $T^{\Delta(M)}$ est obtenue à partir de T^M par ajout de boucles $q \xrightarrow{\delta} q$ pour tous les états q de quiescence).

■

Remarque 7.2
Les traces de $\Delta(M)$ sont appelées traces de suspension de M. ■

Définition 7.3 (Relation de conformité)
Une *relation de conformité* est une relation entre implémentations et spécifications qui définit exactement l'ensemble des implémentations conformes à une spécification donnée. ■

Remarque 7.3
Pour les STEEs, plusieurs relations sont envisageables suivant les observations permises par le test (par exemple, des traces, blocages, transitions tirables, refus, ...) et la notion de correction voulue. ■

Parmi les relations de conformité existantes, nous avons choisi de retenir la relation de conformité **ioco** [Tre96] entre l'IUT et la spécification M, qui semble appropriée à bon nombre de cas.

Définition 7.4 (Relation de conformité ioco)
Soit M un STEES et IUT un STEES complet en entrée et compatible avec M. Dire que IUT est conforme modulo **ioco** à M signifie : IUT **ioco** $M \triangleq \forall \sigma \in Traces(\Delta(M)) \mid Out(\Delta(\text{IUT})\ after\ \sigma) \subseteq Out(\Delta(M)\ after\ \sigma)$ ■

Remarque 7.4
Intuitivement, la relation de conformité **ioco** spécifie qu'après avoir exécuté chaque trace de suspension (incluant les quiescences) de M, l' (automate de suspension de l') IUT renvoie seulement les sorties et les quiescences qui sont permises par M. ■

La génération de tests requiert une déterminisation de M, puisque deux séquences avec les mêmes traces d'actions observables ne peuvent être distinguées. Puisque la quiescence doit être préservée, la déterminisation doit avoir lieu après la construction de l'automate de suspension $\Delta(M)$.

Définition 7.5 (Cas de test)
Un *cas de test* est un STEES $TC = (S^{TC}, A^{TC}, T^{TC}, s_0^{TC})$ déterministe équipé de trois ensembles distincts d'états puits **Pass** $\cup$ **Fail** $\cup$ **Inconc** $\subseteq S^{TC}$ dénotant des verdicts.

Les actions de TC sont partitionnées en $A^{TC} = A_I^{TC} \cup A_O^{TC}$, où $A_O^{TC} \subseteq A_I^M$ (TC n'émet que des entrées de M) et $A_I^{TC} \subseteq A_O^{\text{IUT}} \cup \{\delta\}$ (TC capture les sorties et les quiescences de l'IUT). ■

De plus, un cas de test TC doit satisfaire plusieurs propriétés structurelles, détaillées en [JJ05], qui sont :

- les états **Fail** and **Inconc** ne sont accessibles en un pas que par des actions d'entrée,
- un verdict est accessible à partir de chaque état,

- dans tous les états, aucun choix n'est permis entre deux sorties, ou entre une sortie et une entrée (principe de *contrôlabilité*), et
- chaque état où une entrée est possible doit être complet en entrées (c'est-à-dire toute action d'entrée de l'alphabet A^{TC} est au moins présente une fois sur l'ensemble des transitions successeurs de l'état).

L'exécution d'un cas de test TC sur une IUT est modélisée par une composition parallèle $TC||\Delta(IUT)$ avec une synchronisation sur les actions communes observables, les verdicts étant déterminés par les états puits accessibles par les traces maximales de $TC||\Delta(IUT)$.

La technique de génération de tests de TGV est basée sur les *objectifs de test*, lesquels permettent de guider la sélection des cas de test.

Définition 7.6 (Objectif de test)
Un *objectif de test* est un STEES déterministe et complet $TP = (S^{TP}, A^{TP}, T^{TP}, s_0^{TP})$, avec les mêmes actions que la spécification $A^{TP} = A^M$, et équipé de deux ensembles d'états puits $Accept^{TP} \subseteq S^{TP}$ et $Reject^{TP} \subseteq S^{TP}$, qui sont utilisés pour sélectionner les comportements cibles et pour couper les explorations de M, respectivement. ■

TGV décompose le processus de génération de test en plusieurs étapes, dont la plupart sont effectuées à la volée pendant le parcours en avant du STEES M modélisant la spécification.

Ici, nous nous concentrons sur le calcul du *graphe complet de test* (CTG), qui contient tous les cas de tests correspondant à une spécification et un objectif de test, et peut ainsi servir de critère de comparaison et de mesures de performance. Les cas de tests contrôlables peuvent être obtenus à partir du CTG en appliquant des algorithmes spécifiques, comme ceux décrits dans [JJ05].

7.2 Génération locale des cas de test de conformité

Le CTG est produit par TGV au moyen de trois opérations, toutes effectuées à la volée :

1. le calcul du produit synchrone $SP = M \times TP$ entre le STEES de la spécification M et l'objectif de test TP, dans le but de marquer les états d'acceptation et de rejet,
2. la suspension et la déterminisation de SP, menant à $SP^{vis} = det(\Delta(SP))$, lequel ne garde que les comportements visibles et la quiescence, et
3. la sélection des cas de test dénotant les comportements de SP^{vis} acceptés par TP, laquelle forme le CTG.

Cette dernière est l'opération principale et consiste à calculer $L2A$, le sous-ensemble d'états de SP^{vis} à partir duquel un état accepteur est atteignable (*lead to accept*), vérifiant si l'état initial de SP^{vis} appartient à $L2A$ (ce qui indique l'existence d'un cas de test) et définissant grâce à $L2A$ les ensembles **Pass**, **Fail** et **Inconc** correspondant aux verdicts. Il s'agit de l'opération que nous avons décidé de traduire en résolution de SEB et génération de diagnostic.

En prenant pour hypothèse que les états accepteurs de SP^{vis} sont marqués par des boucles de transitions étiquetées par une action acc (comme cela est réalisé en pratique), l'accessibilité d'un état accepteur est dénoté par la formule de μ-calcul suivante :

$$\phi_{acc} = \mu Y.(\langle acc \rangle \mathsf{T} \vee \langle \mathsf{T} \rangle Y)$$

L'ensemble $L2A$ contient tous les états satisfaisant ϕ_{acc}. Il peut être calculé en arrière en utilisant une itération de point-fixe évaluant ϕ_{acc} sur SP^{vis}. Puisque cela requiert le calcul *a priori* de SP^{vis},

nous avons recherché une autre solution adaptée à l'exploration à la volée, en considérant la formule suivante :

$$\phi_{l2a} = \nu X.(\phi_{acc} \wedge [\mathsf{T}]\,(\phi_{acc} \Rightarrow X))$$

La formule ϕ_{l2a} a la même interprétation que ϕ_{acc}, ce qui signifie que sa satisfaction par l'état initial de SP^{vis} dénote l'existence d'un cas de test. De plus, l'évaluation à la volée de ϕ_{l2a} sur un état s satisfaisant ϕ_{acc} requiert l'inspection de chaque successeur s' de s et, s'il satisfait ϕ_{acc}, l'évaluation récursive de ϕ_{l2a} sur s', etc., jusqu'à ce que tous les états dans $L2A$ ont été explorés.

Le CTG peut alors être obtenu sous forme du diagnostic produit par un évaluateur à la volée (comme EVALUATOR) pour la formule ϕ_{l2a}.

7.3 Traduction de la génération des cas de tests de conformité en termes de SEBs

Cependant, pour annoter le CTG avec des informations de verdict et éviter les redondances causées par les deux occurrences de ϕ_{acc} présentes dans ϕ_{l2a} (ce qu'un évaluateur de logique temporelle n'est pas prévu d'effectuer), une traduction à un grain plus fin du problème en termes de résolution de SEB avec diagnostic est préférée.

Le SEB correspondant dénote le problème d'évaluation de ϕ_{l2a} sur SP^{vis}, en appliquant la traduction donnée dans la section 6.3 (s et s' sont des états de SP^{vis}) :

$$\phi_{l2a} = \nu X.(\mu Y.(\langle acc\rangle\,\mathsf{T} \vee \langle\mathsf{T}\rangle\,Y) \vee$$
$$[\mathsf{T}]\,(\nu Z.([acc]\,\mathsf{F} \wedge [\mathsf{T}]\,Z) \vee X))$$

Ce qui donne sous forme de SEB :

$$\{X_s =_\nu Y_s \wedge \textstyle\bigwedge_{s\to s'}(Z_{s'} \vee X_{s'})\}, \{Y_s =_\mu \textstyle\bigvee_{s \xrightarrow{acc} s'} \mathsf{T} \vee \bigvee_{s\to s'} Y_{s'}\},$$
$$\{Z_s =_\nu \textstyle\bigwedge_{s \xrightarrow{acc} s'} \mathsf{F} \wedge \bigwedge_{s\to s'} Z_{s'}\}$$

Si $X_{s_0^{vis}}$ est vraie, alors un diagnostic positif (c'est-à-dire un exemple) peut être extrait sous forme d'un sous-graphe booléen [Mat00] contenant, pour chaque variable conjonctive (telle que X_s et Z_s), toutes ses variables successeurs, et pour chaque variable disjonctive (telle que Y_s) seulement un successeur qui est évalué à vrai. Ce diagnostic peut être obtenu par un autre parcours en avant de la portion de graphe booléen déjà explorée pour évaluer $X_{s_0^{vis}}$. Nous le transformons en un CTG de la manière suivante :

- un état du CTG est associé à chaque variable X_s,
- une transition sortante d'acceptation est produite pour la variable X_s seulement si la première sous-formule dans la partie droite de l'équation définissant Y_s est vraie (c'est-à-dire, s a un successeur *acc*),
- une transition $X_s \xrightarrow{a} X_{s'}$ est produite pour chaque état s' tel que $Z_{s'}$ est faux.

Remarque 7.5
Il est à noter que le diagnostic pour les variables Z_s ne nécessite pas d'être exploré. ■

Des informations supplémentaires de verdict sous forme de transitions de rejet (ou *refuse*) et non concluantes (ou *inconclusive*) sont produites de façon similaire durant la génération de diagnostic, puisque l'information nécessaire pour ces verdicts dans le CTG est locale par rapport aux états de $L2A$ [JJ05].

7.4 Réalisation et applications

Dans la discussion ci-dessus, la formule ϕ_{l2a} a été évaluée sur le STEEs SP^{vis} obtenu après suspension et déterminisation de SP. Cependant, ces deux opérations peuvent aussi être réalisées *après* la sélection de cas de test. En d'autres mots, la procédure de génération basée sur les SEBs et décrite dans ses grandes lignes précédemment, peut être appliquée directement au produit synchrone SP entre la spécification et l'objectif de test, produisant un CTG *brut*, qui est ultérieurement suspendu et déterminisé pour obtenir le CTG final.

