Nantes Atlantique Universités
Université de Nantes - UFR Sciences

Rapport scientifique

pour une

Habilitation à Diriger des Recherches

Informatique

Contributions aux approches formelles de développement de logiciels

Intégration de méthodes formelles et analyse multifacette

J. Christian ATTIOGBÉ

L I N A - FRE CNRS 2729

13 Septembre 2007

Jury :

Rapporteurs		
Didier	BERT	Chargé de Recherches CNRS, LIG, Grenoble
Dominique	MERY	Professeur, Université Henri-Poincaré, Nancy
William	STODDART	Principal Lecturer, University of Teessidde
Examinateurs		
Henri	HABRIAS	Professeur, Université de Nantes
Claude	JARD	Professeur, Ecole Normale Supérieure Cachan
Olivier	ROUX	Professeur, Ecole Centrale de Nantes
Véronique	VIGUIÉ DONZEAU-GOUGE	Professeur, CNAM Paris

MERCI à
– Didier BERT, Chargé de Recherches au CNRS, LIG, Grenoble,
– Dominique MERY, Professeur à l'Université Henri-Poincaré, Nancy,
– William STODDART, Professeur à l'Université de Teesside, Royaume-Uni
qui ont accepté de lire mon manuscrit, de faire de nombreuses remarques pertinentes qui ont permis d"améliorer le documenr et de faire le rapport sur les travaux que j'y ai présentés. Qu'ils trouvent tous les trois le témoignage de mon respect pour leur rigueur scientifique ainsi que mon admiration pour leur qualité humaine.

MERCI aux examinateurs
– Véronique VIGUIÉ DONZEAU-GOUGE, Professeur au Conservatoire National des Arts et Métiers, Paris,
– Claude JARD, Professeur à l'Ecole Normale Supérieure, Cachan,
– Olivier ROUX, Professeur à l'Ecole Centrale de Nantes
qui, malgré leurs nombreuses occupations scientifiques et leurs responsabilités administratives à différents niveaux, me font l'honneur de participer au Jury et d'examiner mes travaux.

Un MERCI tout particulier à Henri HABRIAS, qui m'a accueilli il y a quelques années dans son équipe à Nantes, avec qui j'ai échangé de nombreuses idées sur le plan de la recherche, de l'enseignement et sur notre société en générale ; nous n'avons pas toujours été d'accord mais c'est enrichissant ; il m'a laissé dès le début de notre collaboration, le champ libre pour développer mes recherches.

MERCI à mes collègues du département Informatique de l'UFR Sciences Nantes et du laboratoire LINA, particulièrement pour leur bonne humeur et leurs encouragements. Je ne peux oublier ici les personnels administratifs et technqiues qui nous rendent la vie quotidienne facile.

MERCI à Frédéric BENHAMOU, collègue et directeur du laboratoire LINA, pour sa confiance et son soutien constant.

Enfin, MERCI à ma petite famille pour avoir subi le partage du temps familial avec mes gourmandes préoccupations professionnelles.

à Maïthé, Charline, Maïlys et Cédric

Table des matières

Table des figures

Liste des tableaux

CHAPITRE 1

Introduction

Nous présentons dans ce mémoire pour une habilitation à diriger des recherches, un ensemble de travaux relatifs aux *approches formelles* de construction de logiciels. Nos travaux vont de la définition de langage de spécification formelle à la réalisation d'outils logiciels en passant par des systèmes d'analyse formelle. Ils relèvent à la fois des volets pragmatique et prospectif. Nous présentons alors des morceaux choisis, notamment les travaux sur l'intégration de méthodes formelles et l'analyse multifacette. Ces travaux s'inscrivent dans un historique des approches formelles, allant des travaux sur la logique formelle jusqu'aux techniques de construction systématique de programmes, que nous rappelons brièvement dans cette introduction afin de situer le contexte.

