INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Modélisation et analyse de systèmes asynchrones avec CADP

Radu Mateescu

N° 5953

Juillet 2006

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--5953--FR+ENG

Modélisation et analyse de systèmes asynchrones avec CADP

Radu Mateescu*

Thème COM — Systèmes communicants
Projet VASY

Rapport de recherche n° 5953 — Juillet 2006 — 31 pages

Résumé : La conception des systèmes industriels critiques comportant du parallélisme asynchrone nécessite l'utilisation de méthodes formelles, assistées par des outils de vérification adaptés, afin de détecter et corriger les erreurs le plus tôt possible. Dans ce rapport, nous illustrons l'emploi de la boîte à outils CADP pour la modélisation et la vérification formelle de tels systèmes, à travers l'exemple d'une unité dédiée au perçage des pièces métalliques. Nous décrivons en langage LOTOS deux versions différentes de l'unité, régies par un contrôleur principal séquentiel, respectivement parallèle. Ensuite, nous effectuons la génération et la minimisation des deux espaces d'états sous-jacents, ainsi que l'inspection visuelle de celui, plus petit, correspondant à la version équipée du contrôleur séquentiel. Finalement, nous analysons le comportement des deux versions de l'unité de perçage en employant deux méthodes de vérification complémentaires, basées sur les bisimulations (*equivalence checking*) et les logiques temporelles (*model checking*).

Mots-clés : algèbre de processus, bisimulation, logique temporelle, LOTOS, model checking, mu-calcul, spécification, système d'équations booléennes, système de transitions étiquetées, vérification

Une version abrégée de ce rapport est également disponible comme "Modélisation et analyse de systèmes asynchrones avec CADP", *Systèmes temps réel 1 — techniques de description et de vérification*, Traité IC2, chapitre 5, Lavoisier, 2006.

* Radu.Mateescu@inria.fr

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Modeling and Analysis of Asynchronous Systems using CADP

Abstract: The design of industrial critical systems involving asynchronous parallelism requires the use of formal methods, assisted by appropriate verification tools, in order to detect and correct errors as early as possible. In this report, we illustrate the use of the CADP toolbox for the formal modeling and verification of such systems by considering as an example a unit dedicated to the drilling of metal products. We describe in the LOTOS language two different versions of the unit, supervised by a sequential and a parallel controller, respectively. Then, we perform the generation and minimisation of the two underlying state spaces, and also the visual inspection of the smaller one, corresponding to the version equipped with a sequential controller. Finally, we analyse the behaviour of the two versions of the drilling unit by means of two complementary verification methods, based on bisimulations (*equivalence checking*) and temporal logics (*model checking*).

Key-words: bisimulation, boolean equation system, labeled transition system, LOTOS, model checking, mu-calculus, partial order reduction, process algebra, specification, temporal logic, verification

1 Introduction

Les systèmes industriels comportant du parallélisme asynchrone, tels que les protocoles de communication, les logiciels embarqués et les architectures matérielles multiprocesseurs, sont caractérisés par une grande complexité. En même temps, ils ont souvent un caractère critique, car un dysfonctionnement de leur part peut entraîner la perte de vies humaines ou des dégâts considérables. C'est pourquoi, les éventuelles erreurs doivent être détectées le plus tôt possible dans le processus de conception de ces systèmes, ce qui rend indispensable l'utilisation de méthodes de description et vérification formelle, assistées par des outils informatiques adaptés.

La méthode de vérification basée sur les modèles (*explicit-state verification*) offre un bon compromis coût-performances, ce qui a entraîné son utilisation avec succès en milieu industriel. Cette méthode consiste à construire, à partir d'une description formelle du système, un modèle sémantique (espace d'états) sur lequel les propriétés de bon fonctionnement, exprimées au moyen d'un formalisme approprié (automates ou logiques temporelles), sont vérifiées grâce à des algorithmes spécifiques. La vérification basée sur les modèles permet une détection rapide et économique des erreurs dans des systèmes complexes, étant particulièrement utile pendant les premières phases du processus de conception, quand les erreurs sont susceptibles d'être les plus fréquentes.

Les systèmes industriels complexes comportent habituellement des parties asynchrones, consistant en plusieurs entités physiquement distribuées, qui communiquent et se synchronisent par échange de messages, ainsi que des parties temps réel "dur", régies par des contraintes temporelles fortes (par exemple, respect d'échéances). Dans ce rapport, nous abordons uniquement les aspects asynchrones, la modélisation et l'analyse des aspects temps réel dur pouvant être effectuées par des approches spécifiques, comme les automates temporisés et l'outil Uppaal [BDL04] ou les réseaux de Petri temporels et l'outil Tina [BV06].

La boîte à outils Cadp [GLM02] pour l'ingénierie des systèmes asynchrones offre une large gamme de fonctionnalités permettant d'assister efficacement le processus de conception : modélisation, simulation, prototypage rapide, vérification et génération de tests. Les outils sous-jacents ont été conçus suivant une architecture modulaire, centrée autour de l'environnement générique Open/Cæsar [Gar98] pour l'exploration à la volée des espaces d'états, ce qui assure leur indépendance vis-à-vis du langage de description et favorise la réutilisation des composants grâce à des interfaces bien définies. Bien que certaines fonctionnalités pour l'évaluation de performances ont été récemment ajoutées à Cadp [GH02, HJ03], nous illustrons ici uniquement la vérification fonctionnelle, qui prend en compte l'enchaînement logique des événements au cours de l'exécution du système.

Afin de montrer la démarche de modélisation et d'analyse promue par Cadp, nous considérons un système industriel critique dédié au perçage de pièces métalliques. Plus précisément, nous détaillons la conception du contrôleur informatique chargé de piloter les différents dispositifs physiques composant l'unité de perçage. Ce système a servi comme cas étude pour comparer plusieurs langages de description par rapport à leur adéquation et à la puissance des outils de vérification associés [BK02, BTW+05].

Ce rapport est structuré de la manière suivante. La section 2 décrit brièvement le langage Lotos, ainsi que quelques outils de vérification de Cadp utilisés dans cette étude. Ensuite, la modélisation de l'unité de perçage en Lotos est détaillée en section 3 et sa vérification fonctionnelle au moyen d'équivalences et de logiques temporelles est présentée en section 4. Enfin, la section 5 conclut le rapport et précise quelques directions de recherche actuelles sur les techniques de vérification.

2 La boîte à outils CADP

Cadp[1] (*Construction and Analysis of Distributed Processes*) [GLM02] est une boîte à outils très performante pour la modélisation et la vérification de systèmes parallèles asynchrones. Ces systèmes comportent plusieurs entités (processus ou agents) qui s'exécutent en parallèle, communiquent et se synchronisent par échange de messages. Pour modéliser l'exécution de ces systèmes, Cadp utilise la sémantique d'entrelacement des actions (événements), qui repose sur le fait que chaque action est atomique et qu'une seule action peut être observée à un instant donné. Des exemples de systèmes asynchrones sont les protocoles de télécommunication, les systèmes d'exploitation, les bases de données distribuées, les architectures matérielles multiprocesseurs et les logiciels embarqués.

2.1 Le langage LOTOS

Cadp accepte en entrée plusieurs formalismes de description, allant des langages de haut niveau, comme Lotos, jusqu'à des langages plus basiques, comme les réseaux d'automates communicants. Lotos (*Language Of Temporal Ordering Specification*) [ISO89] est une technique de description formelle normalisée par l'Iso. Bien que défini à l'origine pour la description des protocoles de télécommunication selon le modèle Osi, le langage Lotos s'est avéré tout aussi adapté pour décrire d'autres classes de systèmes asynchrones, comme celles mentionnées plus haut. Lotos comporte deux parties "orthogonales" :

Une partie "données", basée sur les types abstraits algébriques, et plus particulièrement sur le langage ActOne [EM85]. Cette partie permet de décrire les structures de données manipulées par le système au moyen de sortes et d'opérations algébriques définies de manière équationnelle.

Une partie "contrôle", basée sur les meilleures primitives des algèbres de processus Ccs [Mil89] et Csp [BHR84]. Cette partie permet de décrire les processus parallèles composant un système sous forme de termes construits en appliquant des opérateurs algébriques (préfixage, mise en parallèle avec synchronisation par rendez-vous, abstraction, etc.).

Cadp contient deux compilateurs pour Lotos : Cæsar.Adt [Gar89] traduit la partie données d'une description Lotos en langage C (les sortes et les opérations Lotos étant respectivement traduites vers des types et des fonctions C) et Cæsar [GS90] traduit la partie contrôle vers un programme C qui peut être embarqué dans un système réel ou utilisé à des fins de simulation, vérification ou génération de tests.

[1]Voir la distribution en ligne `http ://www.inrialpes.fr/vasy/cadp`

2.2 Systèmes de transitions étiquetées

Les systèmes de transitions étiquetées (STEs) constituent le modèle sémantique sous-jacent aux formalismes de description utilisés par CADP. Un STE M est un quadruplet (S, A, T, s_0) comprenant un ensemble d'états S, un ensemble d'étiquettes A représentant des actions, une relation de transition $T \subseteq S \times A \times S$ et un état initial $s_0 \in S$. L'action invisible $\tau \notin A$ permet de modéliser les comportements internes (non observables) du système. Une transition $(s_1, a, s_2) \in T$, également notée $s_1 \xrightarrow{a} s_2$, signifie que dans l'état s_1, l'action a fait passer le système dans l'état s_2. CADP offre deux représentations complémentaires pour les STEs :

Une représentation explicite, sous forme de la liste de transitions du STE, mémorisée dans un fichier au format binaire compact, appelé BCG (*Binary Coded Graphs*). L'environnement BCG fournit un ensemble d'outils et bibliothèques permettant la manipulation des fichiers BCG (lecture/écriture, visualisation, minimisation, conversion vers d'autres formats, etc.). Cette représentation explicite des STEs est appropriée pour la vérification *globale* (ou énumérative), qui procède par une exploration en arrière de la relation de transition et, par conséquent, requiert la construction préalable du STE dans sa totalité.

Une représentation implicite, sous forme de la fonction successeur du STE, codée comme un programme C respectant une interface de programmation définie par l'environnement OPEN/CÆSAR [Gar98]. Outre les types C implémentant les états, actions et transitions du STE, munis de fonctions associées (comparaison, hachage, énumération des états successeurs, etc.), OPEN/CÆSAR fournit également des bibliothèques de primitives pour l'exploration à la volée de STEs (tables d'états, listes de transitions, piles, etc.). Cette représentation implicite des STEs est appropriée pour la vérification *locale* (ou à la volée), qui procède par une exploration en avant de la relation de transition et, par conséquent, autorise une construction incrémentale[2] du STE.