7.4.1 Développement de l'outil EXTRACTOR

Cette procédure est à la base de l'outil EXTRACTOR (voir la figure 7.2) que nous avons développé dans CADP [GLM02] en utilisant l'environnement générique OPEN/CÆSAR [Gar98] pour l'exploration à la volée de STEs. EXTRACTOR produit des CTGs bruts, qui sont ensuite suspendus et déterminisés au moyen de l'outil DETERMINATOR [HJ03], conduisant à des CTGs fortement bisimilaires à ceux produits par TGV.

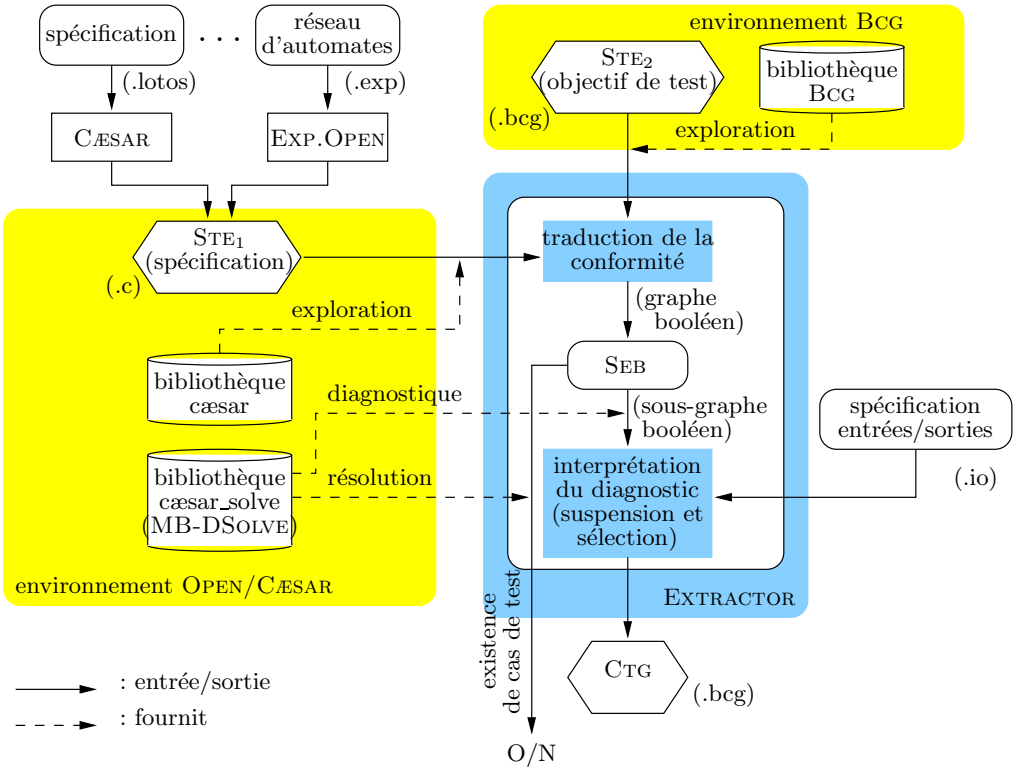


Figure 7.2: Architecture de l'outil EXTRACTOR

Remarque 7.6
Bien que cet ordre des opérations ne soit pas le plus efficace pour la génération séquentielle à la volée de cas de test (puisque le STEEs de la spécification peut contenir une grande proportion de τ-transitions), il semble être adapté à un environnement distribué, puisqu'il entraîne la manipulation

sous-jacente de larges SEBs, qui sont résolus efficacement par notre algorithme distribué MB-DSOLVE. ■

EXTRACTOR (1 300 lignes de code C) consiste en deux parties : une partie avant, chargée de traduire la sélection de tests guidée par l'objectif de test STE_2 sur le STE_1 vers une résolution de SEB et de produire un CTG correspondant en interprétant le diagnostic fourni par la résolution de SEB ; et une partie arrière, responsable de la résolution du SEB jouant le rôle d'un moteur de vérification.

Globalement, notre approche pour la génération distribuée à la volée de cas de test est de construire à la volée le STE et le SEB correspondant, ainsi que de déterminer de manière distribuée la valeur finale de la variable principale du SEB.

Les versions séquentielles et distribuées de l'outil EXTRACTOR sont obtenues en utilisant comme partie arrière soit les algorithmes séquentiels de la bibliothèque CÆSAR_SOLVE [Mat03a, Mat06], soit l'algorithme MB-DSOLVE, respectivement.

DETERMINATOR (voir la figure 7.3) a été étendu pour permettre la déterminisation des CTGs (notamment la fermeture transitive et la compression des τ-transitions en utilisant les algorithmes proposés en [Mat05]) et la sélection des transitions (comme la détection de famine, ou encore l'existence et insertion d'un état q_{AT} d'acceptation pour déplier les états accepteurs étant aussi des états de quiescence, c'est-à-dire ayant des transitions sortantes δ).

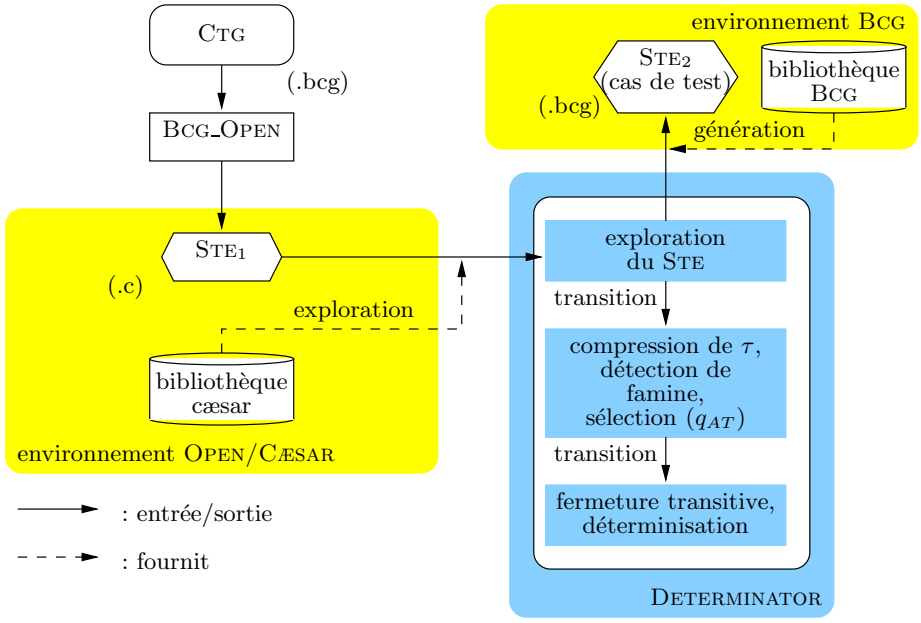


Figure 7.3: Architecture de l'outil DETERMINATOR

Ainsi, une solution alternative àTGV consiste Ã utiliser conjointement EXTRACTOR et DETERMINATOR tel que le montre la figure 7.4.

7.4.2 Analyse de performances

Nous présentons dans cette section des mesures expérimentales comparant la version distribuée d'EXTRACTOR avec son homologue séquentiel utilisant les algorithmes optimisés de CÆSAR_SOLVE

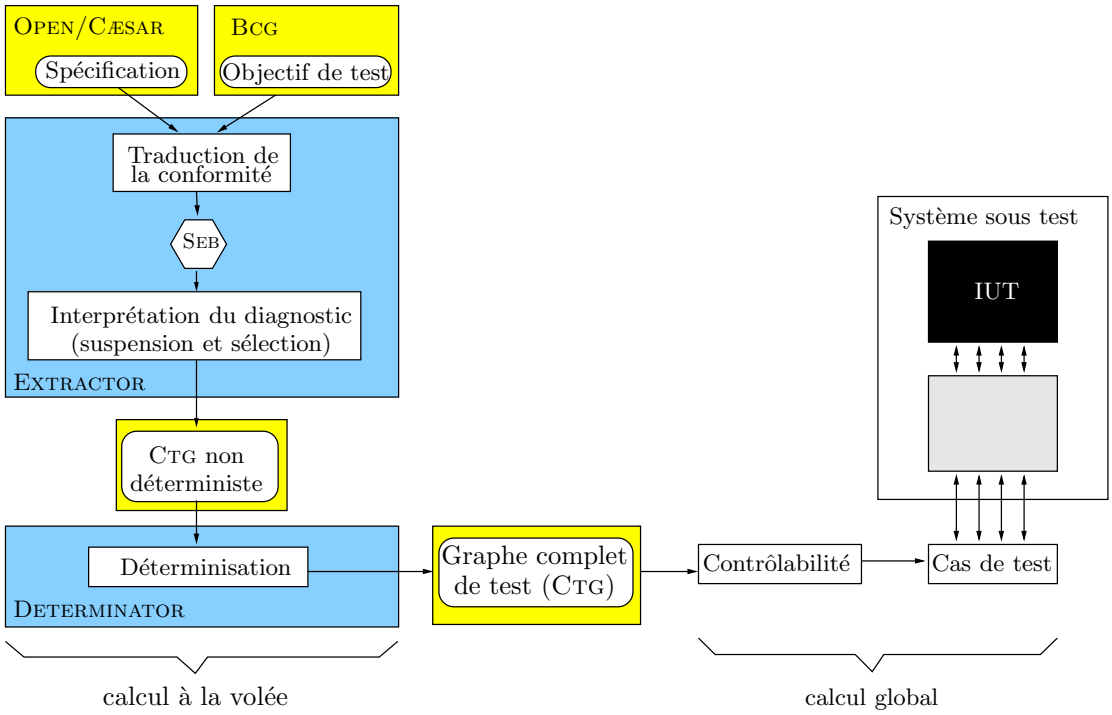


Figure 7.4: Génération de cas de tests avec Extractor et Determinator

pour la résolution de Sebs disjonctifs ou conjonctifs, mais aussi avec le générateur séquentiel de cas de test Tgv.

Objectif de test générique et études de cas Symétriquement aux expériences d'évaluation de formules de logique temporelle présentées dans la section 6.4.2, nous avons adopté une approche générique pour l'expérimentation de la génération de cas de test de conformité.

Nous avons utilisé un objectif de test générique, qui est le suivant (donné en format *.AUT*) :

```
==============
tp_generic.aut
==============
des (0, 11, 11)
(0, "[^i].*", 1)
(1, "[^i].*", 2)
(2, "[^i].*", 3)
(3, "[^i].*", 4)
(4, "[^i].*", 5)
(5, "[^i].*", 6)
(6, "[^i].*", 7)
(7, "[^i].*", 8)
(8, "[^i].*", 9)
(9, "[^i].*", 10)
(10, ACCEPT, 10)
```

Cet objectif de test indique que "après 10 actions visibles, un état accepteur est atteignable". L'état accepteur est ici représenté par une boucle sur lui-même portant l'action $ACCEPT$.

Plusieurs STEs extraits du VLTS ont été de nouveau utilisés comme entrées pour notre outil EXTRACTOR de génération distribuée à la volée de cas de test.