On qualifie d'approche formelle de développement, celles qui s'appuient sur des outils mathématiques divers pour construire et/ou analyser les systèmes logiciels et plus largement les systèmes informatiques (qui incluent à la fois le matériel et le logiciel). Les approches formelles ont vocation soit à construire correctement depuis l'expression des besoins (ou cahier de charges), un logiciel répondant aux fonctionnalités spécifiées, soit à analyser un système existant, soit à étudier/analyser un système avant de le développer formellement ou non.

De façon générale, on regroupe dans les approches formelles : les langages, techniques, outils et méthodes fondées sur des bases mathématiques. Elles se distinguent de certaines approches dépourvues de supports fondamentaux qui conduisent à des impasses. Au contraire, avec les approches formelles la (preuve de la) correction des systèmes par rapport à leurs spécifications initiales est une préoccupation essentielle.

Les travaux pionniers sont par exemple ceux effectués par HOARE, DIJKSTRA, BACKOUSE, MILNER, GRIES, PARNAS, BACK, etc [**Par72, Dij75, Dij76, Gri81, Bac89, BD02**]. Ces travaux eux-mêmes ce sont appuyés sur les fondements logiques de l'informatique dite fondamentale.

L'informatique fondamentale, comme discipline scientifique trouve ses fondements dans la logique formelle (BROUWER, G. FREGE, G. BOOLE, J. HERBRAND), la théorie des automates (W. MCCULLOCH, W. PITTS, KLEENE, M. RABIN, D. SCOTT, CHOMSKY), la théorie des langages (G. FREGE, D. HILBERT, K. GÖDEL), le lambda calcul (A. CHURCH, HUET), la théorie des types (B. RUSSEL) et les logiques constructives (P. MARTIN-LÖF), la théorie de la calculabilité (J. HERBRAND, A. TURING), etc

La discipline rassemble différentes spécialités ou branches qui se développent et se spécialisent sans cesse. Ces branches sont plus ou moins fondamentales, théoriques, appliquées, technologiques. Les travaux de recherche dans la discipline s'étendent ainsi des champs les plus théoriques aux champs les plus appliqués.

Parmi les branches fondamentales, il y a la construction des ordinateurs (au sens *hardware*) et la construction d'algorithmes ; ce sont des activités formelles, systématiques, pour élaborer des objets à partir de structures de bases et de méthodes de calcul, de composition et de transformation. Les structures de base sont soit des structures mathématiques élémentaires (mathématiques discrètes) soit des modèles mathématiques spécifiques plus élaborées : on parle alors de spécifications formelles.

La programmation ou l'élaboration de codes exécutables sur ordinateur à partir de modèles mathématiques (les spécifications formelles) ou d'algorithmes, est une spécialité où on relie les niveaux abstraits (élaboration d'algorithmes indépendemment de tout contexte d'exécution) et les niveaux concrets (envi-

ronnement d'exécution induisant un codage particulier). Il s'agit là des voies et moyens pour effectuer la transformation de structures abstraites en structures concrètes équivalentes. Deux principales démarches encadrent cette activité : la synthèse de programmes à partir de leur spécification (les logiques constructives servent de fondements, voir par exemple les travaux autour de la théorie des types constructifs [**Pau88**] et du système Coq [**DFH+93, BC04**]) ; le raffinement de spécifications par applications successives de règles de la théorie de raffinement (voir les travaux autour de la théorie de raffinement par DIJKSTRA, BACK ET WRIGHT [**Wir71a, BW98**], MORGAN [**Mor90**], ABRIAL [**Abr96a**], etc).

L'isomorphisme de CURRY-HOWARD [**SU98, SU06**] est un cadre théorique fondamental qui sert de cadre unificateur des deux champs énoncés ci-dessus. Il s'énonce comme suit : la construction d'un programme à partir de spécifications est similaire à la construction d'une preuve à partir d'axiomes. En somme, programmer c'est prouver et réciproquement. On a ainsi la possibilité d'extraire ou de synthétiser un programme à partir d'une preuve ou de façon équivalente de construire pas à pas le programme en même temps que la preuve.