La vérification à la volée constitue un moyen simple pour combattre l'explosion d'états (taille prohibitive du STE pour les systèmes comportant beaucoup de processus parallèles et des types de données complexes), permettant la détection d'erreurs même lorsque la construction complète du STE excède les ressources de calcul disponibles.

2.3 Quelques outils de vérification

CADP offre une large panoplie d'outils dédiés à l'analyse des STEs, couvrant tout le spectrum de fonctionnalités nécessaires pour assister le processus de développement : simulation interactive et guidée par des objectifs, exécution aléatoire, minimisation modulo des relations d'équivalence, réduction par ordres partiels, vérification par équivalences et par logiques temporelles, génération de tests de conformité. Nous présentons brièvement ci-dessous quelques outils de CADP que nous avons utilisé dans cette étude et dont le fonctionnement sera illustré par la suite.

[2]Bien entendu, un STE explicite déjà construit peut être exploré à la volée ; en CADP cela est réalisé au moyen de l'outil BCG_OPEN, qui implémente une représentation des fichiers BCG compatible avec l'interface OPEN/CÆSAR.

BCG_MIN effectue la minimisation d'un STE, représenté sous forme de fichier BCG, modulo une relation d'équivalence, telle que la bisimulation forte ou la bisimulation de branchement. BCG_MIN permet également de traiter les STEs probabilistes et stochastiques [GH02].

CAESAR_SOLVE [Mat03, Mat06] est une bibliothèque générique pour la résolution à la volée des systèmes d'équations booléennes (SEBs) d'alternance 1. Ces SEBs comportent plusieurs blocs d'équations ayant en partie gauche des variables booléennes et en partie droite des formules propositionnelles sur les variables contenant seulement des disjonctions et des conjonctions. Chaque bloc d'équations représente le plus petit ou le plus grand point fixe de la fonctionnelle prenant en entrée les variables situées en partie gauche des équations et renvoyant les valeurs des formules situées en partie droite des équations. Une variable définie en partie gauche d'une équation dépend des variables présentes en partie droite de l'équation. Un bloc dépend d'un autre s'il définit une variable qui dépend de variables définies dans l'autre bloc ; la condition d'alternance 1 signifie l'absence de dépendances circulaires entre les blocs. Cette classe de SEBs bénéficie d'algorithmes de résolution ayant une complexité en temps et en mémoire linéaire par rapport à la taille du SEB (nombre de variables et d'opérateurs), tout en étant suffisamment générale pour représenter plusieurs types d'analyses sur les STEs (vérification par équivalences et par logiques temporelles, réduction par ordres partiels, etc.).

La résolution d'un SEB à la volée consiste à calculer la valeur d'une variable booléenne d'intérêt en explorant de manière incrémentale uniquement la partie du SEB nécessaire pour calculer la valeur de cette variable. Les algorithmes sous-jacents peuvent être développés de manière plus intuitive en représentant les SEBs sous forme de *graphes booléens* [And94], dont les sommets et les arcs dénotent respectivement les variables booléennes et les dépendances définies par les équations. Formulés dans ce contexte, les algorithmes de résolution effectuent une exploration en avant du graphe booléen à partir de la variable d'intérêt, entrelacée avec une propagation en arrière des variables stables (dont la valeur a été calculée) suivant les dépendances.

CÆSAR_SOLVE offre actuellement quatre algorithmes de résolution à la volée, ayant une complexité linéaire en taille du SEB. Les algorithmes A1 et A2, basés respectivement sur des parcours du graphe booléen en profondeur et en largeur, peuvent résoudre des SEBs généraux, sans imposer des contraintes sur les formules booléennes situées en partie droite des équations. Les algorithmes A3 et A4, basés sur des parcours en profondeur, sont spécialisés pour résoudre des SEBs sans circuit, respectivement disjonctifs/conjonctifs — rencontrés souvent en pratique — avec une consommation mémoire réduite. Ces algorithmes de résolution produisent également des *diagnostics*, c'est à dire des portions du graphe booléen illustrant le résultat de la résolution, selon l'approche proposée en [Mat00].

CÆSAR_SOLVE définit une représentation implicite des graphes booléens en langage C semblable à celle des STEs définie par OPEN/CÆSAR : le type C codant les variables booléennes est équipé des primitives nécessaires aux parcours du graphe booléen (comparaison et hachage des variables, énumération des variables successeurs, etc.). Les diagnostics des résolutions sont également fournis en termes de sous-graphes booléens représentés de manière implicite par leur fonction successeur. CÆSAR_SOLVE sert actuellement dans CADP comme moteur de résolution pour plusieurs outils de vérification à la volée, dont nous mentionnons deux ci-dessous.

BISIMULATOR [Mat03, Mat06] effectue la comparaison à la volée de deux STEs par rapport à une relation d'équivalence ou de préordre. Le premier STE, en forme implicite, représenté par un programme OPEN/CÆSAR, dénote le comportement d'un système (protocole), tandis que le deuxième STE, en forme explicite, représenté par un fichier BCG, dénote le comportement externe (service) attendu de la part du système. BISIMULATOR implémente sept relations d'équivalence entre STEs : quatre bisimulations (forte, de branchement, observationnelle et $\tau^*.a$) et trois équivalences de simulation (de sûreté et de trace, avec ses variantes forte et faible), ce qui en fait un des outils de vérification à la volée les plus riches disponibles à l'heure actuelle. Pour chaque relation, l'outil offre la vérification de l'équivalence et du préordre correspondant (inclusion d'un STE dans l'autre).

La méthode utilisée par BISIMULATOR consiste à traduire le problème de vérification vers la résolution d'un SEB contenant un seul bloc d'équations de plus grand point fixe, directement dérivé à partir de la définition mathématique de la relation d'équivalence. L'outil est structuré en deux parties indépendantes : une "partie avant", chargée de traduire la comparaison modulo une relation d'équivalence en termes d'un SEB et d'interpréter le diagnostic de la résolution en termes des deux STEs à comparer, et une "partie arrière" (bibliothèque CÆSAR_SOLVE), chargée de calculer la variable d'intérêt, qui représente le fait que les états initiaux des deux STEs sont équivalents ou inclus modulo le préordre considéré.

Cette architecture modulaire facilite l'ajout de nouvelles relations d'équivalence (chaque relation est implémentée dans un module indépendant, contenant la traduction en SEB et l'interprétation des diagnostics) et en même temps ne pénalise pas les performances, BISIMULATOR s'avérant plus efficace que des implémentations d'algorithmes spécialisés de vérification à la volée par équivalences [BDJM05]. Les diagnostics (contre-exemples) fournis par l'outil en cas de non-équivalence (ou non-inclusion) des deux STEs sont des sous-graphes sans circuit contenant des séquences d'actions qui, exécutées simultanément dans les deux STEs, conduisent à des états non équivalents.

BISIMULATOR emploie tous les algorithmes fournis par CÆSAR_SOLVE : A1 et A2 s'appliquent à toutes les équivalences (l'algorithme A2, basé sur un parcours en largeur, présente l'avantage pratique de fournir des contre-exemples de profondeur réduite) ; A3, optimisé en mémoire pour résoudre les SEBs sans circuit, sert à vérifier l'inclusion de séquences ou d'arbres d'exécution dans un STE ; enfin A4, optimisé en mémoire pour résoudre les SEBs conjonctifs, est utile lorsqu'un des deux STEs est déterministe (pour l'équivalence forte) et ne possède pas de transitions invisibles (pour les équivalences faibles).

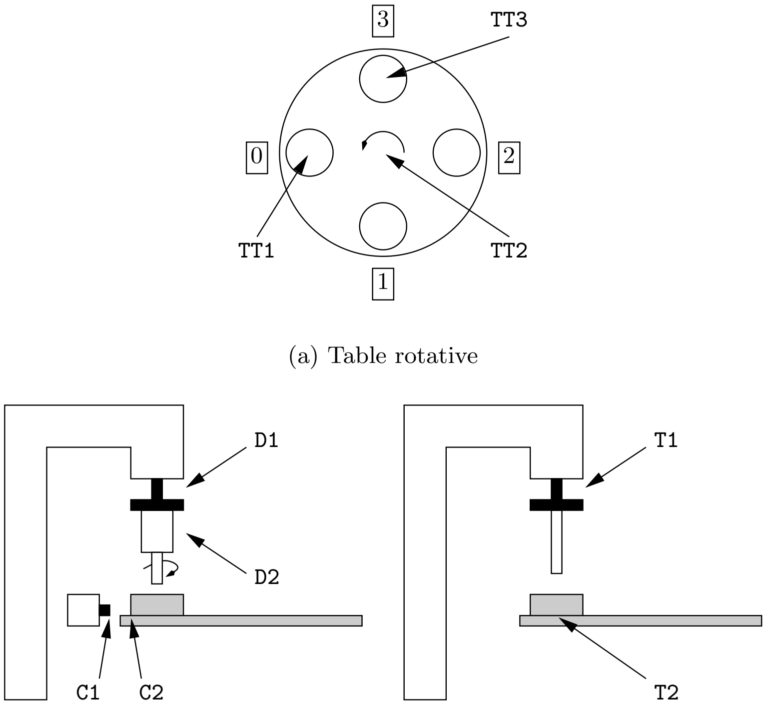
EVALUATOR 3.5 [Mat03, Mat06] effectue l'évaluation à la volée d'une formule de logique temporelle sur un STE. La logique acceptée en entrée est le μ-calcul régulier d'alternance 1 [MS03], constitué des opérateurs booléens, des modalités de possibilité et de nécessité contenant des expressions régulières sur les séquences d'actions (similaires à celles de PDL [FL79]) et des opérateurs de plus petit et de plus grand point fixe du μ-calcul modal [Koz83]. La condition d'alternance 1, signifiant l'absence de récursion mutuelle entre les formules de plus petit et de plus grand point fixe, permet d'obtenir des algorithmes de vérification ayant une complexité linéaire en taille de la formule (nombre d'opérateurs) et du STE (nombre d'états et de transitions). Le μ-calcul régulier d'alternance 1 autorise une description concise et intuitive des propriétés classiques sur les STEs (sûreté, vivacité, ainsi que certaines formes d'équité). L'outil permet également de définir des bibliothèques réutilisables d'opérateurs temporels dérivés, comme ceux d'ACTL (*Action-Based* CTL) [NV90] ou les schémas génériques de propriétés du catalogue proposé en [DAC99].