Le tableau 7.1 décrit chacun des STEs utilisés pour l'expérimentation de la génération à la volée de cas de test de conformité grâce à l'outil EXTRACTOR connecté à CÆSAR_SOLVE et MB-DSOLVE (et utilisation de la grappe de PCs XEON mentionnée au chapitre 2) et grâce à l'outil TGV. Pour chacun d'entre eux, nous donnons le nombre d'états, de transitions totales ainsi que le nombre de transitions invisibles.

EXEMPLE	états	transitions	τ-transitions
vasy_164_1619	164 865	1 619 204	109 910
vasy_166_651	166 464	651 168	91 392
cwi_214_684	214 202	684 419	550 611
cwi_371_641	371 804	641 565	445 600
vasy_386_1171	386 496	1 171 872	122 976
cwi_566_3984	566 640	3 984 157	3 666 614
vasy_1112_5290	1 112 490	5 290 860	0
b256	6 067 712	19 505 146	17 663 482

Table 7.1: STEs pris comme exemples de spécification formelle

Les exemples ont des tailles variant de 6 067 712 états et 19 505 146 transitions ($b256$) à 164 865 états et 1 619 204 transitions ($vasy\_164\_1619$), et sont tous (exceptés l'exemple $b256$) extraits de la base de tests VLTS.

$b256$ est une variante du protocole du *bit alterné* écrit en *full* LOTOS par [Gar89]. La taille de l'espace d'état correspondant dépend du nombre de messages différents possibles échangés. Dans l'exemple $b256$, le nombre de messages différents est de 256.

Remarque 7.7
Nous ne montrons dans cette étude que des résultats obtenus à partir d'espace d'états pré-construit, car la base de test VLTS ne propose que ce format explicite. Nous avons néanmoins testé les différentes versions de notre outil EXTRACTOR sur des spécifications formelles implicites de haut niveau, par exemple décrite en LOTOS pour l'exemple $b256$, à partir desquelles l'espace d'état correspondant est parcouru et construit dynamiquement à l'aide d'une fonction successeur. ■

Nous avons étudié le comportement d'EXTRACTOR sur la génération des CTGs bruts correspondants, ultérieurement suspendus et déterminisés en utilisant DETERMINATOR.

Le tableau 7.2 décrit les résultats obtenus en temps d'exécution (minutes), en espace mémoire (MO) et en taille (nombre d'états et de transitions) pour les CTG obtenus avec chacun des deux outils TGV et EXTRACTOR (version séquentielle) sur chacun des exemples du tableau 7.1. Nous indiquons également le pourcentage d'amélioration (en temps et espace mémoire) obtenu grâce à l'outil EXTRACTOR par rapport à l'outil TGV.

Remarque 7.8
Etant donné que EXTRACTOR ne produit que le CTG brut (donné par la taille en nombre d'états et de transitions dans le tableau 7.2), une comparaison plus fine avec TGV devrait en outre ajouter les temps et espaces mémoires utilisés par l'outil DETERMINATOR, qui, dû à des contraintes architecturales logicielles, ne peut être assemblé à EXTRACTOR en un seul outil à l'instant présent. L'addition des performances donne néanmoins une idée de l'applicabilité de notre méthode ainsi que de son efficacité

Exemple	Tgv				Extractor (séquentiel)					
	min.	Mo	états	trans.	min.	%	Mo	%	états	trans.
vasy_164_1619	15'8s	242	100 319	231 266	3'47s	75	210	13	438 861	2 982 696
vasy_166_651	20'23s	242	170 657	586 602	1'41s	92	113	53	444 542	1 504 985
cwi_214_684	39'	$\geq$ 3000	échec	échec	8s	∞	19	∞	aucun cas de test	
cwi_371_641	6'5s	1600	125 894	597 445	5'20s	12	310	81	1 912 260	3 163 177
vasy_386_1171	9s	11	3 319	3 892	7s	22	10	9	5 561	6 324
cwi_566_3984	2'36s	$\geq$ 3000	échec	échec	1'11s	∞	82	∞	240 668	524 298
vasy_1112_5290	23s	33	10 827	20 888	13s	44	28	15	15 008	41 225
b256	597'4s	2322	264 194	854 786	139'22s	77	2772	-2	12 139 232	39 020 231

Table 7.2: Comparaison des performances de Tgv avec Extractor (séquentiel) sur quelques exemples du Vlts avec l'objectif de test générique

en observant la taille du Ctg final (donné dans le tableau 7.3) et le temps d'exécution et espace mémoire global nécessaire à son calcul. Il est à noter que les Ctgs finaux peuvent être minimisés par l'équivalence forte afin d'obtenir des Ctgs minimaux pour l'étape de contrôlabilité. ■

Le tableau 7.3 montre les performances obtenues avec la version distribuée de Extractor sur 7 nœuds de la grappe de Pcs Xeon, et le Ctg final (nombre d'états et de transitions) produit par Determinator, avec les temps d'exécution et espaces mémoires utilisés pour chacun des outils.

Exemple	Extractor (distribué)		Determinator			
	min.	Mo	min.	Mo	états (final)	transitions (final)
vasy_164_1619	4'39s	470	4'40s	55	103 658	975 594
vasy_166_651	2'59s	335	2'27s	50	173 259	801 675
cwi_214_684	19s	144	inexistence de cas de test de conformité			
cwi_371_641	12'4s	880	25'8s	185	127 218	777 278
vasy_386_1171	16s	104	15s	6	2 452	3 894
cwi_566_3984	3'40s	926	18'44s	63	32	49
vasy_1112_5290	27s	228	17s	7	8 369	41 225
b256	180'	6127	19'	459	527 875	1 709 058

Table 7.3: Performances réalisées avec Extractor (distribué sur 7 nœuds), et Ctg final obtenu par Determinator

Accélération Concernant les résultats obtenus par Extractor séquentiel, la première constatation est que l'utilisation d'un mécanisme de résolution de Seb correspondant au problème de génération de graphes complets de test permet toujours un gain en temps d'exécution par rapport à celui de Tgv qui varie entre 12% et ∞ (pour les exemples *cwi_566_3984* et *cwi_214_684* qui ne terminent pas avec Tgv).

Quelque soit le type de Ste à traiter, Extractor montre un comportement homogène et les différences de temps d'exécution ne sont pas aussi démesurées que celles observées avec Tgv. Les performances en temps d'exécution sont très liées à la structure du Ste pris comme spécification formelle de l'implémentation à tester. En effet, la présence (ou absence) de τ-transitions va induire (ou éviter) des calculs de fermeture transitive de Cfc plus ou moins complexes, et accélérer la vérification de l'existence de cas de test. Par exemple, le Ste *vasy_1112_5290* ne contient aucune τ-transition, et les outils Tgv et Extractor trouvent immédiatement les différentes traces de 10 actions visibles menant à un état accepteur indiqué par l'objectif de test générique.

Un autre aspect intéressant qui confirme les bonnes performances de nos algorithmes de résolution de Seb, est la rapidité avec laquelle un contre-exemple est trouvé par Extractor. En effet, pour

l'exemple *cwi_214_684* aucun cas de test, conforme à l'objectif de test générique et à la spécification donnée, n'existe. Cette non-existence est détectée en 9 secondes avec EXTRACTOR alors que TGV, après 39 minutes de calcul, a consommé toute la mémoire dynamique (et secondaire sur disque) disponible et ne peut pas rendre un résultat.

D'autres tests ont aussi été effectués pour la génération de CTG à partir d'exemples de STE et d'objectifs de test présent dans la boîte à outils CADP. Les CTG ont été générés avec succès à la fois par TGV et par EXTRACTOR, mais sont trop rapides pour une comparaison fine des performances des deux approches.

Pour ce qui est de l'outil EXTRACTOR distribué, le tableau 7.3 montre également un bon comportement de la distribution (avec des accélérations significatives) de la résolution du SEB sous-jacent, et des temps d'exécution comparables à ceux obtenus par EXTRACTOR séquentiel, ainsi que les mêmes CTG finaux fortement équivalents à ceux produits par TGV.

Remarque 7.9
Sur les exemples présentés dans cette section, il est difficile de montrer l'intérêt de la version distribuée d'EXTRACTOR vis-à-vis de ses versions séquentielles utilisant des algorithmes optimisés. En effet, le mécanisme de génération distribuée de diagnostic utilisé par EXTRACTOR distribué, implique une centralisation de l'information du diagnostic sur le nœud superviseur (c'est-à-dire utilisateur) afin de construire explicitement le diagnostic sur disque dans le format BCG. Ce protocole induit un important transfert de données à travers le réseau en plus d'un goulot d'étranglement au niveau du nœud superviseur, ce qui pénalise globalement la génération du diagnostic. Finalement, une grande proportion du temps de calcul est passée sur la synchronisation entre les nœuds travailleurs et superviseur lors de cette génération non-optimisée de diagnostic.

Il existe plusieurs possibilités de continuation de nos travaux au niveau de la génération distribuée de diagnostic. L'une d'elle consisterait à générer de manière distribuée des diagnostics eux-mêmes distribués. Des travaux similaires ont été réalisés dans ce sens par [GMS01, Jou02, GMB+06, GMB+05] dans le cadre de la génération distribuée d'espace d'états au cours de laquelle des espaces d'états partitionnés sont stockés localement sur les disques de chaque nœud dans le format PBG (c'est-à-dire BCG partitionné). En utilisant de nouveau ce format, nous pensons pouvoir accélérer significativement le processus de génération distribuée à la volée de graphe complet brut de test. Les diagnostics distribués résultants en format PBG pourront alors, si nécessaire, être fusionnés en seul CTG global (format BCG) grâce à l'outil BCG_MERGE développé à ce propos et déjà présent dans CADP. ■

La taille des CTGs finaux est également un critère de comparaison entre les différentes approches. En observant les résultats du tableau 7.3, nous constatons que dans la moitié des cas EXTRACTOR combiné avec DETERMINATOR génèrent un CTG final plus petit que celui généré par TGV. Les tailles de CTG finaux obtenus par DETERMINATOR varient de 32 états et 49 transitions pour l'exemple *cwi_566_3984* à 527 875 états et 1 709 058 transitions pour l'exemple *b256*, alors que pour TGV elles varient de 3 319 états et 3 892 transitions pour l'exemple *vasy_386_1171* à 264 194 états et 854 786 transitions pour l'exemple *b256*.

Nous avons aussi comparé par équivalence forte, grâce à l'outil BISIMULATOR, les différents CTG obtenus pour EXTRACTOR et TGV, qui sont effectivement équivalents deux à deux. La différence de taille de CTG entre les deux méthodes n'étant pas importante, l'alternative de générer des CTGs en passant par une résolution de SEBs est donc pleinement justifiée.