Le logiciel est un ensemble de programmes coopérants et destinés à rendre des services spécifiques aux utilisateurs. La construction de programmes fonde la construction de logiciels. La double problématique de maîtrise de la construction et de la complexité de logiciels a motivé les travaux sur la modularité (procédures, classes/objets, composants, aspects) et la réutilisation de programmes.

La construction de programmes *corrects* a motivé toute une branche allant de la logique de HOARE/FLOYD [**Hoa69**] à la théorie du raffinement de BACK, BACK-WRIGHT [**BW98**], en passant par la preuve de correction à posteriori de programmes HOARE [**Hoa76**].

Malgré les dizaines d'années passées entre les premières pratiques et recherches en Informatique (A. TURING 1936+, VON NEWMAN 1946, etc) et les pratiques d'aujourd'hui, il existe relativement peu de méthodes établies pour le développement industriel, ou à grande échelle, de systèmes informatiques exempts de fautes. Les méthodes de construction progressive basées sur la théorie du raffinement introduite par BACK et WRIGHT, ont un impact relativement faible en milieu industriel.

La pratique industrielle aujourd'hui repose en grande partie (en dehors de quelques domaines comme l'aérospatial et le ferroviaire) sur des technologies ad hoc plus ou moins rigoureuses, soumises à des contraintes économiques de productivité immédiate : il existe peu d'industries (éditeurs de logiciels, constructeurs de systèmes embarquant du logiciels, etc) qui développent les logiciels sur des bases systématiques et formelles. On a souvent affaire à des méthodes ad hoc, non normalisées, non pérennes.

Il y a des percées significatives notables qui présagent d'un meilleur lendemain. Les réseaux de Pétri ont du succès (notammen en Allemagne) et sont largement utilisés en industrie (voir pour exemple, sur `www.daimi.au.dk/CPnets/intro/example_indu.html`, la liste de quelques projets ou d'entreprises utilisant les réseaux de Petri). Les algèbres de processus (CSP/FDR, Lotos/CADP normalisé ISO, ACP/MuCRL, etc) sont aussi utilisées dans de nombreux projets industriels et dans des entreprises.

La méthode B a du succès dans le domaine ferroviaire notamment les systèmes de contrôle embarqués dans les équipements roulants ou plus récemment dans le système de contrôle de portes palières [**PPS06**]. La méthode B [**Abr96a**] est de la famille des méthodes de développement formel basées sur les techniques de raffinement. La preuve de théorème est utilisée pour décharger les preuves de correction (cohérence) et les preuves de raffinement. Des développements de la méthode, connues sous le nom *Event B* (ou B événementiel) [**Abr96b, AM98, Abr02**] permettent de développer plus généralement des systèmes à événements discrets. L'approche de *Event B* est très proche de celle des *Action Systems* [**BKS83, BM95**]. La méthode B

est bien outillée avec des plateformes industrielles (telles que Atelier-B[1], B-Toolkit[2]) et des plateformes plus académiques (telles que B4free[3], JBTools[4]) qui aident les développeurs dans la réalisation de leurs projets.

Les approches dites Objet (O-J. DAHL, K. NYGAARD [**DN66, DN67**], [**Cox86**]) constituent indéniablement une avancée technologique importante par rapport aux techniques de développement modulaire (E.W. DIJKSTRA [**Dij72**], D. PARNAS [**Par72, Par75**], etc) mais elles ont montré leurs limitations par rapport à la correction, la maintenance et la réutilisation dans de très grands projets et pour la répartition des applications sur le réseau Internet. D'où les nouveaux développements autour des composants et des langages plus ouverts sur les systèmes d'exploitation (afin de tirer profit des possibilités déjà offertes au niveau des systèmes d'exploitation).