La méthode utilisée par EVALUATOR 3.5 consiste à traduire le problème de vérification vers la résolution d'un SEB d'alternance 1 contenant un bloc d'équations pour chaque opérateur temporel contenu dans la formule. Le SEB est obtenu après plusieurs phases de traitement de la formule (traduction en forme normale positive, élimination des opérateurs dérivés, traduction vers des systèmes d'équations modales, élimination des expressions régulières contenues dans les modalités, simplification). L'outil est structuré en deux parties indépendantes : une "partie avant", chargée de traduire l'évaluation de la formule en termes d'un SEB et d'interpréter le diagnostic de la résolution en termes du STE à vérifier, et une "partie arrière" (bibliothèque CÆSAR_SOLVE), chargée de calculer la variable d'intérêt, qui représente le fait que l'état initial du STE satisfait la formule. Les diagnostics (exemples et contre-exemples) fournis par l'outil sont des sous-graphes du STE illustrant la valeur de vérité de la formule sur l'état initial du STE.

EVALUATOR 3.5 emploie tous les algorithmes fournis par CÆSAR_SOLVE : A1 et A2 s'appliquent à toutes les formules du μ-calcul régulier d'alternance 1 (l'algorithme A2, basé sur un parcours en largeur, présente l'avantage pratique de fournir des diagnostics de profondeur réduite) ; A3, spécialisé pour résoudre des SEBs sans circuit, sert à vérifier toute formule du μ-calcul modal sur des STEs sans circuit, comme lesscénarios d'exécution ou de simulation ; enfin A4, optimisé en mémoire pour résoudre les SEBs disjonctifs/conjonctifs, s'applique aux formules d'ACTL et aux modalités munies d'expressions régulières, fréquemment rencontrées en pratique.

3 Modélisation d'une unité de perçage

Nous développons dans cette section un modèle en LOTOS d'une unité de perçage des pièces métalliques. Cet exemple de système industriel critique [BK02, BTW+05] a servi de support pour expérimenter les capacités de modélisation de différents langages de description (χ [SvBM+03], Promela [Hol03], automates temporisés [BDL04], μCRL [Gro97]) et les fonctionnalités des outils de vérification associés (SPIN [Hol03], UPPAAL [BDL04], CADP). La modélisation en LOTOS ci-dessous est basée sur la description en langage χ proposée initialement en [BK02], avec quelques détails (tels que la présence du capteur `TT3`) inspirés de la description plus élaborée présentée en [BTW+05]. L'unité de perçage, illustrée dans la figure 1, comporte une table rotative, une perceuse avec verrou et un testeur.



(a) Table rotative

(b) Perceuse (c) Testeur

FIG. 1 – Unité de perçage des pièces métalliques

La table rotative (a) sert à transporter les pièces métalliques à la perceuse et au testeur. Elle est de forme circulaire et comporte quatre emplacements, chacun d'entre eux pouvant contenir au plus une pièce. Un emplacement peut être en l'une des quatre positions suivantes : position d'entrée (0), position de perçage (1), position de test (2) et position de sortie (3). Trois capteurs `TT1`, `TT2` et `TT3` attachés à la table indiquent respectivement si une pièce est présente en position 0, si la table vient d'accomplir une rotation de 90° en sens anti-horaire et si une pièce est absente en position 3.

La perceuse (b), située en position 1 de la table, comporte également un verrou permettant de bloquer la pièce pendant l'opération de perçage. Deux capteurs `D1` et `D2` attachés à la perceuse détectent respectivement si elle est en position haute ou basse. Deux capteurs `C1` et `C2` attachés au verrou détectent respectivement s'il est en position relâchée ou bloquée.

Le testeur (c), situé en position 2 de la table, sert à détecter si une pièce a été correctement percée ou non. Il est équipé de deux capteurs `T1` et `T2`, qui détectent respectivement si le testeur est en position haute ou basse. Si le testeur est en position basse, cela signifie soit que la pièce présente en position 2 a été percée correctement, soit qu'il n'y a pas de pièce en cette position.

Chaque dispositif physique (table rotative, verrou, perceuse, testeur) est équipé d'un contrôleur local chargé du pilotage du dispositif. Les contrôleurs locaux reçoivent des signaux de la part des capteurs et renvoient des commandes aux actionneurs attachés aux dispositifs

Portes pour recevoir les signaux des capteurs		
Dispositif	Porte	Fonction
Table	`TT1`	Pièce présente en position 0
	`TT2`	Rotation de 90° accomplie
	`TT3`	Pièce absente en position 3
Verrou	`C1`	Verrou relâché
	`C2`	Verrou bloqué
Perceuse	`D1`	Perceuse en position haute
	`D2`	Perceuse en position basse
Testeur	`T1`	Testeur en position haute
	`T2`	Testeur en position basse

Portes pour envoyer les commandes aux actionneurs		
Dispositif	Porte	Fonction
Table	`TurnOn`	Démarre une rotation de 90°
Verrou	`COnOff`	Bloque ou relâche le verrou
Perceuse	`DOnOff`	Démarre ou arrête le moteur de la perceuse
	`DUpDown`	Démarre le mouvement ascendant ou descendant
Testeur	`TUpDown`	Démarre le mouvement ascendant ou descendant

TAB. 1 – Dialogue des contrôleurs locaux avec les dispositifs physiques

physiques. La table 1 indique les canaux de communication (appelés *portes* en LOTOS) utilisés par les contrôleurs locaux.

La table rotative est contrôlée à travers la porte `TurnOn`, qui commande une rotation de 90° en sens anti-horaire. Ainsi, les pièces sont transportées de la position d'entrée en position de perçage, puis en position de test et enfin en position de sortie. Le verrou, la perceuse et le testeur sont contrôlés à travers des portes modélisant des commandes de *changement de mode*, ainsi appelées puisqu'elles ont deux effets différents, qui alternent à chaque nouvelle invocation. Par exemple, la commande `COnOff` provoque le blocage du verrou s'il est relâché, respectivement le relâchement du verrou s'il est bloqué.

Le fonctionnement de l'unité dans son ensemble est géré par un contrôleur principal, chargé de coordonner l'activité des différents dispositifs et d'interagir avec l'environnement. Le contrôleur principal communique avec les contrôleurs associés aux dispositifs physiques à travers les portes `CMD` (envoi de commandes) et `INF` (réception d'informations) et avec l'environnement à travers la porte `REQ` (envoi de requêtes). La table 2 indique les différents signaux (éléments d'un type énuméré `Sig`) envoyés sur ces portes.

La modélisation suppose que l'environnement réagit correctement aux requêtes du contrôleur principal : en particulier, les pièces sont chargées en position 0 et récupérées en position 3 de la table lors de chaque signal `Add` et `Remove`.

Porte	Signal	Signification
CMD	Turn	Rotation de 90° de la table
	Drill	Perçage de la pièce en position 1
	Lock	Blocage du verrou
	Unlock	Relâchement du verrou
	Test	Test de la pièce en position 2
INF	Turned	Rotation de 90° de la table accomplie
	Present	Pièce présente en position 0
	Drilled	Perçage de la pièce en position 1 accompli
	Locked	Verrou bloqué
	Unlocked	Verrou relâché
	Tested	Pièce testée en position 2
	Absent	Pièce absente en position 3
REQ	Add	Chargement d'une pièce en position 0
	Remove	Récupération de la pièce en position 3

TAB. 2 – Dialogue du contrôleur principal avec les contrôleurs locaux et l'environnement

3.1 Architecture

L'architecture du système modélisé en LOTOS est illustrée dans la figure 2. Les rectangles représentent les différents processus parallèles et les flèches indiquent les portes de communication. Chaque dispositif physique et son contrôleur associé sont représentés par les couples de processus suivants : `TT` et `TTC` (table rotative), `D` et `DC` (perceuse), `C` et `CC` (verrou), `T` et `TC` (testeur). Le contrôleur principal est modélisé par le processus `MC`.

Le processus `TT` communique avec l'environnement à travers les portes `ADD` et `REM`, modélisant respectivement l'insertion d'une pièce en position 0 et la récupération d'une pièce présente en position 3 de la table.

La description en LOTOS de l'architecture est illustrée ci-dessous. Chaque élément est représenté par un appel de processus, paramétré par des portes de communication et éventuellement par des valeurs. L'exécution concurrente des processus est décrite au moyen des opérateurs de composition parallèle "`|||`" et "`|[...]|`" de LOTOS, dénotant l'exécution parallèle asynchrone, respectivement avec synchronisation sur un ensemble de portes. Par exemple, les processus parallèles `TT` et `TTC` se synchronisent sur les portes `TT1`, `TT2`, `TT3` et `TurnOn`, mais s'exécutent de manière asynchrone avec les autres processus `D`, `DC`, etc. Les portes utilisées pour le dialogue entre les dispositifs physiques et leurs contrôleurs locaux sont masquées (rendues invisibles) à l'aide de l'opérateur "**hide**" de LOTOS.