Remarque 7.10
Une possibilité d'extension de ce travail de génération de tests basée sur les SEBs est d'introduire la fermeture transitive de transitions τ gardant les informations de famine (réalisée par DETERMINATOR) plus tôt dans la génération de diagnostic afin de réduire la taille du CTG brut, et par suite du

temps d'exécution total, et espace mémoire utilisé. Des premiers expériences ont montré des résultats encourageants, avec une taille du CTG brut en nombre d'états et de transitions divisée par deux. ■

Coût en mémoire Un autre point intéressant est la consommation mémoire réalisée par l'outil EXTRACTOR qui présente un bon comportement à la fois en séquentiel et en distribué. Pour la plupart des exemples, EXTRACTOR séquentiel consomme moins de mémoire que TGV. Le gain en mémoire varie de -2% (sur l'exemple *b*256 où tous les outils ont du utilisé de la mémoire secondaire sur disque qui fausse les résultats de performances) à ∞ (pour les exemples *cwi*_566_3984 et *cwi*_214_684 qui ne terminent pas avec TGV).

En effet, sur les deux exemples que sont les tests *cwi*_566_3984 et *cwi*_214_684 pris dans la suite de tests VLTS, TGV est très rapidement (en moins de deux minutes pour le premier exemple) à court de mémoire, et dépasse même l'espace mémoire secondaire (sur disque) alloué par le système d'exploitation, soit plus de 3 GO de mémoire au total.

Par contre, pour les mêmes exemples, EXTRACTOR les résout sans problème entre 1 et 3 minutes pour le premier exemple, et entre 8 et 19 secondes pour le second, avec la version séquentielle et distribuée, respectivement.

La consommation mémoire exhibée par EXTRACTOR permet ainsi de résoudre des graphes complets de test non réalisables avec TGV.

Passage à l'échelle Le passage à l'échelle se fait à la fois en taille du problème à résoudre, mais également par l'ajout de nœuds supplémentaires au calcul. Ce deuxième point nécessiterait des expérimentations complémentaires du type de ceux présentés dans les chapitres précédents.

Par contre, pour ce qui est du premier point, nous avons déjà vu que certains exemples (*cwi*_566_3984 et *cwi*_214_684) n'étaient pas traitables par TGV alors qu'ils le sont très rapidement avec EXTRACTOR.

En ce qui concerne le passage à l'échelle d'EXTRACTOR distribué, nous avons pu résoudre sur 16 nœuds le SEB correspondant à la génération d'un CTG pour l'objectif de test générique et le STE *b*256, avec 6 067 712 états et 19 505 146 transitions, en moins de 372 secondes, alors que la version séquentielle d'EXTRACTOR a pris plus de 30 minutes pour réaliser cette même résolution. Le même exemple exécuté avec TGV prend plus de 597 minutes en tout soit 4 fois plus lentement que EXTRACTOR séquentiel combiné avec DETERMINATOR.

——— ⋆ ——— ⋆ ——— ⋆ ———

En appliquant MB-DSOLVE à la résolution du SEB traduisant le problème, nous avons obtenu un générateur distribué à la volée de cas de test dont les capacités passent bien à l'échelle par rapport à des outils séquentiels existants bien établis.

L'architecture modulaire de notre outil EXTRACTOR permet de bénéficier à la fois des algorithmes séquentiels optimisés de la bibliothèque CÆSAR_SOLVE mais également de constituer le premier outil à notre connaissance capable de distribuer la génération automatique à la volée de cas de test de conformité sur un ensemble de machines interconnectées, de type grappe de PCS.

Par cette application à un autre problème de vérification formelle, nous avons montré l'applicabilité de notre approche distribuée ainsi que les bonnes performances réalisées par notre algorithme de résolution de SEBS d'alternance 1 à blocs multiples MB-DSOLVE, à des cas concrets de taille réaliste.

Conclusion

Bilan

La vérification des propriétés des programmes parallèles constitue une étape indispensable pour en garantir le bon fonctionnement. Depuis plus de dix ans, la distribution de la vérification a été proposée et reconnue comme une technique adaptée au problème d'explosion d'états de la vérification basée sur les modèles.

Toutefois, les travaux consacrés à la vérification distribuée se sont orientés dans de multiples directions. Un grand nombre d'algorithmes distribués ont été définis, la plupart d'entre eux se focalisant sur un aspect précis du problème de vérification, par exemple l'équilibrage de charge ou, au contraire, l'efficacité des algorithmes de vérification pour un système décrit dans un langage spécifique. De plus, des approches antagonistes (par exemple, l'opposition des algorithmes pour architectures à mémoire distribuée et partagée) ont encore accentué l'aspect fragmentaire des travaux dans ce domaine.

Dans cette étude, nous avons cherché, au contraire, une démarche unificatrice basée sur une approche pragmatique adaptée aux besoins de la vérification :

- Nous avons étudié attentivement la littérature scientifique consacrée à la vérification distribuée et parallèle, ce qui nous a permis une classification des différents problèmes de vérification qui ont été distribués durant ces dix dernières années, ainsi que les modèles d'architecture, de programmation et d'algorithmes distribués proposés (voir le chapitre 1).
- En confrontant l'état de l'art avec notre propre expérience acquise lors de l'étude de cas réalistes, nous avons constaté que les techniques de vérification distribuée "classiques" ont des algorithmes dépendants à la fois du formalisme utilisé pour décrire le système à vérifier, et du problème de vérification à distribuer, ce qui les rend ni utilisables directement pour de nouveaux problèmes de vérification, ni adaptables à d'autres techniques de vérification classiques. Ceci nous a incité à définir des algorithmes distribués pour une représentation théorique unifiante des méthodes de vérification, basée sur les SEBs (*systèmes d'équations booléennes*). Cette approche modulaire basée sur la résolution de SEB simplifie le développement des outils de vérification en séparant clairement la traduction en SEB d'un problème spécifique (partie avant de l'outil), du moteur de résolution de SEB (partie arrière), qui peuvent être développés et optimisés indépendamment.
- Ce travail a débouché sur la définition et la conception de deux composants génériques et efficaces de vérification facilitant le développement d'outils robustes d'analyse d'espace d'états explicites. Il s'agit des algorithmes distribués, DSOLVE et MB-DSOLVE, qui permettent respectivement la résolution distribuée à la volée de SEBs monoblocs et multiblocs en utilisant plusieurs machines connectées par un réseau. Ces algorithmes présentent un certain nombre de caractéristiques qui les rendent très utiles en pratique :

  - Un mécanisme de passage par messages pour la distribution des données reposant sur une interface de communication non-bloquante asynchrone, utilisée efficacement par le biais de protocoles standards bas niveau (tels que les sockets UNIX avec tampons de communication bornés et protocoles TCP/IP) ;
  - Une complexité de résolution en temps, en mémoire et en nombre de messages échangés linéaire en taille du système et de la propriété à vérifier, qui rend l'approche faisable pour des problèmes réels de vérification de larges systèmes ;
  - Une génération distribuée de diagnostic pour illustrer par un exemple ou un contre-exemple la valeur de vérité de la propriété évaluée ;
  - Un haut degré de parallélisme du calcul permis par une détection distribuée de terminaison à plusieurs niveaux de la résolution du SEB et des blocs parcourus.

- Nous avons entièrement implémenté ces idées, ce qui a donné naissance à une bibliothèque de résolution de SEB, appelée CÆSAR_SOLVE_2. Cette mise en œuvre a nécessité un important travail d'implémentation (plus de 10 000 lignes de code C pour DSOLVE, et plus de 15 000 lignes de code C pour son extension MB-DSOLVE), puisqu'il s'agit d'algorithmes distribués complexes, incluant : le comportement du nœud superviseur, le comportement des nœuds travailleurs, le protocole de détection distribuée de terminaison, la génération distribuée de diagnostic, la prise en compte du grain fin des communications (comme l'échec d'émissions ou de réceptions) et la distribution des données.

  Les algorithmes de résolution DSOLVE et MB-DSOLVE ont été définis de manière complètement formelle, notamment à l'aide de spécifications attribuées. A partir de ces spécifications, ils ont été implémentés de manière systématique et rigoureuse. Ces algorithmes devraient, en toute rigueur, être accompagnés d'une preuve formelle de leur correction. Mais la plupart du temps, ils nous ont semblés suffisamment sûrs et confrontés systématiquement à des problèmes d'implémentation et d'expérimentation réels pour qu'il soit permis de s'en dispenser. Lorsque cela s'est révélé nécessaire, les points délicats ont été soigneusement détaillés et étayés par des arguments de preuve.

- Dans un souci d'indépendance vis-à-vis des langages sources définissant les applications à vérifier, nos implémentations ont utilisé l'environnement d'exploration générique de STE OPEN/CÆSAR [Gar98] de la boîte à outils CADP [GLM02], et ont suivi l'interface générique pour la manipulation de SEB définie par la bibliothèque CÆSAR_SOLVE [Mat03a] de CADP afin de pouvoir être appliquées directement comme moteur de vérification pour les outils d'analyse ayant déjà une connexion avec la bibliothèque CÆSAR_SOLVE.

- Nous avons utilisé nos algorithmes distribués pour obtenir des versions distribuées de trois outils de vérification à la volée existants et développés dans CADP : le comparateur par équivalences BISIMULATOR [BDJM05], le réducteur par τ-confluence TAU_CONFLUENCE [PLM03] et l'évaluateur de formules logiques EVALUATOR [Mat03a]. Les gains en performance obtenus en utilisant DSOLVE et MB-DSOLVE permettent de passer à l'échelle les capacités de ces outils pour vérifier des systèmes plus larges et plus rapidement (sans pour autant consommer linéairement plus de mémoire) que les versions séquentielles correspondantes. Nous avons validé les extensions distribuées proposées sur des études de cas complexes et réelles, souvent industrielles, présentes dans la base de démonstrations de CADP, et dans celle de tests de performance VLTS [VLT03].

- Enfin, nous avons également proposé une nouvelle application de vérification : la génération distribuée à la volée de cas de tests et l'outil associé EXTRACTOR (1 300 lignes de code C). Nous avons défini la traduction de ce problème en termes de SEBs multiblocs et de génération de diagnostic spécifique, et comparé favorablement sa résolution basée sur des algorithmes de CÆSAR_SOLVE ou sur MB-DSOLVE, avec des outils séquentiels existants, tels que TGV. Les

tailles des IUT (*implémentations à tester*) pouvant être traités par notre approche passent bien à l'échelle et dépassent celles analysables par des solutions séquentielles existantes.

Notre travail de thèse visait principalement à concevoir et construire une infrastructure générique pour la vérification distribuée à la volée basée sur les modèles, et son application à plusieurs problèmes de vérification complexes. Les résultats de ce travail ont confirmé l'intérêt de notre approche : les algorithmes distribués DSOLVE et MB-DSOLVE permettent une distribution efficace du calcul de résolution de SEBs sur un ensemble de machines interconnectées, de type grappe de PCs, tout en bénéficiant d'une facilité et d'une efficacité d'utilisation dans les outils de vérification existants et nouveaux.