Les techniques de d'évaluation ou vérification de modèles à posteriori (*model checking*) améliorent les logiciels en aidant à la vérification des propriétés, la détection des fautes, mais ne fournissent pas des moyens pour mieux développer ces logiciels. Elles traitent surtout les systèmes finis.

L'approche par programmation directe, exploite au mieux la puissance des langages de programmation, répond aux besoins pressants des entreprises et des sociétés de services informatiques ; elle est de loin la plus pratiquée : elle nécessite une bonne expérience des programmeurs, la maîtrise des environnements de programmation et des domaines d'application. Cependant cette approche se révèle coûteuse à long terme (voir le rapport *The CHAOS Report (1994)* du Standish Group qui montre diverses causes des échecs de projets informatiques) ; en effet l'activité de maintenance est très coûteuse lorsque les bases du développement ne sont pas rigoureuses.

D'autres approches rigoureuses mais moins pratiquées à grande échelle sont par exemple : l'approche de construction et de preuves d'algorithmes/programmes (FLOYD, HOARE, DIJKSTRA, KNUTH) [**Gri81, Bac89**] et l'approche algorithmique (*Art of programming* de D. KNUTH [**Knu97**]) ; ici on s'attache à la construction d'algorithmes *efficaces*, la construction de structure des données, avec des outils d'analyse utilisant les résultats de la calculabilité, la complexité des problèmes, etc. Il y a des champs de recherche très actifs sur ces thématiques (algorithmique, théorie des graphes et applications, combinatoires, etc) ; les résultats de ces champs nourrissent d'autres domaines en facilitant le développement de programmes/logiciels de calculs scientifiques, ou la mise à disposition d'algorithmes corrects dont les programmes résultats constituent des bibliothèques communautaires de programmes corrects. Ils peuvent constituer dans une certaine mesure (lorsque les algorithmes et leur programmation sont prouvés corrects) la garantie de correction de bibliothèques de structures de données ou de procédures, voir de composants logiciels.

Les approches formelles se situent dans la lignée des approches de construction d'algorithmes évoquées ci-dessus. En effet il s'agit, de la même manière que précédemment, de construire progressivement le logiciel correct en justifiant toutes les différentes étapes de sa construction. La différence est sur le produit (un algorithme dans un cas, un logiciel exécutable dans l'autre cas) et les méthodes employées. Aussi, dans un cas on reste sur des structures mathématiques ou sur des structures algorithmiques alors que dans l'autre cas il s'agit de transformations successives à partir de structures abstraites, le passage de structures mathématiques aux structures informatiques.

Où en sommes-nous aujourdhui sur le développement et l'utilisation de programmes ou logiciels corrects ? Lorsqu'un logiciel de base ou d'application ne revêt pas un caractère critique pour son utilisateur, la correction du logiciel est mise en second plan. Lorsque le logiciel est un composant critique d'un système

[1] ClearSy, France, www.atelierb.societe.com

[2] B-Core, UK, `www.b-core.com`

[3] `www.b4free.com`

[4] `lifc.univ-fcomte.fr/ tatibouet/JBTOOLS/`

global ou bien lorsqu'il revêt un caractère critique pour son utilisateur, on a une toute autre vision de la nécessité de correction du logiciel. Plusieurs exemples dans la littérature ont mis l'accent sur cette nécessité : le bug du processeur Intel (1994), l'échec du vol 501 de Ariane V (1996), etc

De façon générale, les systèmes logiciels font de plus en plus partie intégrante de systèmes plus généraux ; le problème de la correction ou de la fiabilité du logiciel est accru. Le développement des technologies de l'information et de la communication de façon générale, exige une maîtrise de la gestion des équipements (devenus par conséquent complexes et critiques) qui ne peut plus se faire de façon empirique. Le recours à des pratiques rigoureuses est d'ores et déjà indispensable.

Dans ce mémoire, nous présentons une série de travaux effectués sur plusieurs années et des pistes de développement futur de ces travaux en vue de contribuer aux avancées dans le domaine du *logiciel*.

Plan du mémoire. Ce mémoire est organisé en six parties.