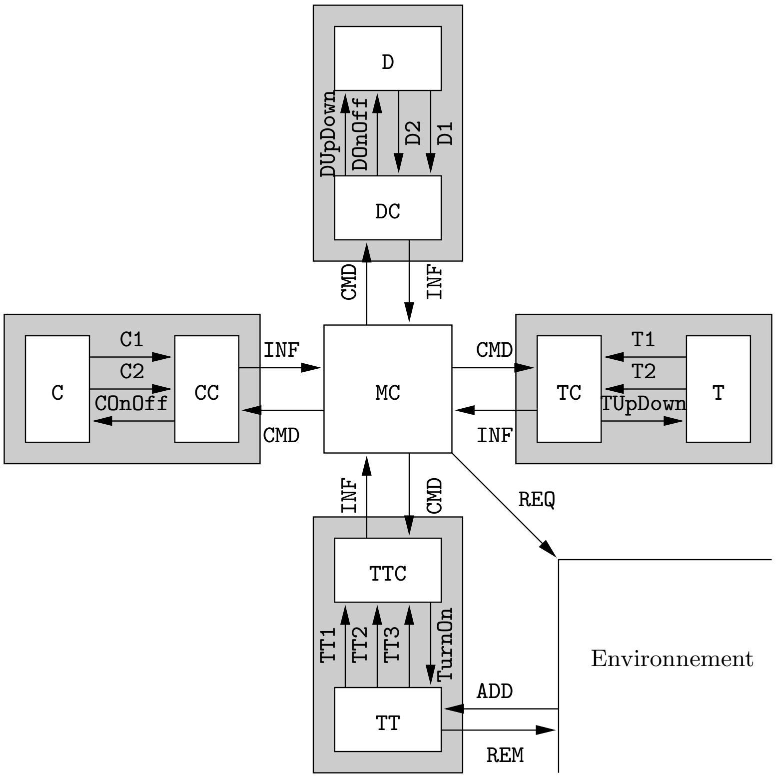


FIG. 2 – Architecture du modèle de l'unité de perçage en LOTOS

```
( ( hide TT1, TT2, TT3, TurnOn, D1, D2, DUpDown, DOnOff,
         C1, C2, COnOff, T1, T2, TUpDown in
      ( (
           TT [TT1, TT2, TT3, TurnOn, ADD, REM]
              (false, false, false, false)
           |[TT1, TT2, TT3, TurnOn]|
           TTC [TT1, TT2, TT3, TurnOn, INF, CMD]
        )
        |||
        (
           D [D1, D2, DUpDown, DOnOff] (false, true)
           |[D1, D2, DUpDown, DOnOff]|
           DC [D1, D2, DUpDown, DOnOff, INF, CMD]
        )
        |||
        (
           C [C1, C2, COnOff] (false)
           |[C1, C2, COnOff]|
           CC [C1, C2, COnOff, INF, CMD]
        )
        |||
```

```
          (
             T [T1, T2, TUpDown] (true)
             |[T1, T2, TUpDown]|
             TC [T1, T2, TUpDown, INF, CMD]
          )
       )
    )
    |[INF, CMD]|
    MC [REQ, INF, CMD] (false, false, false, false, false)
 )
 |[REQ, ADD, REM]|
 Env [REQ, ADD, REM, ERR]
```

Les processus TT, D, C et T, dénotant les dispositifs physiques, sont également munis de paramètres valeurs (dont la signification sera définie dans les sections suivantes) servant à mémoriser leur état courant : au démarrage du système, les valeurs de ces paramètres indiquent le fait que les emplacements de la table rotative sont vides, la perceuse est arrêtée en position haute, le verrou est relâché et le testeur est en position haute. Le processus MC modélisant le contrôleur principal comporte également des paramètres valeurs qui reflètent l'état courant du système.

3.2 Dispositifs physiques et contrôleurs locaux

Nous détaillons ci-dessous la modélisation en Lotos des différents dispositifs physiques et des contrôleurs locaux qui pilotent leur fonctionnement. Par souci de simplicité, les processus correspondant aux contrôleurs locaux sont illustrés de manière graphique dans la figure 3 (les états initiaux sont marqués en gras).

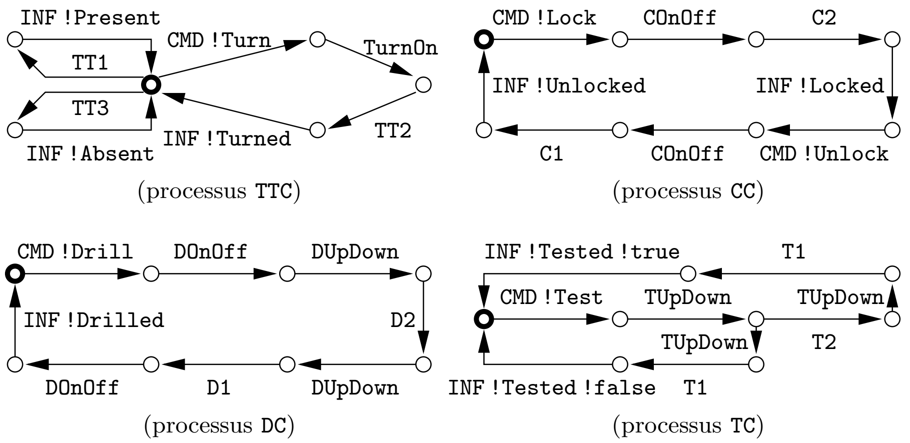


Fig. 3 – Stes modélisant le comportement des contrôleurs locaux

3.2.1 Table rotative

Le processus TT comporte quatre paramètres booléens p0, p1, p2 et p3, qui sont positionnés à true si une pièce se trouve dans la position correspondante de la table rotative et à false sinon. A tout instant, TT peut effectuer un des comportements suivants : il reçoit une commande de rotation de la part de son contrôleur, l'effectue (ce que l'on ne modélise pas explicitement ici[3]), puis renvoie au contrôleur la réponse correspondante ; si la position 0 de la table est libre, il peut recevoir une pièce fournie par l'environnement et signaler à son contrôleur le fait que la position 0 est devenue occupée ; si la position 3 de la table est occupée, il peut délivrer la pièce correspondante à l'environnement. Après avoir effectué un de ces comportements, il continue son exécution de manière cyclique, en mettant à jour ses paramètres.

```
process TT [TT1, TT2, TT3, TurnOn, ADD, REM] (p0, p1, p2, p3:Bool) :
           noexit :=
   TurnOn;
      TT2;
         TT [TT1, TT2, TT3, TurnOn, ADD, REM] (p3, p0, p1, p2)
   []
   [not (p0)] -> ADD;
      TT1;
         TT [TT1, TT2, TT3, TurnOn, ADD, REM] (true, p1, p2, p3)
   []
   [p3] -> REM;
      TT3;
         TT [TT1, TT2, TT3, TurnOn, ADD, REM] (p0, p1, p2, false)
endproc
```

Le processus TTC (voir la figure 3) exécute de manière cyclique la session suivante : lorsque le processus TT l'informe de la présence d'une pièce en position 0 de la table rotative, il transmet cette information au contrôleur principal ; lorsque celui-ci lui envoie une commande de rotation de la table, il transmet cette commande au processus TT, attend la réponse de celui-ci, puis la propage vers le contrôleur principal ; enfin, lorsque le processus TT l'informe de la présence d'une pièce en position 3 de la table, il transmet cette information au contrôleur principal.

3.2.2 Verrou

Le processus C comporte un paramètre booléen locked qui est positionné à true si le verrou est bloqué et à false sinon. C exécute de manière cyclique le comportement suivant : il reçoit une commande de blocage ou de relâchement de la part de son contrôleur, l'exécute (ce que l'on ne modélise pas explicitement ici), puis, suivant son état courant, il renvoie la réponse correspondante au contrôleur.

[3]Dans une modélisation du système prenant en compte le temps quantitatif, les actions TurnOn et TT2 (ainsi que les commandes des actionneurs et les réponses des capteurs attachés aux autres dispositifs physiques) seraient séparées par un délai.

```
process C [C1, C2, COnOff] (locked:Bool) : noexit :=
   COnOff;
   (  [locked] -> C1;
         C [C1, C2, COnOff] (not (locked))
      []
      [not (locked)] -> C2;
         C [C1, C2, COnOff] (not (locked))
   )
endproc
```

Le processus CC (voir la figure 3) exécute de manière cyclique la session suivante : lorsqu'il reçoit une commande de blocage de la part du contrôleur principal, il commande le blocage du verrou, attend la réponse du processus C, puis transmet la réponse correspondante au contrôleur principal ; ensuite, il attend une commande de relâchement de la part du contrôleur principal, il commande le relâchement du verrou, attend la réponse du processus C, puis la propage vers le contrôleur principal.

3.2.3 Perceuse

Le processus D comporte deux paramètres booléens on et up, qui sont respectivement positionnés à true si la perceuse est démarrée et en position haute et à false sinon. A tout instant, D peut effectuer un des comportements suivants : il reçoit de la part de son contrôleur une commande de démarrage ou d'arrêt ; si la perceuse est démarrée, il reçoit une commande de mouvement ascendant ou descendant, l'effectue (ce que l'on ne modélise pas explicitement ici), puis renvoie la réponse correspondante à son contrôleur. Après avoir effectué un de ces comportements, il continue son exécution de manière cyclique, en mettant à jour ses paramètres.

```
process D [D1, D2, DUpDown, DOnOff] (on, up:Bool) : noexit :=
   DOnOff;
      D [D1, D2, DUpDown, DOnOff] (not (on), up)
   []
   [on] -> (
      DUpDown;
      (  [up] -> D2;
            D [D1, D2, DUpDown, DOnOff] (on, not (up))
         []
         [not (up)] -> D1;
            D [D1, D2, DUpDown, DOnOff] (on, not (up))
      )
   )
endproc
```

Le processus DC (voir la figure 3) exécute de manière cyclique la session suivante : lorsqu'il reçoit une commande de perçage de la part du contrôleur principal, il commande le démarrage du moteur, puis la descente de la perceuse et il attend la réponse du processus

D ; ensuite, il commande le relèvement de la perceuse, attend la réponse du processus D, puis commande l'arrêt du moteur ; enfin, il renvoie au contrôleur principal la réponse signifiant l'accomplissement de l'opération de perçage.

3.2.4 Testeur

Le processus T comporte un paramètre booléen up qui est positionné à true si le testeur est en position haute et à false sinon. T exécute le comportement cyclique suivant : il reçoit une commande de mouvement ascendant ou descendant de la part de son contrôleur ; il effectue la commande (ce que l'on ne modélise pas explicitement ici), puis, suivant son état courant, renvoie une réponse correspondante à son contrôleur. Le fait que la pièce située en position 2 est correctement percée ou pas est modélisé comme un choix non déterministe, consistant à renvoyer la réponse au contrôleur (pièce correctement percée) ou à ne pas renvoyer de réponse (pièce incorrectement percée).

```
process T [T1, T2, TUpDown] (up:Bool) : noexit :=
   TUpDown;
   (  [up] -> (
         T2;                             (* Pie‘ce bien perce’e *)
            T [T1, T2, TUpDown] (not (up))
         []
         T [T1, T2, TUpDown] (not (up))     (* Pie‘ce mal perce’e *)
      )
      []
      [not (up)] -> T1;
         T [T1, T2, TUpDown] (not (up))
   )
endproc
```

Le processus TC (voir la figure 3) exécute de manière cyclique la session suivante : lorsqu'il reçoit une commande de test de la part du contrôleur principal, il commande le mouvement descendant du testeur, puis attend la réponse du processus T (une absence de réponse indique une pièce incorrectement percée) ; ensuite, il renvoie une commande de mouvement ascendant du testeur, puis attend la réponse du processus T ; finalement, il renvoie une réponse correspondante au contrôleur principal, indiquant au moyen d'une valeur booléenne true ou false le fait que la pièce située en position 2 de la table a été correctement percée ou pas.