Perspectives

Le travail présenté dans ce document peut être continué dans plusieurs directions. D'un côté, nous pouvons ajouter des fonctionnalités à nos algorithmes distribués pour qu'ils puissent traiter une plus grande classe de problèmes. De l'autre, il est possible d'améliorer et d'étendre les outils de vérification distribuée à la volée proposés, et d'en construire de nouveaux selon le même schéma.

Dans le but d'augmenter sa flexibilité, la bibliothèque CÆSAR_SOLVE_2 peut être enrichie avec d'autres algorithmes distribués de résolution de SEB en développant des versions distribuées d'algorithmes comme LMC [DSC99] ou l'algorithme basé sur élimination de Gauss proposé en [Mad97] en incluant des optimisations séquentielles comme celles décrites dans [MS03, GK04]. Grâce à l'interface (API) bien définie de la bibliothèque et la disponibilité des primitives de OPEN/CÆSAR, le prototypage de nouveaux algorithmes est assez direct ; de ce point de vue, CÆSAR_SOLVE_2 peut être vue comme une plate-forme ouverte pour le développement et l'expérimentation d'algorithmes distribués de résolution de SEB. Nous pouvons également étudier d'autres stratégies de résolution distribuée visant à réduire la consommation mémoire pour les SEBs conjonctifs et disjonctifs qui surviennent fréquemment en pratique [Mat03a] : une telle stratégie pourrait combiner une exploration distribuée du graphe booléen basée sur un parcours en largeur d'abord et une exploration de ce même graphe locale à chaque nœud mais basée sur un parcours en profondeur d'abord.

Des travaux concurrents au notre, en cours au CWI (Amsterdam), visent à distribuer également la résolution de SEB et de SEBP (SEB *paramétré*) à partir de la méthode de Gauss. Il serait intéressant de comparer MB-DSOLVE avec les algorithmes qui résulteront de ces travaux. Une autre amélioration serait d'étendre notre algorithme à des spécifications comportant des données avec valeurs, menant à des modèles infinis et paramétrés, en résolvant les SEBPs correspondants, par réduction à un SEB et utilisation de raisonnement symbolique.

Afin de pouvoir utiliser pleinement les architectures parallèles de demain, comme les grilles de calcul, des éléments de calcul distribué doivent également être ajoutés à nos algorithmes et infrastructure générique de résolution de SEBs. De telles améliorations peuvent être une version multithreadée de nos algorithmes dissociant le flot d'exécution portant sur les calculs et celui portant sur la communication (par exemple, afin d'utiliser pleinement les grappe de SMPs composés de deux processeurs), la prise en compte de l'hétérogénéité des machines de calcul (en particulier, les problèmes d'équilibrage de charge dynamique, d'ordonnancement des tâches et de communication par passage de messages entre architectures hétérogènes) et plus généralement la robustesse des solutions distribuées proposées (comme la tolérance aux pannes).

Une autre direction de recherche est le développement d'outils de vérification. L'outil de comparaison par équivalence, BISIMULATOR, peut être étendu avec d'autres relations d'équivalences (par exemple, la bisimulation Markovienne [HS99], etc.), traduites sous forme de SEBs en suivant le schéma donné

dans la section 3.3. L'outil de réduction par τ-confluence, TAU_CONFLUENCE, peut être étendu avec d'autres formes de réduction d'ordre partiel (τ-confluence faible, τ-inertie [GP00], etc.) et peut être amélioré avec de nouvelles stratégies d'exploration dans le but d'améliorer l'accélération, en suivant les idées avancées dans la section 4.4.2. De nouvelles applications, comme l'analyse de flot de données [FS96] ou la satisfaction de clauses de Horn, pour laquelle il existe des traductions succinctes en termes de SEBs, de taille linéaire par rapport au nombre de littéraux et opérateurs dans la clause de Horn [LS98], peuvent être également développées et expérimentées avec DSOLVE et MB-DSOLVE. La traduction de la synthèse de contrôleurs discrets en termes de résolution de SEB avec diagnostic serait également très intéressante, en suivant l'approche proposée en [ZS05].

Enfin, une manière plus élaborée (que d'utiliser une fonction de hachage statique) de distribuer l'espace d'états, éventuellement dynamiquement, reste à mettre en place. Une idée est d'exploiter l'information sur la structure de l'espace d'états qui peut être obtenue à travers l'interprétation abstraite [OvdPE04].

Construire un réducteur distribué par équivalence dans le style de [Orz04] avec des extensions de parcours à la volée, représente également une continuation attractive du travail présenté dans cette thèse. Ce serait la dernière pièce manquante pour obtenir une solution complète de vérification distribuée à la volée pour la boîte à outils CADP.

Bibliographie

[AC88] A. Arnold and P. Crubillé. A linear algorithm to solve fixed-point equations on transition systems. *Information Processing Letters*, 29 :57–66, 1988.

[ADK97] S. Allmaier, S. Dalibor, and D. Kreische. Parallel Graph Generation Algorithms for Shared and Distributed Memory Machines. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and U. Trottenberg, editors, *Parallel Computing : Fundamentals, Applications and New Directions, Proceedings of the Conference ParCo'97 (Bonn, Germany)*, volume 12 of *Advances in Parallel Computing*, pages 581–588. Elsevier, North-Holland, 1997.

[AFK87] K. R. Apt, N. Francez, and S. Katz. Appraising fairness in languages for distributed programming. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages POPL '87 (Münich, West Germany)*, pages 189–198. ACM Press, January 1987.

[AH97] S. C. Allmaier and G. Horton. Parallel Shared-Memory State-Space Exploration in Stochastic Modeling. In G. Bilardi, A. Ferreira, R. Lüling, and J. Rolim, editors, *Proceedings of the International Conference on Solving Irregularly Structured Problems in Parallel*, volume 1253 of *Lecture Notes in Computer Science*, pages 207–218. Springer Verlag, 1997.

[AKH97] S. Allmaier, M. Kowarschik, and G. Horton. State Space Construction and Steady-State Solution of GSPNs on a Shared-Memory Multiprocessor. In *Proceedings of the 7th IEEE International Workshop on Petri Nets and Performance Models PNPM'97 (Saint Malo, France)*, pages 112–121. IEEE Computer Society Press, 1997.

[And94] H. R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1) :3–30, April 1994.

[AS85] Bowen Alpern and Fred B. Schneider. Defining Liveness. *Information Processing Letters*, 21(4) :181–185, 1985.

[AV95] H. R. Andersen and B. Vergauwen. Efficient Checking of Behavioural Relations and Modal Assertion s using Fixed-Point Inversion. In P. Wolper, editor, *Proceedings of the 7th International Conference on Computer Aid ed Verification CAV '95 (Liege, Belgium)*, volume 939 of *Lecture Notes in Computer Science*, pages 142–154. Springer Verlag, July 1995.

[BBM+03] Dominique Borrione, Menouer Boubekeur, Laurent Mounier, Marc Renaudin, and Antoine Sirianni. Validation of Asynchronous Circuit Specifications using IF/CADP. In Manfred Glesner, Ricardo Augusto da Luz Reis, Hans Eveking, Vincent John Mooney, Leandro Soares Indrusiak, and Peter Zipf, editors, *Proceedings of the International Conference on Very Large Scale Integration of System-on-Chip VLSI-SoC 2003 (Darmstadt, Germany)*, pages 86–91, Darmstadt, December 2003.

[BBS01] J. Barnat, L. Brim, and J. Stribrna. Distributed LTL Model-Checking in SPIN. In *Proc SPIN'01 (Toronto, Canada)*, volume 2057 of *Lecture Notes in Computer Science*, pages 200–216, May 2001.

[BCMS04] Lubos Brim, Ivana Cerná, Pavel Moravec, and Jirí Simsa. Accepting Predecessors Are Better than Back Edges in Distributed LTL Model-Checking. In Alan J. Hu and Andrew K. Martin, editors, *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design FMCAD'04 (Austin, Texas, USA)*, volume 3312 of *Lecture Notes in Computer Science*, pages 352–366. Springer-Verlag, November 2004.

[BCY02] Lubos Brim, Jitka Crhova, and Karen Yorav. Using Assumptions to Distribute CTL Model Checking. In Lubos Brim and Orna Grumberg, editors, *Proceedings of the 1st Workshop on Parallel and Distributed Model Checking PDMC'2002 (Brno, Czech Republic)*, volume 68 of *Electronic Notes in Theoretical Computer Science*, pages 559–574. Elsevier Science Publishers, August 2002.

[BDHGS00] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable Distributed On-the-Fly Symbolic Model Checking. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, (FMCAD'00)*, volume 1954 of *Lecture Notes in Computer Science*, pages 390–404. Springer Verlag, November 2000.

[BDJM05] Damien Bergamini, Nicolas Descoubes, Christophe Joubert, and Radu Mateescu. BISIMULATOR : A Modular Tool for On-the-Fly Equivalence Checking. In Nicolas Halbwachs and Lenore Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* TACAS*'2005 (Edinburgh, Scotland)*, volume 3440 of *Lecture Notes in Computer Science*, pages 581–585. Springer Verlag, April 2005.

[Beh05] Gerd Behrmann. Distributed reachability analysis in timed automata. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 7(1) :19–30, February 2005.

[BFG+91] Ahmed Bouajjani, Jean-Claude Fernandez, Susanne Graf, Carlos Rodríguez, and Joseph Sifakis. Safety for Branching Time Semantics. In *Proceedings of 18th ICALP*. Springer Verlag, July 1991.

[BH05] Alexander Bell and Boudewijn R. Haverkort. Sequential and distributed model checking of Petri nets. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 7(1) :43–60, February 2005.

[BHV00] Gerd Behrmann, Thomas Hune, and Frits W. Vaandrager. Distributing Timed Model Checking - How the Search Order Matters. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification CAV'2000 (Chicago, IL, USA)*, volume 1855 of *Lecture Notes in Computer Science*, pages 216–231. Springer Verlag, July 2000.

[BK84] J. A. Bergstra and J. W. Klop. Process Algebra for Synchronous Communication. *Information and Computation*, 60(1-3) :109–137, March 1984.

[Blo01] Stefan Blom. Partial tau-confluence for efficient state space generation. SEN R0123, CWI, Amsterdam, 2001.

[BLW01a] Benedict Bollig, Martin Leucker, and Michael Weber. Local Parallel Model Checking for the Alternation Free Mu-Calculus. Technical Report AIB-04-2001, Aachen University of Technology, 2001.

[BLW01b] Benedict Bollig, Martin Leucker, and Michael Weber. Parallel Model Checking for the Alternation Free Mu-Calculus. In T. Margaria and W. Yi, editors, *Proceedings of the 7th*

International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'01, volume 2031 of *Lecture Notes in Computer Science*, pages 543–558. Springer Verlag, 2001.