La première partie est consacrée aux matériaux et techniques utilisés ou nécessaires à la lecture de ce mémoire.

Dans la deuxième partie nous présentons des travaux effectués dans le cadre de l'intégration de méthodes formelles. Cette partie est constituée des chapitres 3 et 4. Le premier est consacré à la problématique générale de l'intégration de méthodes formelles. Le chapitre 4 est consacré à une intération des réseaux de Pétri et de la méthode B.

La troisième partie est consacrée à des contriutions à la méthode formelle B. Nous y présentons le chapitre 5 consacré à la communication de systèmes abstraits B ; ensuite le chapitre 6 traite d'une méthode de spécification en B de systèmes multiprocessus à architecture variable.

Une quatrième partie de ce document est dédiée à l'analyse formelle de systèmes avec le chapitre 7 où nous présentons une approche d'analyse multifacette et le chapitre 8 où nous proposons nous esquissons une algèbre de spécifications multiparadigmes.

La cinquième partie est consacrée à la présentation de deux des plateformes que nous avons développées pour expérimenter nos pistes de recherche : la plateforme NatIF/ORYX et la plateforme ATACORA.

Enfin la sixième partie évoque des conclusions et surtout des perspectives de nos travaux. Nous y traitons quelques perspectives de recherches et d'enseignements dans le chapitre 11.

Première partie

Matériaux et techniques

Table des Matières

CHAPITRE 2

Matériaux et techniques

1. Matériaux

1.1. Vocabulaire, préliminaires.

Système : Un système est un ensemble d'objets, réels ou abstraits, qui interagissent et forment un tout. Tout objet ou entité du système interagit ou est lié avec d'autres objets du système.

Un **système complexe** est un système où les interactions entre les objets sont difficiles à définir ou que leur nombre est très grand. C'est par conséquent un système dont l'analyse, la modélisation ou le développement n'est pas trivial.

Modèle : Un modèle est une construction ou une représentation abstraite (théorique) d'un objet, d'un système, d'un phénomène ou même d'une théorie. Une telle construction est faite à l'aide d'objets abstraits, d'ensembles de variables et de relations entre ces variables.

Un modèle permet de raisonner, selon des logiques ou des systèmes formels, sur l'objet ou le système dont il est l'abstration. Les objets abstraits utilisés pour construire un modèle sont des structures mathématiques telles que des ensembles, des relations et fonctions, des tables, des formules, etc ; les relations entre les objets et variables sont symboliques ou quantitatives : formules logiques, systèmes d'axiomes, règles logiques, etc.

L'utilisation de modèles abstraits en génie logiciel est courant ; par exemple l'emploi de graphes pour modéliser différentes situations, l'emploi d'automates à états pour raisonner sur le comportement d'un système, l'emploi de processus stochastiques pour pour simuuler des systèmes, l'emploi de polygones pour représenter des objets graphiques, etc

Système logique : C'est un ensemble d'équations (ou axiomes), reliant des formules dans un langage logique spécifique (par exemple la logique du premier ordre). Un système logique est le support de raisonnement tel que la déduction, la réécriture (où les axiomes sont orientés pour donner des règles).

Opérations sur les systèmes.

Analyse formelle : C'est l'étude des propriétés d'un système à travers un modèle abstrait (mathématique) qui le représente.

Spécification : Spécifier c'est construire un modèle abstrait (formel, mathématique). La spécification est la construction ou la description (syntaxique) qui induit un modèle (sémantique).

Vérification de propriétés : C'est l'examen d'un modèle afin de savoir s'il satisfait ou non certaines propriétés. Cet examen est fait soit par la preuve de théorèmes qui expriment les propriétés ; soit par l'évaluation du modèle (*model checking*) en explorant entièrement le modèle pour confirmer ou infirmer les propriétés. Dans ce dernier cas, le modèle est très souvent encodé par un graphe dont on explore les noeuds.