3.3 Contrôleur principal — version séquentielle

Nous décrivons ci-dessous une première version du contrôleur principal, dans laquelle les différentes phases de dialogue avec les contrôleurs locaux sont enchaînées séquentiellement. Le processus MC comporte cinq paramètres booléens : p0, p1, p2 et p3 indiquent la présence de pièces dans les positions correspondantes de la table rotative et tr est positionné à true si une pièce correctement percée est située en position 3 et à false sinon. MC exécute de manière cyclique la session suivante, comportant cinq phases de contrôle effectuées une après l'autre :

Phase 1. Si la position 0 de la table est libre, il envoie à l'environnement une requête de chargement d'une pièce (l'environnement est supposé réagir immédiatement à la requête en chargeant la pièce), puis attend la réponse correspondante de la part du contrôleur `TTC`.

Phase 2. Si une pièce est présente en position 1, il envoie une commande de blocage du verrou au contrôleur `CC` et attend sa réponse, puis envoie une commande de perçage au contrôleur `DC` et attend sa réponse, et finalement envoie une commande de relâchement du verrou et attend la réponse du contrôleur `CC`.

Phase 3. Si une pièce est présente en position 2, il envoie une commande de test au contrôleur `TC`, puis attend la réponse correspondante.

Phase 4. Si une pièce est présente en position 3, il envoie à l'environnement une requête de récupération de la pièce (l'environnement est supposé récupérer immédiatement la pièce), puis attend la réponse correspondante de la part du contrôleur `TTC`.

Phase 5. Il envoie une commande de rotation de la table au contrôleur `TTC`, puis attend la réponse de celui-ci avant de recommencer son exécution cyclique en mettant à jour ses paramètres.

```
process MC [REQ, INF, CMD] (p0, p1, p2, p3, tr:Bool) : noexit :=
   (  [not (p0)] -> REQ !Add; INF !Present;
         exit (true)
      []
      [p0] -> exit (p0)
   )
   >>
   accept new_p0:Bool in
   (  (  [p1] -> CMD !Lock; INF !Locked;
               CMD !Drill; INF !Drilled;
                  CMD !Unlock; INF !Unlocked;
                     exit
         []
         [not (p1)] -> exit
      )
      >>
      (  [p2] -> CMD !Test; INF !Tested ?r:Bool;
            exit (r)
         []
         [not (p2)] -> exit (tr)
      )
      >>
      accept new_tr:Bool in
      (  (  [p3] -> REQ !Remove !tr; INF !Absent;
                  exit (false)
            []
            [not (p3)] -> exit (p3)
         )
```

```
          >>
          accept new_p3:Bool in
             CMD !Turn; INF !Turned;
                MC [REQ, INF, CMD] (new_p3, new_p0, p1, p2, new_tr)
       )
    )
 endproc
```

L'enchaînement des cinq phases de contrôle est modélisé au moyen des opérateurs LOTOS dévolus à la composition séquentielle des comportements : "**exit**" et "**exit** (V_1, …, V_n)" dénotent la terminaison d'un comportement sans, respectivement avec le retour des valeurs résultat V_1, …, V_n ; "B_1 `>>` B_2" exprime l'exécution du comportement B_2 une fois que l'exécution de B_1 s'est terminée par un "**exit**" ; et "B_1 `>>` **accept** x_1:T_1, …, x_n:T_n **in** B_2" dénote l'exécution de B_2 après que B_1 ait terminé son exécution par un "**exit** (V_1, …, V_n)", dont les valeurs résultat sont affectées respectivement aux variables x_1, …, x_n, utilisées par la suite dans B_2.

3.4 Contrôleur principal — version parallèle

Nous décrivons ci-dessous une deuxième version du contrôleur principal, dans laquelle les phases de dialogue avec les contrôleurs locaux sont effectuées en parallèle. Le comportement de cette version est plus complexe, mais aussi plus efficace que celui de la version séquentielle, le traitement complet d'une pièce étant plus rapide (ceci est confirmé par une évaluation du débit de la table rotative[4], suivant l'approche proposée en [GH02]). Le nouveau processus `MC` est obtenu en composant les quatre premières phases de traitement (associées aux emplacements de la table rotative), décrites à la section 3.3, en utilisant l'opérateur parallèle asynchrone "`|||`" de LOTOS.

```
 process MC [REQ, INF, CMD] (p0, p1, p2, p3, tr:Bool) : noexit :=
    (  (  [not (p0)] -> REQ !Add; INF !Present;
             exit (true, p1, p2, any Bool, any Bool)
          []
          [p0] -> exit (p0, p1, p2, any Bool, any Bool)
       )
       |||
       (  [p1] -> CMD !Lock; INF !Locked;
             CMD !Drill; INF !Drilled;
                CMD !Unlock; INF !Unlocked;
                   exit (any Bool, p1, p2, any Bool, any Bool)
          []
          [not (p1)] -> exit (any Bool, p1, p2, any Bool, any Bool)
       )
       |||
       (  [p2] ->  CMD !Test; INF !Tested ?r:Bool;
```

[4]Une modélisation en LOTOS de la table rotative étendue avec des informations stochastiques permettant le calcul du débit peut être trouvée dans l'exemple `demo_39` de la distribution CADP.

```
            exit (any Bool, p1, p2, any Bool, r)
          []
          [not (p2)] -> exit (any Bool, p1, p2, any Bool, tr)
        )
        |||
        (  [p3] -> REQ !Remove !tr; INF !Absent;
            exit (any Bool, p1, p2, false, any Bool)
          []
          [not (p3)] -> exit (any Bool, p1, p2, p3, any Bool)
        )
    )
    >> accept new_p0, new_p1, new_p2, new_p3, new_tr:Bool in
        CMD !Turn; INF !Turned;
          MC [REQ, INF, CMD] (new_p3, new_p0, new_p1, new_p2, new_tr)
endproc
```

L'exécution de chaque phase produit des changements dans l'état de la table rotative, mémorisé par le contrôleur au moyen des paramètres booléens `p0`, `p1`, `p2`, `p3` et `tr`. Ces changements sont modélisés grâce à l'opérateur "**exit**", qui permet d'indiquer, pour chaque paramètre, soit sa nouvelle valeur obtenue après l'exécution de la phase de traitement correspondante, soit le fait que le paramètre respectif est modifié par une autre phase (filtre "**any**"). Les occurrences de l'opérateur "**exit**" à la fin des quatre phases parallèles doivent être compatibles, c'est à dire que chaque paramètre doit avoir la même valeur ou être filtré par "**any**" dans chacun des quatre "**exit**" (par exemple, le paramètre `p0` est positionné à `true` ou laissé inchangé par la première phase et filtré avec "**any**" par les autres phases). Ceci est nécessaire afin d'assurer la synchronisation correcte des phases lors de leur terminaison (*join*), imposée par la sémantique de l'opérateur "`|||`". Une fois l'ensemble des quatre phases parallèles terminé, le contrôleur principal commande la rotation de la table, attend la réponse correspondante du contrôleur local, puis recommence son exécution cyclique en mettant à jour ses paramètres (récupérés au moyen de l'opérateur "**>> accept** ... **in**").

3.5 Environnement

Le dernier élément à modéliser afin d'obtenir une description complète du système est l'environnement, qui traite les requêtes du contrôleur principal en chargeant et en récupérant des pièces en position 0, respectivement en position 3 de la table rotative.

```
process Env [REQ, ADD, REM, ERR] : noexit :=
    REQ !Add;
       ADD;
          Env [REQ, ADD, REM, ERR]
    []
    REQ !Remove ?r:Bool;
    (  [r] -> REM;
          Env [REQ, ADD, REM, ERR]
       []
```

```
        [not (r)] -> ERR; REM;
           Env [REQ, ADD, REM, ERR]
     )
endproc
```

Lors de la récupération d'une pièce, le contrôleur principal indique également à l'environnement si la pièce a été correctement percée ou pas, ce dernier cas étant signalé par l'environnement à travers la porte ERR.

4 Analyse du fonctionnement de l'unité de perçage

Une fois l'unité de perçage modélisée en Lotos, nous pouvons analyser son comportement avec des outils de vérification fournis par Cadp. Nous étudions d'abord la cohérence de la version du système équipée du contrôleur principal séquentiel par rapport à celle équipée du contrôleur parallèle. Ensuite, nous identifions un ensemble de propriétés de bon fonctionnement, les exprimons en logique temporelle et les vérifions sur les deux versions du système. Les vérifications ont été effectuées sur un Pc équipé d'un processeur Pentium 4 à 2,2 GHz et d'une de mémoire de 1 Go.

4.1 Vérification par équivalences

Nous commençons par construire, à l'aide des compilateurs Cæsar et Cæsar.Adt, le modèle Ste M_{seq} de l'unité de perçage équipée du contrôleur principal séquentiel, afin d'estimer sa taille et de procéder éventuellement à son inspection visuelle. Le Ste correspondant, minimisé modulo la bisimulation de branchement grâce à l'outil Bcg_Min et mis en forme graphique au moyen de l'outil Bcg_Edit, est illustré en figure 4. Il comporte 69 états, 72 transitions et un facteur de branchement moyen $1,04$ (nombre de transitions sortant d'un état), ce qui indique le caractère séquentiel de cette version du système.

A partir de l'état initial (numéroté 0), nous observons une séquence d'actions modélisant le chargement de pièces sur la table rotative (initialement vide) jusqu'à ce que le régime permanent de fonctionnement est atteint (les emplacements 0, 1 et 2 de la table sont occupés). Cette séquence est suivie de deux branches de traitement différentes, correspondant respectivement au fait que la pièce présente en position 2 de la table a été correctement percée (branche de gauche) ou non (branche de droite). Les deux branches ont un comportement semblable, excepté le fait qu'une action ERR, indiquant la récupération par l'environnement d'une pièce incorrectement percée, est issue sur la branche de droite.

Ensuite, nous construisons le modèle Ste M_{par} de l'unité de perçage équipée du contrôleur principal parallèle. Celui-ci s'avère beaucoup plus grand que le Ste correspondant au contrôleur séquentiel : il comporte 24 346 états et 85 013 transitions[5], le facteur de branchement moyen étant $3,49$. Cette augmentation en taille est causée par l'entrelacement des quatre phases de traitement (correspondant aux positions des pièces sur la table rotative), qui auparavant étaient enchaînées séquentiellement. La taille du nouveau Ste rendant son

[5] Une réduction du Ste M_{par} par τ-confluence [PLM03] au moyen de l'outil Reductor de Cadp réduirait sa taille à 5 373 états et 17 711 transitions.