[BLW02] Benedict Bollig, Martin Leucker, and Michael Weber. Local Parallel Model Checking for the Alternation Free Mu-Calculus. In D. Bonaki and S Leue, editors, *Proceedings of the 9th International SPIN Workshop on Model checking of Software (SPIN '02)*, volume 2318 of *Lecture Notes in Computer Science*, pages 128–147. Springer Verlag, 2002.

[BO02] Stefan Blom and Simona Orzan. A Distributed Algorithm for Strong Bisimulation Reduction of State Spaces. In Lubos Brim and Orna Grumberg, editors, *Proceedings of the 1st Workshop on Parallel and Distributed Model Checking PDMC'2002 (Brno, Czech Republic)*, volume 68 of *Electronic Notes in Theoretical Computer Science*, pages 523–538. Elsevier Science Publishers, 2002.

[BO03] Stefan Blom and Simona Orzan. Distributed branching bisimulation reduction of state spaces. In Lubos Brim and Orna Grumberg, editors, *Proceedings of the 2nd International Workshop on Parallel and Distributed Model Checking PDMC'2003 (Boulder, Colorado, USA)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 99–113. Elsevier Science Publishers, 2003.

[BO04] Stefan Blom and Simona Orzan. A Distributed Algorithm for Strong Bisimulation Reduction of State Spaces. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2004.

[BO05] Stefan Blom and Simona Orzan. Distributed State Space Minimization. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2005.

[BRRd96] Amar Bouali, Annie Ressouche, Valérie Roy, and Robert de Simone. The Fc2Tools set : a Toolset for the Verification of Concurrent Systems. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification CAV'96 (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 441–445. Springer Verlag, August 1996.

[BS01] Julian C. Bradfield and Colin Stirling. *Modal logics and mu-calculi : an introduction.* In *Handbook of Process Algebra*, chapter 4, pages 293–330. North-Holland, 2001.

[BvdP02] Stefan Blom and Jaco van de Pol. State Space Reduction by Proving Confluence. In Ed Brinksma and Kim Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification CAV'02 (Copenhagen, Denmark)*, volume 2404 of *Lecture Notes in Computer Science*, pages 596–609. Springer Verlag, July 2002.

[BvLL03] Stefan Blom, Izak van Langevelde, and Bert Lisser. Compressed and Distributed File Formats for Labeled Transition Systems. In Lubos Brim and Orna Grumberg, editors, *Proceedings of the 2nd International Workshop on Parallel and Distributed Model Checking PDMC'2003 (Boulder, Colorado, USA)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 68–83. Elsevier Science Publishers, 2003.

[BZ03] Lubos Brim and Jitka Zidkova. Using Assumptions to Distribute Alternation Free Mu-Calculus Model Checking. In Lubos Brim and Orna Grumberg, editors, *Proceedings of the 2nd International Workshop on Parallel and Distributed Model Checking PDMC'2003 (Boulder, Colorado, USA)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 17–32. Elsevier Science Publishers, 2003.

[CCBF94] S. Caselli, G. Conte, F. Bonardi, and M. Fontanesi. Experiences on SIMD Massively Parallel GSPN Analysis. In G. Haring and G. Kotsis, editors, *Computer Performance Evaluation : Modelling Techniques and Tools*, volume 794 of *Lecture Notes in Computer Science*, pages 266–283. Springer Verlag, 1994.

[CCGR00] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NUSMV : a New Symbolic Model Checker. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2(4) :410–425, April 2000.

[CCM95] S. Caselli, G. Conte, and P. Marenzoni. Parallel State Space Exploration for GSPN Models. In G. De Michelis and M. Diaz, editors, *Proceedings of the 16th International Conference on Applications and Theory of Petri Nets (Torino, Italy)*, volume 935 of *Lecture Notes in Computer Science*, pages 181–200. Springer Verlag, June 1995.

[CCM01] Stefano Caselli, Gianni Conte, and P. Marenzoni. A Distributed Algorithm for GSPN Reachability Graph Generation. *Journal of Parallel Distributed Computing*, 61(1) :79–95, 2001.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2) :244–263, April 1986.

[CGN98] G. Ciardo, J. Gluckman, and D. Nicol. Distributed State Space Generation of Discrete-State Stochastic Models. *INFORMS Journal on Computing*, 10(1) :82–93, 1998.

[Cia01] G. Ciardo. Distributed and Structured Analysis Approaches to Study Large and Complex Systems. In E. Brinksma, H. Hermanns, and J.-P. Katœn, editors, *First EFF/Euro Summer School on Trends in Computer Science (Berg en Dal, The Netherlands)*, volume 2090 of *Lecture Notes in Computer Science*, pages 344–374. Springer Verlag, July 2001.

[Cle90] R. Cleaveland. On Automatically Explaining Bisimulation Inequivalence. In E. M. Clarke and R. P. Kurshan, editors, *Proceedings of the 2nd International Conference on Computer Aided Verification CAV '90 (New Brunswick, New Jersey, USA)*, volume 531 of *Lecture Notes in Computer Science*, pages 364–372, Berlin, June 1990. Springer Verlag.

[CPS89] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench. In J. Sifakis, editor, *Proceedings of the 1st Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, volume 407 of *Lecture Notes in Computer Science*, pages 24–37. Springer Verlag, June 1989.

[CS91a] R. Cleaveland and B. Steffen. Computing behavioural relations, logically. In *Proceedings of the 18th ICALP*, volume 510 of *Lecture Notes in Computer Science*, pages 127–138, 1991.

[CS91b] R. Cleaveland and B. Steffen. A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus. In K. G. Larsen and A. Skou, editors, *Proceedings of 3rd Workshop on Computer Aided Verification CAV '91 (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, pages 48–58, Berlin, July 1991. Springer Verlag.

[CS01] Rance Cleaveland and Oleg Sokolsky. *Equivalence and Preorder Checking for Finite-State Systems.* In *Handbook of Process Algebra*, chapter 6, pages 391–424. North-Holland, 2001.

[DHS03] Salem Derisavi, Holger Hermanns, and William H. Sanders. Optimal state-space lumping in Markov chains. *Information Processing Letters*, 87(6) :309–315, 2003.

[Dic86] A. Dicky. An algebraic and algorithmic method for analysing transition systems. *Theoretical Computer Science*, 46(2-3) :285–303, 1986.

[DSC99] X. Du, S. A. Smolka, and R. Cleaveland. Local Model Checking and Protocol Analysis. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2(3) :219–241, 1999.

[EC80] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th ICALP*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181, Berlin, 1980. Springer Verlag.

[EH86] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited : On branching versus linear time temporal logic. *Journal of the ACM*, 33(1) :151–178, 1986.

[EL86] E. A. Emerson and C-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In *Proceedings of the 1st LICS*, pages 267–278, 1986.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package) : A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag, August 1996.

[FGM+92] Jean-Claude Fernandez, Hubert Garavel, Laurent Mounier, Anne Rasse, Carlos Rodríguez, and Joseph Sifakis. A Toolbox for the Verification of LOTOS Programs. In Lori A. Clarke, editor, *Proceedings of the 14th International Conference on Software Engineering ICSE'14 (Melbourne, Australia)*, pages 246–259. ACM, May 1992.

[FL79] M. J. Fischer and R. E. Ladner. Propositional Dynamic Logic of Regular Programs. *Journal of Computer and System Sciences*, 18(2) :194–211, 1979.

[Fly66] Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12) :1901–1909, December 1966.

[FM90] Jean-Claude Fernandez and Laurent Mounier. Verifying Bisimulations "On the Fly". In Juan Quemada, José Manas, and Enrique Vázquez, editors, *Proceedings of the 3rd International Conference on Formal Description Techniques FORTE'90 (Madrid, Spain)*. Elsevier, North-Holland, November 1990.

[FM91a] Jean-Claude Fernandez and Laurent Mounier. "On the Fly" Verification of Behavioural Equivalences and Preorders. In K. G. Larsen and A. Skou, editors, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, Berlin, July 1991. Springer Verlag.

[FM91b] Jean-Claude Fernandez and Laurent Mounier. A Tool Set for Deciding Behavioral Equivalences. In *Proceedings of CONCUR'91 (Amsterdam, The Netherlands)*, August 1991.

[FS96] Christian Fecht and Helmut Seidl. An Even Faster Solver for General Systems of Equations. In Radhia Cousot and David A. Schmidt, editors, *Proceedings of the 3rd International Symposium on Static Analysis SAS'96 (Aachen, Germany)*, volume 1145 of *Lecture Notes in Computer Science*, pages 189–204. Springer Verlag, September 1996.

[Gar89] Hubert Garavel. *Compilation et vérification de programmes LOTOS*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), November 1989.

[Gar95] Hubert Garavel. Binary Coded Graphs — Definition of the BCG Format (version 1.0). Rapport Technique, INRIA Rhône-Alpes, Grenoble, 1995.

[Gar98] Hubert Garavel. OPEN/CÆSAR : An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84, Berlin, March 1998. Springer Verlag. Full version available as INRIA Research Report RR-3352.

[GH02] Hubert Garavel and Holger Hermanns. On Combining Functional Verification and Performance Evaluation using CADP. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *Proceedings of the 11th International Symposium of Formal Methods Europe FME'2002 (Copenhagen, Denmark)*, volume 2391 of *Lecture Notes in Computer Science*, pages 410–429. Springer Verlag, July 2002. Full version available as INRIA Research Report 4492.

[GK04] Jan Friso Groote and Misa Keinänen. Solving Disjunctive/Conjunctive Boolean Equation Systems with Alternating Fixed Points. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* TACAS *'2004 (Barcelona, Spain)*, volume 2988 of *Lecture Notes in Computer Science*, pages 436–450. Springer Verlag, April 2004.

[GLM02] Hubert Garavel, Frédéric Lang, and Radu Mateescu. An Overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4 :13–24, August 2002. Also available as INRIA Technical Report RT-0254 (December 2001).

[GMB+05] Hubert Garavel, Radu Mateescu, Damien Bergamini, Adrian Curic, Nicolas Descoubes, Christophe Joubert, Irina Smarandache-Sturm, and Gilles Stragier. DISTRIBUTOR and BCG_MERGE : Tools for Distributed Explicit State Space Generation. Tool session of the 4th International Workshop on Parallel and Distributed Methods in Verification PDMC'2005 (Lisboa, Portugal), July 2005.

[GMB+06] Hubert Garavel, Radu Mateescu, Damien Bergamini, Adrian Curic, Nicolas Descoubes, Christophe Joubert, Irina Smarandache-Sturm, and Gilles Stragier. DISTRIBUTOR and BCG_MERGE : Tools for Distributed Explicit State Space Generation. In Holger Hermanns and Jens Palsberg, editors, *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* TACAS *'2006 (Vienna, Austria)*, Lecture Notes in Computer Science. Springer Verlag, april 2006.