Développement formel : Développer un système c'est construire (correctement) le système à partir d'un modèle abstrait qui le représente. Le développement est ainsi une transformation systématique de spécification en codes exécutables mais il peut aussi bien nécessiter des outils appropriés pour construire les modèles puis les transformer de l'abstrait vers le concret. Le développement peut se faire aussi par réutilisation de l'existant : algorithmes, modules, codes et bibliothèques.

1.2. Structures, logiques, théories (modèles de descriptions). Nous rappelons ici très brièvement les concepts fondamentaux pour la construction de modèles formels.

Les modèles sont décrits par des concepts ou des structures mathématiques. Les langages fournissent le moyen formel pour l'expression ou la description des structures ou des modèles et des systèmes. Le raisonnement formel sur les modèles est effectué à l'aide de logique ou de théorie.

Graphe : C'est une structure mathématique définie par un ensemble d'éléments appelés noeuds ou sommets munie d'une relation entre les nœuds décrivant les arcs.

Structure de KRIPKE : Voir Modèle de KRIPKE ci-dessous. Les structures de KRIPKE sont utilisées par exemple pour l'évaluation de modèles (*Model checking*).

La logique classique (avec le tiers-exclu) : du premier ordre constitue le fondement de la plupart des langages de spécification. Elle permet de formaliser des énoncés représentant des données ou des propriétés sur les données ou des systèmes. Un énoncé étant soit vrai soit faux.

La logique intuitionniste (ou constructive): À la différence de la logique classique, elle permet d'établir la vérité des énoncés à partir de leur construction. La logique intuitionniste fonde la correspondance ou isomorhisme de CURRY-HOWARD qui établit l'analogie entre la preuve de théorèmes à partir d'axiomes et la développement de programmes à partir de spécifications.

La logique modale : elle ajoute des modes (temporels avec les opérateurs *toujours*, *possible*, *jusqu'à*, etc ;) à la logique classique pour effectuer des raisonnements spécifiques dans le temps.

La logique (classique) d'ordre supérieur : Elle étend la logique des prédicats du premier ordre en permettant l'emploi de variables pour représenter des fonctions ou d'autres prédicats dans les expressions manipulées. Dans la pratique on a recours à un système de type pour catégoriser les variables.

Système de propriétés (présentation) : Il s'agit d'un ensemble d'axiomes exprimant des propriétés, algébriques ou non, qui caractérisent un modèle ou une structure mathématique.

Théorie: Une théorie (du 1er ordre) est tout ensemble de formules logiques (du 1er ordre) dans un langage donné précisément défini.

La théorie des types (comme branche de la logique) : Elle a été introduite par B. RUSSEL pour résoudre les paradoxes de la théorie des ensembles. Ici, les objets manipulés ne sont plus des éléments d'un ensemble mais des fonctions dont les arguments et les valeurs retournées ont un type.

La théorie des ensembles: CANTOR définit un ensemble comme une collection d'objets abstraits appelés éléments. La théorie (naïve) des ensembles permet de raisonner sur les structures mathématiques. L'axiomatisation de la théorie (théorie des ensembles axiomatisée), considère comme point de départ non plus les ensembles mais l'appartenance des éléments (aux ensembles), c'est-à-dire, sont ensembles des collections d'éléments ayant certaines propriétés.

Théorie des automates : La théorie des automates étudie les automates et permet de comprendre leur fonctionnement et leur combinaison.

Théorie des catégories : C'est une théorie introduite en 1945 par SAMUEL EILENBERG et SAUNDERS MACLANE pour étudier les structures mathématiques et les relations entre elles.

1.3. Modèles sémantiques (modèles d'interprétation).

HOARE : A scientific theory is formalised as a mathematical model of reality, from which can be deduced or calculated the observable properties and of a well-defined class of processes in the physical world.

Il y a deux principales notions de modèles utilisées en Informatique.

(1) Un modèle est une approximation de la réalité par une structure mathématique. Un objet O est modèle d'une réalité R, si O permet de répondre aux questions que l'on se pose sur R.