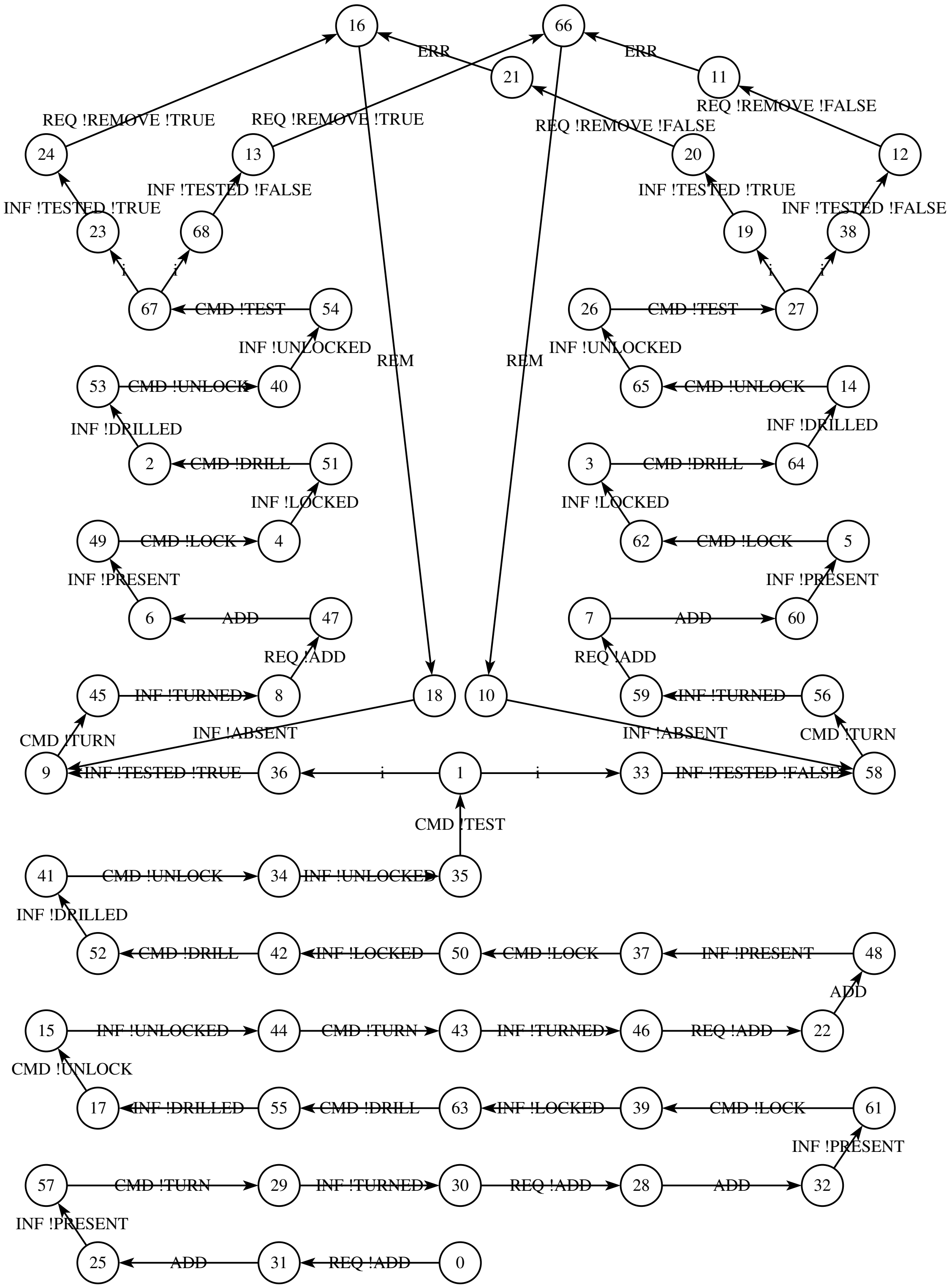


Fig. 4 – Ste de l'unité de perçage équipée du contrôleur séquentiel

inspection visuelle impraticable, l'emploi des outils de vérification devient indispensable afin de s'assurer du bon fonctionnement du système.

Une première vérification consiste à s'assurer que les comportements des deux versions du système, modélisés respectivement par M_{seq} et M_{par}, sont cohérents. Intuitivement, puisque l'enchaînement séquentiel des phases de traitement est un cas particulier de leur exécution parallèle, les comportements du système équipé du contrôleur séquentiel devraient être "simulés" par le système équipé du contrôleur parallèle. Ceci est effectivement le cas, puisqu'au moyen de Bisimulator nous pouvons vérifier que M_{seq} est inclus dans M_{par} modulo le préordre de la bisimulation de branchement. Ceci signifie que — abstraction faite des actions modélisant le dialogue avec les capteurs et les actionneurs (voir la section 3.1) — les arbres d'exécution contenus dans M_{seq} sont également contenus dans M_{par}. Cette vérification peut être effectuée en parcourant M_{par} à la volée ; le Seb sous-jacent exploré par Bisimulator comporte 515 variables booléennes et 934 opérateurs. La vérification a nécessité 1 seconde de temps de calcul et 9,6 Mo de mémoire.

En revanche, les deux Stes M_{seq} et M_{par} ne sont équivalents modulo aucune des sept relations d'équivalence implémentées dans Bisimulator : en effet, la séquence d'exécution suivante (restreinte aux seules actions visibles) est présente dans M_{par} mais pas dans M_{seq} :

$$s_0 \xrightarrow{\texttt{REQ !ADD}} s_1 \xrightarrow{\texttt{ADD}} s_2 \xrightarrow{\texttt{INF !PRESENT}} s_3 \xrightarrow{\texttt{CMD !TURN}} s_4 \xrightarrow{\texttt{INF !TURNED}} s_5 \xrightarrow{\texttt{CMD !LOCK}} s_6$$

Cette séquence, obtenue automatiquement comme contre-exemple en essayant de vérifier l'équivalence de M_{seq} et M_{par} modulo l'équivalence de trace faible, montre le fait qu'après l'insertion de la première pièce sur la table et la première rotation de celle-ci, le contrôleur séquentiel n'entame pas la phase de perçage de la pièce (actuellement située en position 1) en commandant le blocage du verrou, mais recommence son cycle de fonctionnement en traitant l'insertion d'une nouvelle pièce en position 0 (devenue libre). Par contre, le contrôleur parallèle peut commander le perçage avant l'insertion, puisqu'il permet l'exécution concurrente des quatre phases de traitement.

4.2 Vérification par logiques temporelles

La vérification par équivalences a fourni une certaine indication sur la cohérence des deux versions du système ; cependant, elle ne garantit pas leur bon fonctionnement. Dans cette section, nous identifions plusieurs propriétés temporelles caractérisant l'enchaînement correct des actions exécutées par le système dans le temps et nous les exprimons en μ-calcul régulier d'alternance 1 [MS03], la logique temporelle acceptée en entrée par l'outil Evaluator 3.5. Nous considérons deux classes de propriétés classiques (illustrées de manière graphique en figure 5) :

Les propriétés de sûreté expriment intuitivement que "rien de mal n'arrivera" pendant l'exécution du système. Elles peuvent être exprimées en μ-calcul régulier au moyen de la formule "$[R]$ false", où R est une expression régulière (définie sur le vocabulaire des prédicats sur les actions du Ste) caractérisant les séquences d'actions indésirables qui violent les propriétés de sûreté. La modalité de nécessité ci-dessus stipule que toutes les séquences d'actions issues de l'état courant et qui satisfont R mènent nécessairement à des états satisfaisant false ; comme de tels états n'existent pas, les séquences correspondantes n'existent pas non plus.

Les propriétés de vivacité expriment intuitivement que "quelque chose de bien arrivera" pendant l'exécution du système. La plupart des propriétés de vivacité que nous utiliserons ici s'expriment en μ-calcul régulier par (des variantes de) la formule "$[R]$ inev(A, B, P)", où R dénote une expression régulière sur actions, A, B dénotent des prédicats sur actions, et P dénote une propriété sur états. La formule ci-dessus stipule que toutes les séquences d'actions issues de l'état courant et qui satisfont R mènent nécessairement à des états à partir desquels toutes les séquences consistent en zéro ou plusieurs actions satisfaisant A, suivies d'une action satisfaisant B et menant à un état satisfaisant P. Autrement dit, après toute séquence satisfaisant R, il est *inévitable* d'arriver (après zéro ou plusieurs actions satisfaisant A, suivies d'une action satisfaisant B) à un état satisfaisant P.

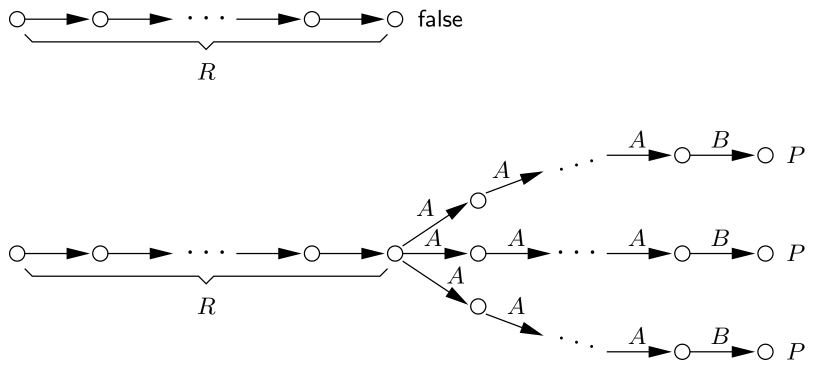


FIG. 5 – Illustration des opérateurs "$[R]$ false" et "$[R]$ inev(A, B, P)"

La table 3 montre sept propriétés de sûreté de l'unité de perçage, ainsi que leurs définitions en μ-calcul régulier d'alternance 1. EVALUATOR 3.5 permet l'expression de prédicats de base sur actions sous la forme de chaînes de caractères entourées de " (dénotant une seule action) ou d'expressions régulières sur chaînes de caractères entourées de ' (dénotant un ensemble d'actions). Les propriétés P_1–P_5 caractérisent l'ordre des traitements d'une pièce par l'unité de perçage, imposé par le sens de rotation de la table (insertion, verrouillage, perçage, déverrouillage, test et récupération). La propriété P_6 exprime la sûreté de l'unité de perçage vis-à-vis du test des pièces. Enfin, la propriété P_7 exprime une contrainte sur l'ordre d'exécution du perçage et du test lorsque les emplacements correspondants de la table sont occupés.