[GMS01] Hubert Garavel, Radu Mateescu, and Irina Smarandache. Parallel State Space Construction for Model-Checking. In Matthew B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software SPIN'2001 (Toronto, Canada)*, volume 2057 of *Lecture Notes in Computer Science*, pages 217–234, Berlin, May 2001. Springer Verlag. Revised version available as INRIA Research Report RR-4341 (December 2001).

[GMS05] Jan Friso Groote, François Monin, and Jan Springintveld. A computer checked algebraic verification of a distributed summation algorithm. *Formal Aspects of Computation*, February 2005.

[GP00] J.F. Groote and J. van de Pol. State Space Reduction using Partial τ-Confluence. In Mogens Nielsen and Branislav Rovan, editors, *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science MFCS'2000 (Bratislava, Slovakia)*, volume 1893 of *Lecture Notes in Computer Science*, pages 383–393, Berlin, August 2000. Springer Verlag. Also available as CWI Technical Report SEN-R0008, Amsterdam, March 2000.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the Temporal Analysis of Fairness. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages POPL '80 (Las Vegas, Nevada)*, pages 163–173. ACM Press, January 1980.

[Gro05] Jan Friso Groote. mCRL2 : a Language and Toolset for Behavioural Modelling and Analysis. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects FMCO'2005 (Amsterdam, The Netherlands)*, Lecture Notes in Computer Science. Springer Verlag, November 2005.

[GS86] S. Graf and J. Sifakis. A Logic for the Description of Non-deterministic Programs and Their Properties. *Information and Control*, 68(1–3) :254–270, 1986.

[GS96] J. F. Groote and M. P. A. Sellink. Confluence for process verification. *Theoretical Computer Science*, 170(1-2) :47–81, 1996.

[GV90] Jan Friso Groote and Frits Vaandrager. An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In M. S. Patterson, editor, *Proceedings of the 17th ICALP (Warwick)*, volume 443 of *Lecture Notes in Computer Science*, pages 626–638. Springer Verlag, 1990.

[GvV94] J-F. Groote and S. M. F. van Vlijmen. A modal logic for μCRL. Technical Report 114, Logic Group Preprint Series, Department of Philosophy, Utrecht University, 1994.

[HBB99] B. R. Haverkort, A. Bell, and H. C. Bohnenkamp. On the Efficient Sequential and Distributed Generation of Very Large Markov Chains from Stochastic Petri Nets. In *Proceedings of the 8th International Workshop on Petri Nets and Performance Models PNPM'99 (Zaragoza, Spain)*, pages 12–21. IEEE Computer Society Press, September 1999.

[HD01] John Hatcliff and Matthew B. Dwyer. Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *Proceedings of the 12th International Conference on Concurrency Theory CONCUR'01 (Aalborg, Denmark)*, volume 2154 of *Lecture Notes in Computer Science*, pages 39–58. Springer Verlag, 2001.

[HJ03] Holger Hermanns and Christophe Joubert. A Set of Performance and Dependability Analysis Components for CADP. In Hubert Garavel and John Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* TACAS*'2003 (Warsaw, Poland)*, volume 2619 of *Lecture Notes in Computer Science*, pages 425–430. Springer Verlag, April 2003.

[HK91] S. T. Huang and P. W. Kao. Detecting termination of distributed computations by external agents. *Journal of Information Science and Engineering*, 7(2) :187–201, 1991.

[HLL04] Fredrik Holmén, Martin Leucker, and Marcus Lindström. UppDMC – A Distributed Model Checker for Fragments of the μ-calculus. In Lubos Brim and Martin Leucker, editors, *Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification* PDMC*'2004 (London,* UK*)*, volume 128 of *Electronic Notes in Theoretical Computer Science*, pages 91–105. Elsevier Science Publishers, 2004.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32 :137–161, 1985.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communincation of the ACM*, 21(8) :666–677, 1978.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Software Series. Prentice Hall, 1991.

[Hol97] G. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5) :279–295, May 1997.

[HS99] Holger Hermanns and Markus Siegle. Bisimulation Algorithms for Stochastic Process Algebras and their BDD-based Implementation. In Joost-Pieter Katœn, editor, *Proceedings of the 5th International AMAST Workshop ARTS'99 (Bamberg, Germany)*, volume 1601 of *Lecture Notes in Computer Science*, pages 244–265. Springer Verlag, May 1999.

[IB02] Cornelia P. Inggs and Howard Barringer. On the Parallelisation of Model Checking. In *Proceedings of the 2nd Workshop on Automated Verification of Critical Systems (AVOCS'02)*. Technical report, University of Birmingham, April 2002.

[ISO89] ISO. IS ISO/OSI 9074 - ESTELLE : A formal description technique based on an extended state transition model. International Standard, ISO, 1989.

[ISO01] ISO/IEC. E-LOTOS — Enhancements to LOTOS. International Standard 15437, International Organization for Standardization – Information Technology, Genève, September 2001.

[IT92] ITU-T. Specification and Description Language (SDL). ITU-T Recommendation Z.100, International Telecommunication Union, Genève, 1992.

[JJ05] Claude Jard and Thierry Jéron. TGV : theory, principles and algorithms. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 7(4) :297–315, 2005.

[JM04] Christophe Joubert and Radu Mateescu. Distributed On-the-Fly Equivalence Checking. In Lubos Brim and Martin Leucker, editors, *Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification* PDMC*'2004 (London,* UK*)*, volume 128 of *Electronic Notes in Theoretical Computer Science*, pages 47–62. Elsevier, September 2004.

[JM05] Christophe Joubert and Radu Mateescu. Distributed Local Resolution of Boolean Equation Systems. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network based Processing* PDP*'05 (Lugano, Switzerland)*. IEEE Computer Society Press, February 2005.

[JM06] Christophe Joubert and Radu Mateescu. Distributed On-the-Fly Model-Checking and Test Case Generation. In A. Valmari, editor, *Proceedings of the 13th International SPIN Workshop on Model Checking of Software (SPIN '06)*, volume 3925 of *Lecture Notes in Computer Science*, pages 126–145. Springer Verlag, 2006.

[Jou02] Christophe Joubert. Techniques et outils pour la construction massivement parallèle de systèmes de transitions. Mémoire de DEA en informatique, Institut National Polytechnique de Grenoble and Université Joseph Fourier, Grenoble, June 2002.

[Jou03] Christophe Joubert. Distributed Model Checking : From Abstract Algorithms to Concrete Implementations. In Lubos Brim and Orna Grumberg, editors, *Proceedings of the 2nd International Workshop on Parallel and Distributed Model Checking* PDMC*'2003 (Boulder, Colorado,* USA*)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 114–127. Elsevier, July 2003.

[KC90] S. Kimura and E. M. Clarke. A Parallel Algorithm for Constructing Binary Decision Diagrams. In *Proceedings of the International Conference on Computer-Aided Design ICCD'90 (Cambridge, MA)*, pages 220–223, September 1990.

[KH99] W. J. Knottenbelt and P. G. Harrison. Distributed Disk-Based Solution Techniques for Large Markov Models. In B. Plateau, W. J. Stewart, and M. Silva, editors, *Proceedings of the 3rd International Meeting on the Numerical Solution of Markov Chains NSMC'99 (Zaragoza, Spain)*, pages 58–75. Prensas Universitarias de Zaragoza, September 1999.

[KM04] Rahul Kumar and Eric Mercer. Load Balancing Parallel Explicit State Model Checking. In Lubos Brim and Martin Leucker, editors, *Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification* PDMC*'2004 (London,* UK*)*, volume 128 of *Electronic Notes in Theoretical Computer Science*, pages 19–34. Elsevier, September 2004.

[KMHK98] W. J. Knottenbelt, M. A. Mestern, P. G. Harrison, and P. Kritzinger. Probability, Parallelism and the State Space Exploration Problem. In R. Puigjaner, N. N. Savino, and B. Serra, editors, *Proceedings of the 10th International Conference on Computer Performance Evaluation - Modelling, Techniques and Tools TOOLS'98 (Palma de Mallorca, Spain)*, volume 1469 of *Lecture Notes in Computer Science*, pages 165–179. Springer Verlag, September 1998.

[KNP02] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM : Probabilistic Symbolic Model Checker. In Tony Field, Peter G. Harrison, Jeremy T. Bradley, and Uli Harder, editors, *Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools TOOLS'2002 (London, UK)*, volume 2324 of *Lecture Notes in Computer Science*, pages 200–204. Springer Verlag, April 2002.

[Koz83] D. Kozen. Results on the Propositional μ-calculus. *Theoretical Computer Science*, 27 :333–354, 1983.

[KS90] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1) :43–68, May 1990.

[Lam80] L. Lamport. "Sometime" is sometimes "not never". On the temporal logic of programs. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages POPL '80 (Las Vegas, Nevada)*, pages 163–173. ACM Press, January 1980.

[Lam83] Leslie Lamport. What Good is Temporal Logic ? In *Proceedings of the IFIP 9th World Computer Congress IFIP'83 (Paris, France)*, volume 83 of *Information Processing*, pages 657–668, September 1983.

[Lar88] K. G. Larsen. Proof Systems for Hennessy-Milner logic with Recursion. In *Proceedings of the 13th Colloquium on Trees in Algebra and Programming CAAP '88 (Nancy, France)*, volume 299 of *Lecture Notes in Computer Science*, pages 215–230, Berlin, March 1988. Springer Verlag.

[Lar92] K. G. Larsen. Efficient Local Correctness Checking. In G. v. Bochmann and D. K. Probst, editors, *Proceedings of 4th International Workshop in Computer Aided Verification CAV '92 (Montréal, Canada)*, volume 663 of *Lecture Notes in Computer Science*, pages 30–43, Berlin, June 1992. Springer Verlag.

[LS98] X. Liu and S. A. Smolka. Simple Linear-Time Algorithms for Minimal Fixed Points. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming ICALP'98 (Aalborg, Denmark)*, volume 1443 of *Lecture Notes in Computer Science*, pages 53–66. Springer Verlag, July 1998.

[LS99] F. Lerda and R. Sisto. Distributed-Memory Model Checking with SPIN. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking SPIN'99*, volume 1680 of *Lecture Notes in Computer Science*, pages 22–39. Springer Verlag, July 1999.

[LSW03] Martin Leucker, Rafal Somla, and Michael Weber. Parallel Model Checking for LTL, CTL* and L^2_μ. In Lubos Brim and Orna Grumberg, editors, *Proceedings of the 2nd International Workshop on Parallel and Distributed Model Checking PDMC'2003 (Boulder, Colorado, USA)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 4–16. Elsevier Science Publishers, 2003.

[LY94] David Lee and Mihalis Yannakakis. Testing Finite-State Machines : State Identification and Verification. *IEEE Transactions on Computers*, 43(3) :306–320, march 1994.

[Mad97] Angelika Mader. *Verification of Modal Properties Using Boolean Equation Systems*. VERSAL 8, Bertz Verlag, Berlin, 1997.

[Mat87] F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2 :161–175, 1987.