C'est l'usage proche de celui qui est fait en Mathématique ou en Physique où on construit un modèle comme un système d'équations portant sur des grandeurs (masses, énergie, ...) ou des lois hypothétiques.

(2) (Logique, théorie des modèles)
Un modèle d'une théorie T est une structure dans laquelle les axiomes de T sont valides. Une *structure* S est modèle d'une théorie T, ou bien S satisfait T si toute formule de T est satisfaite dans S. La réalité serait un modèle d'une théorie.

Ainsi un modèle est vu comme interpétation d'une spécification (formelle). Par exemple une algèbre comme modèle d'une spécification algébrique (qui est une axiomatisation).

Table de vérité: C'est une table élaborée à partir des valeurs de vérité (vrai, faux) de propositions élémentaires et étendue aux combinaisons des propositions à l'aide des connecteurs logiques dont on donne la sémantique ; les connecteurs de base sont ET ($\wedge$), OU ($\vee$), NOT ($\neg$), IMPLIQUE ($\Rightarrow$), etc.

Modèle de KRIPKE (S. KRIPKE, 1950): Il s'agit d'un ensemble d'éléments applés *mondes* muni d'une relation d'accessibilité entre mondes et d'une relation entre l'ensemble des mondes et un ensemble de formules indiquant quelles formules sont vraies dans un monde donné. On l'utilise pour la sémantique formelle des logiques non-classiques (initialement pour la logique modale mais généralisée ensuite aux logiques intuitionnistes).

Algèbre universelle: Elle étudie les structures algébriques et les relations entre elles.

Automate à états (et système de transition), Machines à états finis: Un automate à état est un ensemble d'états muni d'une relation d'accessiblité entre ces états. Un automate [*(finite) state automaton - FSA, ou (finite) state machine - FSM*] est constitué d'états et de transitions entre ses états formant un graphe orienté et étiqueté dont les sommets sont les états et les arêtes sont les transitions de l'automate. L'automate représente une machine abstraite qui fonctionne en présence d'une entrée qui est une suite de symboles ; le fonctionnement ou le comportement d'un automate est dirigé par la suite de symboles (les mêmes qui étiquettent les transitions) comme décrit ci-après : l'automate passe d'un état à un autre en suivant la transition qui correspond à la lecture du symbole courant de la suite en entrée.

Un automate est fini (machine à états finis) lorsqu'il possède un nombre fini d'états distincts. Les automates à états fini sont utilisés en théorie de la calculabilité (comme des machines abstraites) et aussi dans l'étude des langages formels. Les systèmes de transition, les réseaux de Pétri sont des généralisations ou des formes avancées des automates à états.

Système de transition: Un système de transition est un ensemble d'états muni d'une relation d'accessibilité entre états et d'une relation d'étiquettage des transitions. On distingue parmi les états un ou plusieurs états initiaux ou finaux. Les étiquettes des transitions peuvent être étendues à des instructions élaborées. Les systèmes de transition généralisent les automates à états par le fait que les transitions sont plus élaborées mais aussi par le fait que les états peuvent ne pas être enumérés mais caractériser par des propriétés ou les transitions qui les relient.

2. Techniques formelles

Un des problèmes ouverts en génie logiciel est le développement correct de systèmes informatiques non triviaux. Il s'agit de pouvoir construire de systèmes complexes, ou de très grands systèmes, à l'aide de méthodes établies et d'en fournir la garantie des fonctionnalités et des propriétés voulues et spécifiées formellement à partir du cahier de charges.

Les seules techniques qui permettent d'établir en toute rigueur la correction d'un programme ou d'un système sont celles basées sur des approches formelles (ayant des fondements mathématiques qui servent de support au raisonnement). Celles-ci préconisent soit la synthèse des programmes à partir de leur spécification formelle (en mettant en œuvre l'isomorphisme de CURRY-HOWARD), soit la construction progressive des programmes à partir de leurs spécifications. Dans les deux cas, la *preuve de théorème* est une des techniques de base.