Grâce à EVALUATOR 3.5, nous pouvons vérifier que toutes les propriétés P_1–P_6 sont satisfaites par les STEs M_{seq} et M_{par} correspondant aux deux versions de l'unité de perçage. La taille des SEBs sous-jacents explorés par EVALUATOR 3.5 varie entre 321 variables et 336 opérateurs (pour la propriété P_2 sur M_{seq}) et 48 712 variables et 171 645 opérateurs (pour la propriété P_4 sur M_{par}). Par contre, la propriété P_7 est satisfaite par M_{seq}, mais pas par M_{par} ; EVALUATOR 3.5 exécuté avec la stratégie de parcours en largeur fournit une séquence minimale de contre-exemple comportant 23 actions visibles, illustrées ci-dessous :

$$s_0 \xrightarrow{\texttt{REQ !ADD}} s_1 \xrightarrow{\texttt{ADD}} s_2 \xrightarrow{\texttt{INF !PRESENT}} s_3 \xrightarrow{\texttt{CMD !TURN}} s_4 \xrightarrow{\texttt{INF !TURNED}} s_5 \xrightarrow{\texttt{CMD !LOCK}}$$

$$s_6 \xrightarrow{\texttt{INF !LOCKED}} s_7 \xrightarrow{\texttt{CMD !DRILL}} s_8 \xrightarrow{\texttt{INF !DRILLED}} s_9 \xrightarrow{\texttt{CMD !UNLOCK}} s_{10} \xrightarrow{\texttt{INF !UNLOCKED}} s_{11} \xrightarrow{\texttt{REQ !ADD}}$$

$$s_{12} \xrightarrow{\texttt{ADD}} s_{13} \xrightarrow{\texttt{INF !PRESENT}} s_{14} \xrightarrow{\texttt{CMD !TURN}} s_{15} \xrightarrow{\texttt{INF !TURNED}} s_{16} \xrightarrow{\texttt{CMD !LOCK}} s_{17} \xrightarrow{\texttt{INF !LOCKED}}$$

$$s_{18} \xrightarrow{\texttt{REQ !ADD}} s_{19} \xrightarrow{\texttt{ADD}} s_{20} \xrightarrow{\texttt{INF !PRESENT}} s_{21} \xrightarrow{\texttt{CMD !TEST}} s_{22} \xrightarrow{\texttt{CMD !DRILL}} s_{23}$$

Cette séquence, absente de M_{seq} mais présente dans M_{par}, contient trois insertions de pièces et deux rotations de la table (ce qui assure l'occupation des positions de perçage et de test), une commande de blocage du verrou (en préparation du perçage) et une commande de test suivie d'une commande de perçage. En effet, le contrôleur séquentiel traite les pièces présentes sur la table dans un ordre bien précis — le test de la pièce en position 2 étant effectué après le perçage de la pièce présente en position 1 — tandis que celui parallèle est capable d'exécuter les différents traitements de manière simultanée.

Bien que les propriétés de sûreté décrites ci-dessus empêchent les mauvais fonctionnements de l'unité de perçage, elles n'assurent pas l'accomplissement des divers traitements sur les pièces : en effet, une unité de perçage dont la table rotative reste immobile satisfait toutes ces propriétés. Afin d'assurer le démarrage et la progression des phases de traitement, le système doit également satisfaire des propriétés de vivacité. La table 4 montre sept propriétés de vivacité de l'unité de perçage et les formules temporelles correspondantes. La propriété P_8 décrit la séquence de démarrage du système, pendant laquelle des pièces sont insérées sur tous les emplacements de la table rotative, qui atteint le régime normal de fonctionnement. Les propriétés P_9–P_{12} sont les contreparties des propriétés P_1–P_4 : elles caractérisent la progression de chaque traitement et la rotation de la table. La propriété P_{13} précise les réponses aux commandes et requêtes du contrôleur principal, qui sont renvoyées par le système ou par l'environnement lors de chaque cycle de traitement. Enfin, la propriété P_{14} spécifie la réaction correcte du système lorsque le perçage de certaines pièces a échoué.

Grâce à Evaluator 3.5, nous pouvons vérifier que toutes les propriétés P_8–P_{14} sont satisfaites par les Stes M_{seq} et M_{par} correspondant aux deux versions de l'unité de perçage. La taille des Sebs sous-jacents varie entre 650 variables et 947 opérateurs (pour la propriété P_8 sur M_{seq}) et 275 277 variables et 606 747 opérateurs (pour la propriété P_{13} sur M_{par}). La vérification des 14 propriétés a été effectuée en 44,5 secondes, avec un pic de consommation mémoire de 21,5 Mo pour la propriété P_{13} sur M_{par}.

No.	Formule	Description
P_1	[true* . "INF !PRESENT" . (not "INF !TURNED")* . "INF !TURNED" . (not "INF !LOCKED")* . "CMD !DRILL"] false	Après l'insertion d'une pièce et une rotation de la table, le contrôleur principal ne peut pas commander un perçage avant que le verrou n'ait été bloqué.
P_2	[true* . "INF !DRILLED" . (not 'INF !UNLOCKED.*')* . "CMD !TURN"] false	Après le perçage d'une pièce, le contrôleur principal ne peut pas commander une rotation avant que le verrou n'ait été relâché.
P_3	[true* . "INF !UNLOCKED" . (not "INF !TURNED")* . "INF !TURNED" . (not 'INF !TESTED.*')* . "CMD !TURN"] false	Après le relâchement du verrou sur une pièce et une rotation de la table, le contrôleur principal ne peut pas commander une rotation avant que la pièce n'ait été testée.
P_4	[true* . 'INF !TESTED.*' . (not "INF !TURNED")* . "INF !TURNED" . (not "INF !ABSENT")* . "CMD !TURN"] false	Après le test d'une pièce et une rotation de la table, le contrôleur principal ne peut pas commander une rotation avant que la pièce n'ait été récupérée.
P_5	[true* . "INF !ABSENT" . (not "INF !TURNED")* . "INF !TURNED'' . (not "INF !PRESENT")* . "CMD !TURN"] false	Après la récupération d'une pièce et une rotation de la table, le contrôleur principal ne peut pas commander une rotation avant qu'une nouvelle pièce n'ait été insérée.
P_6	[true* . "INF !TESTED !TRUE" . (not "INF !TURNED")* . "INF !TURNED" . (not "INF !TURNED")* . "ERR"] false	Chaque fois que le testeur détecte une pièce correctement percée, aucune erreur ne sera signalée lors du cycle de traitement suivant.
P_7	[true* . "INF !PRESENT" . true* . "INF !PRESENT" . true* . "INF !PRESENT" . true* . "CMD !TEST" . (not "INF !TURNED")* . "CMD !DRILL"] false	Après que les positions de perçage et de test de la table ont été occupées, le contrôleur principal ne peut pas commander un test avant de commander un perçage.

TAB. 3 – Propriétés de sûreté de l'unité de perçage

No.	Formule	Description
P_8	inev (not "CMD !TURN", "REQ !ADD", inev (not "CMD !TURN", "CMD !TURN", inev (not "CMD !TURN", "REQ !ADD", inev (not "CMD !TURN", "CMD !TURN", inev (not "CMD !TURN", "REQ !ADD", inev (not "CMD !TURN", "CMD !TURN", inev (not "CMD !TURN", "REQ !ADD", true)))))))	Initialement, le contrôleur principal commande inévitablement l'insertion de pièces dans toutes les positions de la table rotative.
P_9	[true* . "INF !PRESENT"] inev (not "CMD !TURN", "CMD !TURN", inev (not "CMD !TURN", "CMD !LOCK", inev (not "CMD !TURN", "CMD !DRILL", inev (not "CMD !TURN", "CMD !UNLOCK", true))))	Chaque pièce insérée sera percée après la prochaine rotation de la table.
P_{10}	[true* . "INF !UNLOCKED"] inev (not "CMD !TURN", "CMD !TURN", inev (not "CMD !TURN", "CMD !TEST", true))	Chaque pièce percée sera testée après la prochaine rotation de la table.
P_{11}	[true* . 'INF !TESTED.*'] inev (not "CMD !TURN", "CMD !TURN", inev (not "CMD !TURN", 'REQ !REMOVE.*', true))	Chaque pièce testée sera récupérée après la prochaine rotation de la table.
P_{12}	[true* . "INF !ABSENT"] inev (not "CMD !TURN", "CMD !TURN", inev (not "CMD !TURN", "REQ !ADD", true))	Chaque récupération de pièce sera suivie par l'insertion d'une nouvelle pièce après la prochaine rotation de la table.
P_{13}	[true*] (["REQ !ADD"] inev (not "INF !TURNED", "INF !PRESENT", true) and ["CMD !LOCK"] inev (not "INF !TURNED", "INF !LOCKED", true) and ["CMD !DRILL"] inev (not "INF !TURNED", "INF !DRILLED", true) and ["CMD !UNLOCK"] inev (not "INF !TURNED", "INF !UNLOCKED", true) and ["CMD !TEST"] inev (not "INF !TURNED", 'INF !TESTED.*', true) and ['REQ !REMOVE.*'] inev (not "INF !TURNED", "INF !ABSENT", true) and ["CMD !TURN"] inev (not "INF !TURNED", "INF !TURNED", true))	Chaque commande (respectivement requête) envoyée par le contrôleur principal aux dispositifs physiques (respectivement à l'environnement) sera inévitablement suivie de son acquittement avant la prochaine rotation de la table.
P_{14}	[true* . "INF !TESTED !FALSE"] inev (not "CMD !TURN", "CMD !TURN", inev (not "CMD !TURN", "ERR", true))	Chaque fois qu'une pièce incorrectement percée est détectée, une erreur sera inévitablement signalée lors du cycle de traitement suivant.

TAB. 4 – Propriétés de vivacité de l'unité de perçage

5 Conclusion et perspectives

A travers l'exemple détaillé dans ce rapport, nous avons tenté d'illustrer l'utilité des méthodes de modélisation et vérification formelle fournies par la boîte à outils CADP pour analyser les propriétés fonctionnelles des systèmes industriels critiques comportant du parallélisme. Le langage LOTOS s'avère adapté pour décrire de manière succincte et abstraite le fonctionnement des contrôleurs chargés de piloter les dispositifs physiques. A l'heure actuelle, CADP a été utilisée pour traiter 86 études de cas industrielles[6] et les divers composants génériques des environnements BCG et OPEN/CÆSAR ont permis le développement de 24 outils dérivés[7]. Les retours d'expérience respectifs ont conduit au développement de nouveaux outils, ainsi qu'à l'amélioration des outils existants en termes de performance et d'ergonomie.