[Mat98] Radu Mateescu. *Vérification des propriétés temporelles des programmes parallèles*. Thèse de Doctorat, Institut National Polytechnique de Grenoble, April 1998.

[Mat00] Radu Mateescu. Efficient Diagnostic Generation for Boolean Equation Systems. In Susanne Graf and Michael Schwartzbach, editors, *Proceedings of 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2000 (Berlin, Germany)*, volume 1785 of *Lecture Notes in Computer Science*, pages 251–265. Springer Verlag, March 2000. Full version available as INRIA Research Report RR-3861.

[Mat02] Radu Mateescu. Local Model-Checking of Modal Mu-Calculus on Acyclic Labeled Transition Systems. In Joost-Pieter Katœn and Perdita Stevens, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2002 (Grenoble, France)*, volume 2280 of *Lecture Notes in Computer Science*, pages 281–295. Springer Verlag, April 2002. Full version available as INRIA Research Report RR-4430.

[Mat03a] Radu Mateescu. A Generic On-the-Fly Solver for Alternation-Free Boolean Equation Systems. In Hubert Garavel and John Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2003 (Warsaw, Poland)*, volume 2619 of *Lecture Notes in Computer Science*, pages 81–96. Springer Verlag, April 2003. Full version available as INRIA Research Report RR-4711.

[Mat03b] Radu Mateescu. Logiques temporelles basées sur actions pour la vérification des systèmes asynchrones. *Technique et Science Informatiques*, 22(4) :461–495, 2003. Full version available as INRIA Research Report RR-5032.

[Mat05] Radu Mateescu. On-the-Fly State Space Reductions for Weak Equivalences. In Stefania Gnesi, Tiziana Margaria, and Mieke Massink, editors, *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2005 (Lisbon, Portugal)*, September 2005.

[Mat06] Radu Mateescu. CAESAR_SOLVE : A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2006.

[MC98] J. Matocha and T. Camp. A taxonomy of distributed termination detection algorithms. *Journal of Systems and Software*, 43 :207–221, 1998.

[MCC97] P. Marenzoni, S. Caselli, and G. Conte. Analysis of Large GSPN Models : A Distributed Solution Tool. In *Proceedings of the 7th IEEE International Workshop on Petri Nets and Performance Models PNPM'97 (Saint Malo, France)*, pages 122–131. IEEE Computer Society Press, 1997.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mou92] Laurent Mounier. *Méthodes de vérification de spécifications comportementales : étude et mise en œuvre.* Thèse de Doctorat, Université Joseph Fourier (Grenoble), January 1992.

[MP90] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing PODC'90 (Quebec City, Quebec, Canada)*, pages 377–410. ACM Press, August 1990.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, volume I : Specification.* Springer-Verlag, 1992.

[MPI] MPI. http ://www-unix.mcs.anl.gov/mpi/.

[MS93] Eric Madelaine and Robert de Simone. *FC2 : Reference Manual Version 1.1.* INRIA, Sophia-Antipolis (France), July 1993.

[MS03] Radu Mateescu and Mihaela Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3) :255–281, March 2003.

[NC97] D. Nicol and G. Ciardo. Automated Parallelization of Discrete State-Space Generation. *Journal of Parallel and Distributed Computing*, 47(2) :153–167, 1997.

[NFGR91] R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An action-based framework for verifying logical and behavioural properties of concurrent systems. In K. G. Larsen and A. Skou, editors, *Proceedings of 3rd Workshop on Computer Aided Verification CAV'91 (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, pages 37–47, Berlin, July 1991. Springer Verlag.

[NG98] Ratan Nalumasu and Ganesh Gopalakrishnan. PV : An Explicit Enumeration Model-Checker. In *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design FMCAD'98 (Palo Alto, California, USA)*, volume 1522 of *Lecture Notes in Computer Science*, pages 523–528. Springer-Verlag, November 1998.

[NG02] Ratan Nalumasu and Ganesh Gopalakrishnan. Deriving Efficient Cache Coherence Protocols Through Refinement. *Formal Methods in System Design*, 20(1) :107–125, 2002.

[NV90] R. De Nicola and F. W. Vaandrager. *Action versus State based Logics for Transition Systems.* In Irène Guessarian, editor, *Semantics of Systems of Concurrent Processes*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer Verlag, April 1990.

[Orz04] Simona Orzan. *On distributed verification and verified distribution.* PhD thesis, Free University Amsterdam, November 2004.

[OvdP05] Simona Orzan and Jaco van de Pol. Detecting strongly components in large distributed state spaces. Technical Report SEN-E0501, CWI, Amsterdam, January 2005.

[OvdPE04] Simona Orzan, Jaco van de Pol, and Miguel Valero Espada. Distributed Partial Order Reduction of State Spaces. 128 :25–45, 2004.

[Par81] David Park. Concurrency and Automata on Infinite Sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Verlag, March 1981.

[PBY+05] Robert Palmer, Steve Barrus, Yu Yang, Ganesh Gopalakrishnan, and Robert M. Kirby. Gauss : A Framework for Verifying Scientific Computing Software. 2005.

[Pel96] Doron Peled. Combining Partial Order Reductions with On-the-Fly Model-Checking. *Formal Methods in System Design*, 8(1) :39–64, 1996.

[PG02] Robert Palmer and Ganesh Gopalakrishnan. Partial Order Reduction Assisted Parallel Model-Checking. Technical Report, University of Utah, August 2002.

[PLM03] Gordon Pace, Frédéric Lang, and Radu Mateescu. Calculating τ-Confluence Compositionally. In Jr Warren A. Hunt and Fabio Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification CAV'2003 (Boulder, Colorado, USA)*, volume 2725 of *Lecture Notes in Computer Science*, pages 446–459. Springer Verlag, July 2003. Full version available as INRIA Research Report RR-4918.

[PSC94] Y. Parasuram, E. Stabler, and S. K. Chin. Parallel Implementation of BDD Algorithms Using a Distributed Shared Memory. In *Proceedings of the 27th Hawaii International Conference on System Sciences Vol I : Architecture*, pages 16–25, January 1994.

[PT87] Robert Paige and Robert E. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal on Computing*, 16(6) :973–989, December 1987.

[QS83] Jean-Pierre Queille and Joseph Sifakis. Fairness and Related Properties in Transition Systems — A Temporal Logic to Deal with Fairness. *Acta Informatica*, 19 :195–220, 1983.

[RS97] Y.S. Ramakrishna and S.A. Smolka. Partial-Order Reduction in the Weak Modal Mu-Calculus. In A. Mazurkiewicz and J. Winkowski, editors, *Proceedings of the 8th International Conference on Concurrency Theory CONCUR'97*, volume 1243 of *Lecture Notes in Computer Science*, pages 5–24. Springer Verlag, 1997.

[RSBSV96] R. K. Ranjan, J. V. Sanghavi, R. K. Brayton, and A. Sangiovanni-Vincentelli. Binary Decision Diagrams on Network of Workstations. In *Proceedings of the IEEE International Conference on Computer Design (Austin, Texas, USA)*, pages 358–364. IEEE Computer Society Press, October 1996.

[SB96] T. Stornetta and F. Brewer. Implementation of an Efficient Parallel BDD Package. In *Proceedings of the 33rd Annual IEEE Conference on Design Automation (Las Vegas, Nevada, USA)*, pages 641–644. IEEE Computer Society Press, June 1996.

[SD97] Ulrich Stern and David L. Dill. Parallelizing the Murφ Verifier. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer-Aided Verification CAV'97 (Haifa, Israel)*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–267. Springer Verlag, June 1997.

[SD04] Jeremy Sproston and Susanna Donatelli. Backward Stochastic Bisimulation in CSL Model Checking. In *Proceedings of the 1st International Conference on Quantitative Evaluation of Systems QEST'2004 (Enschede,The Netherlands)*, pages 220–229. IEEE Computer Society, 2004.

[SI94] Bernhard Steffen and Anna Ingolfsdottir. Characteristic formulae for processes with divergence. *Information and Computation*, 110(1) :149–163, April 1994.

[SS98] Perdita Stevens and Colin Stirling. Practical Model-Checking Using Games. In Bernhard Steffen, editor, *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* TACAS *'1998 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 85–101. Springer Verlag, Apr 1998.

[SS05] Gwen Salaün and Wendelin Serwe. Translating Hardware Process Algebras into Standard Process Algebras — Illustration with CHP and LOTOS. In Jaco van de Pol, Judi Romijn, and Graeme Smith, editors, *Proceedings of the 5th International Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands)*, Lecture Notes in Computer Science. Springer Verlag, November 2005. Full version available as INRIA Research Report RR-5666.

[Sti95] Colin Stirling. Lokal Model Checking Games. In Insup Lee and Scott A. Smolka, editors, *Proceedings of the 6th International Conference on Concurrency Theory CONCUR'1997*

(Philadelphia, USA), volume 962 of *Lecture Notes in Computer Science*, pages 1–11. Springer Verlag, 1995.

[Tar55] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, (5) :285–309, 1955.

[Tre96] Jan Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software - Concepts and Tools*, 17(3) :103–120, 1996.

[vG90] R. J. van Glabbeek. The linear time-branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings of Theories of Concurrency : Unification and Extension CONCUR'90 (Amsterdam, The Netherlands)*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer Verlag, august 1990.

[vGW96] R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3) :555–600, 1996.

[VL92] B. Vergauwen and J. Lewi. A linear algorithm for solving fixed-point equations on transition systems. In *Proceedings of the 17th Colloquium on Trees in Algebra and Programming CAAP '92 (Rennes, France)*, volume 581 of *Lecture Notes in Computer Science*, pages 322–341, Berlin, February 1992. Springer Verlag.

[VL94] B. Vergauwen and J. Lewi. Efficient Local Correctness Checking for Single and Alternating Boolean Equation Systems. In S. Abiteboul and E. Shamir, editors, *Proceedings of the 21st ICALP (Vienna)*, volume 820 of *Lecture Notes in Computer Science*, pages 304–315, Berlin, July 1994. Springer Verlag.

[VLT03] VLTS. *http ://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html*. INRIA/VASY and CWI/SEN2, 2003.

[YBO+98] Bwolen Yang, Randal E. Bryant, David R. O'Hallaron, Armin Biere, Olivier Coudert, Geert Janssen, Rajeev K. Ranjan, and Fabio Somenzi. A Performance Study of BDD-Based Model Checking. In *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design FMCAD'98 (Palo Alto, California, USA)*, volume 1522 of *Lecture Notes in Computer Science*, pages 255–289. Springer-Verlag, January 1998.

[Yin00] Mingsheng Ying. Weak confluence and tau-inertness. *Theoretical Computer Science*, 238(1-2) :465–475, 2000.

[ZS05] Roberto Ziller and Klaus Schneider. Combining Supervisor Synthesis and Model Checking. *Trans. on Embedded Computing Sys.*, 4(2) :331–362, 2005.

Liste des figures

Liste des tables

Index