Nous avons présenté ici uniquement quelques outils basiques de CADP, qui mettent en œuvre les méthodes d'analyse classiques (vérification par équivalences et par logiques temporelles) sur STEs. CADP offre également des fonctionnalités d'analyse évoluées, permettant de traiter des systèmes de grande taille : vérification compositionnelle au moyen de l'outil EXP.OPEN 2.0 [Lan05], vérification distribuée au moyen des outils DISTRIBUTOR et BCG_MERGE [GMS01, GMB+06] et des algorithmes de résolution distribuée de SEBs [JM04, JM05, JM06], réduction par ordres partiels [PLM03, Mat05]. Ces fonctionnalités sont orthogonales et opèrent à la volée en utilisant la représentation implicite des STEs définie par OPEN/CÆSAR ; par conséquent, elles peuvent être combinées librement, suivant la nature et la taille du système à vérifier, afin de cumuler leurs bénéfices. En outre, CADP dispose du langage SVL [GL01] et du compilateur associé, qui autorisent une description succincte et abstraite de scénarios d'analyse complexes, comportant des centaines d'invocations des outils de vérification.

Les travaux de recherche et développement autour de CADP sont poursuivis dans plusieurs directions. La montée en puissance actuelle des grappes et des grilles de machines rend nécessaire la conception d'algorithmes de vérification distribuée spécifiques, ainsi que la définition de représentations des STEs adaptées à la répartition [GMS01, GMB+06]. CADP peut également jouer le rôle de moteur d'analyse pour d'autres langages, notamment ceux dédiés à la description d'architectures matérielles asynchrones [SS05]. Finalement, l'application des techniques de vérification de CADP dans d'autres domaines situés actuellement en plein essor, tels que la bio-informatique, s'avère particulièrement prometteuse [BBdJ+04, BRdJ+05].

Références

[And94] Henrik R. Andersen. Model Checking and Boolean Graphs. *Theoretical Computer Science*, 126(1) :3–30, April 1994.

[BBdJ+04] Grégory Batt, Damien Bergamini, Hidde de Jong, Hubert Garavel, and Radu Mateescu. Model Checking Genetic Regulatory Networks using GNA and CADP. In Susanne Graf and Laurent Mounier, editors, *Proc. of the 11th Int.*

[6] Voir le catalogue en ligne `http ://www.inrialpes.fr/vasy/cadp/case-studies`
[7] Voir le catalogue en ligne `http ://www.inrialpes.fr/vasy/cadp/software`

SPIN Workshop on Model Checking of Software SPIN'2004 (Barcelona, Spain), volume 2989 of *Lecture Notes in Computer Science*, pages 156–161. Springer Verlag, April 2004.

[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, *Proc. of the 11th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2005 (Edinburgh, Scotland, UK)*, volume 3440 of *Lecture Notes in Computer Science*, pages 581–585. Springer Verlag, April 2005.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Proc. of the 4th Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems SFM-RT'04 (Bertinoro, Italy)*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Verlag, September 2004.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3) :560–599, July 1984.

[BK02] V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems.* Thèse de Doctorat, Technical University of Eindhoven, March 2002.

[BRdJ$^+$05] Grégory Batt, Delphine Ropers, Hidde de Jong, Johannes Geiselmann, Radu Mateescu, Michel Page, and Dominique Schneider. Validation of Qualitative Models of Genetic Regulatory Networks by Model Checking : Analysis of the Nutritional Stress Response in Escherichia Coli. *Bioinformatics*, 21(Suppl 1) :i19–i28, 2005.

[BTW$^+$05] Elena Bortnik, Nikola Trcka, Anton J. Wijs, S. P. Luttik, Joanna M. van de Mortel-Fronczak, Jos C. M. Baeten, Willem J. Fokkink, and J. E. Rooda. Analyzing a χ Model of a Turntable System using Spin, CADP and UPPAAL. *Journal of Logic and Algebraic Programming*, 65(2) :51–104, novembre–décembre 2005.

[BV06] Bernard Berthomieu and François Vernadat. *Réseaux de Petri temporels : méthodes d'analyse et vérification avec TINA*. In Nicolas Navet, editor, *Systèmes temps reél 1 — techniques de description et de vérification*, chapter 1, pages 25–58. Traité IC2. Lavoisier, 2006.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In Barry Boehm, David Garlan, and Jeff Kramer, editors, *Proc. of the 21st Int. Conference on Software Engineering ICSE'99 (Los Angeles, CA, USA)*, pages 411–420. ACM, May 1999.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1 — Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, 1985.

[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.

[Gar89] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proc. of the 2nd Int. Conference on Formal Description Tech-*

niques FORTE'89 (Vancouver B.C., Canada), pages 147–162. North Holland, December 1989.

[Gar98] Hubert Garavel. OPEN/CÆSAR : An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proc. of the First Int. 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. Springer Verlag, March 1998.

[GH02] Hubert Garavel and Holger Hermanns. On Combining Functional Verification and Performance Evaluation using CADP. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *Proc. of the 11th Int. Symposium of Formal Methods Europe FME'2002 (Copenhagen, Denmark)*, volume 2391 of *Lecture Notes in Computer Science*, pages 410–429. Springer Verlag, July 2002.

[GL01] Hubert Garavel and Frédéric Lang. SVL : a Scripting Language for Compositional Verification. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Proc. of the 21st IFIP WG 6.1 Int. Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea)*, pages 377–392. IFIP, Kluwer Academic Publishers, August 2001.

[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. également disponible comme Rapport Technique INRIA RT-0254.

[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 Palberg, editors, *Proc. of the 12th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2006 (Vienna, Austria)*, volume 3920 of *Lecture Notes in Computer Science*, pages 445–449. Springer Verlag, March 2006.

[GMS01] Hubert Garavel, Radu Mateescu, and Irina Smarandache. Parallel State Space Construction for Model-Checking. In Matthew B. Dwyer, editor, *Proc. of the 8th Int. SPIN Workshop on Model Checking of Software SPIN'2001 (Toronto, Canada)*, volume 2057 of *Lecture Notes in Computer Science*, pages 217–234. Springer Verlag, May 2001.

[Gro97] Jan Friso Groote. The Syntax and Semantics of Timed muCRL. Technical Report SEN-R9709, CWI, Amsterdam, 1997.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proc. of the 10th Int. Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394. IFIP, North Holland, June 1990.

[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.

[Hol03] Gerard Holzmann. *The SPIN Model Checker — Primer and Reference Manual.* Addison-Wesley Professional, 2003.

[ISO89] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Int. Standard 8807, Int. Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1989.

[JM04] Christophe Joubert and Radu Mateescu. Distributed On-the-Fly Equivalence Checking. In Lubos Brim and Martin Leucker, editors, *Proc. of the 3^{rd} Int. Workshop on Parallel and Distributed Methods in Verification PDMC'2004 (London, UK)*, volume 128 of *Electronic Notes in Theoretical Computer Science.* Elsevier, 2004.

[JM05] Christophe Joubert and Radu Mateescu. Distributed Local Resolution of Boolean Equation Systems. In Francisco Tirado and Manuel Prieto, editors, *Proc. of the 13^{th} Euromicro Conference on Parallel, Distributed and Network-Based Processing PDP'2005 (Lugano, Switzerland)*, pages 264–271. IEEE Computer Society, February 2005.

[JM06] Christophe Joubert and Radu Mateescu. Distributed On-the-Fly Model Checking and Test Case Generation. In Antti Valmari, editor, *Proceedings of the 13th International SPIN Workshop on Model Checking of Software SPIN'2006 (Vienna, Austria)*, Lecture Notes in Computer Science. Springer Verlag, March–April 2006.

[Koz83] D. Kozen. Results on the Propositional μ-calculus. *Theoretical Computer Science*, 27 :333–354, 1983.

[Lan05] Frédéric Lang. EXP.OPEN 2.0 : A Flexible Tool Integrating Partial Order, Compositional, and On-the-Fly Verification Methods. In Jaco van de Pol, Judi Romijn, and Graeme Smith, editors, *Proc. of the 5^{th} Int. Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands)*, volume 3771 of *Lecture Notes in Computer Science*, pages 70–88. Springer Verlag, November 2005.

[Mat00] Radu Mateescu. Efficient Diagnostic Generation for Boolean Equation Systems. In Susanne Graf and Michael Schwartzbach, editors, *Proc. of 6^{th} Int. 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.

[Mat03] Radu Mateescu. A Generic On-the-Fly Solver for Alternation-Free Boolean Equation Systems. In Hubert Garavel and John Hatcliff, editors, *Proc. of the 9^{th} Int. 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.

[Mat05] Radu Mateescu. On-the-Fly State Space Reductions for Weak Equivalences. In Tiziana Margaria and Mieke Massink, editors, *Proc. of the 10th Int. Workshop on Formal Methods for Industrial Critical Systems FMICS'05 (Lisbon, Portugal)*, pages 80–89. ERCIM, ACM Computer Society Press, September 2005.

[Mat06] Radu Mateescu. CAESAR_SOLVE : A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems. *Springer Int. Journal on Software Tools for Technology Transfer (STTT)*, 8(1) :37–56, February 2006.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[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.

[NV90] Rocco De Nicola and Frits W. Vaandrager. Action versus State Based Logics for Transition Systems. In Irène Guessarian, editor, *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science (La Roche Posay, France)*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer Verlag, April 1990.

[PLM03] Gordon Pace, Frédéric Lang, and Radu Mateescu. Calculating τ-Confluence Compositionally. In Jr Warren A. Hunt and Fabio Somenzi, editors, *Proc. of the 15th Int. 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.

[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, *Proc. of the 5th Int. Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands)*, volume 3771 of *Lecture Notes in Computer Science*, pages 287–306. Springer Verlag, November 2005.

[SvBM+03] R.R.H. Schiffelers, D.A. van Beek, K.L. Man, M.A. Reniers, and J.E. Rooda. Formal Semantics of Hybrid Chi. In Kim G. Larsen and Peter Niebert, editors, *Proc. of the 1st Int. Workshop on Formal Modeling and Analysis of Timed Systems FORMATS'03 (Marseille, France)*, volume 2791 of *Lecture Notes in Computer Science*, pages 151–165. Springer Verlag, September 2003.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399