Depuis quelques années, face à : *i)* la difficulté inhérente à l'activité de preuve de théorèmes, *ii)* le besoin de preuve de correction de programmes ou systèmes existants et *iii)* la relative lenteur de l'activité de preuve sur de grands développements, d'autres techniques de preuve de correction se sont développées : l'évaluation de modèles (*model checking*) et les tests de programmes.

Dans la suite de cette section, nous présentons brièvement les techniques de preuve de théorèmes et d'évaluation de modèles que nous avons explorées dans le cadre de nos travaux de recherche. Nous n'avons pas abordé les autres techniques de synthèse (le lecteur peut consulter [**DFH+93, BC04**]) ou les techniques de tests (voir [**OBY03**]).

2.1. Preuve de théorème (*Theorem Proving*).

La réduction d'une conjecture, par applications successives de règles de déduction et d'axiomes, à une forme normale (confondue à la valeur booléenne TRUE par hypothèse ou par interprétation) constitue une preuve (ou une démonstration) de cette conjecture ; elle devient alors un théorème.

Dans le contexte de la spécification et de la preuve de propriété de systèmes, on caractérise un système par exemple par un prédicat S à partir duquel on prouve qu'une propriété donnée P est vraie. On fait ainsi la preuve que S a la propriété P : $S \vdash P$

En somme on déduit la propriété P de S qui modélise le système.

On peut aussi prouver (démontrer) qu'une propriété P donnée (en logique du premier ordre par exemple) est vraie, en partant des axiomes de base et des règles (établies au préalable) des théories utilisées (arithmétique, logique) : $\vdash P$

En entrée d'un processus de preuve on a essentiellement : S un système logique (sous forme d'un ensemble de propriétés par exemple) et P une ou des propriétés à prouver. Le résultat est soit un succès (la preuve de la propriété est faite) soit un échec.

L'interprétation de l'échec de la preuve est délicate ; en effet, soit la propriété n'est pas démontrable (problème de l'indécidabilité) soit on n'a pas assez d'éléments ou de stratégies pour aboutir à la démonstration.

Les assistants de preuve sont des logiciels ou des plateformes logicielles qui permettent à l'utilisateur humain d'effectuer pas à pas les différentes étapes d'une preuve (sélection, application de règle, sélection et ajout d'hypothèses, ...). Ils disposent la plupart du temps d'un ensemble de théories avec leur axiomes et leurs règles (systèmes logiques).

Des exemples d'assistant de preuve sont : Nqthm, HOL, PVS, Isabelle, Coq, AtelierB, etc

2.2. Evaluation de modèles (*Model Checking*). L'évaluation de modèle consiste à explorer de façon exhaustive l'espace d'états (un graphe d'états) d'un modèle donné en vue de la vérification des propriétés qui y sont vraies ou fausses. En entrée de l'évaluation de modèle on a donc un modèle du système à vérifier, souvent un système de transition ou un graphe, et une ou plusieurs propriétés que l'on veut vérifier, souvent exprimées dans une logique temporelle. Le résultat de l'évaluation est soit une confirmation que les propriétés sont vraies dans tous les états du modèle en entrée, soit une suite d'états amenant à un état ou une propriété n'est pas vraie : c'est un contre exemple.

Techniquement, l'évaluation de modèle est effectuée sur une structure de KRIPKE (finie) qui représente le modèle M du système qu'on veut analyser. On vérifie que la structure est modèle de la propriété P considérée : $M \vDash P$.

Notons qu'ici le modèle en entrée est une abstraction plus ou moins fidèle du système réel ; en effet pour éviter la complexité du graphe et le coût de son exploration, cette abstraction est d'assez haut niveau en omettant des détails importants du système réel. Par conséquent l'interprétation du résultat de l'évaluation est délicate.

Des exemples d'évaluateurs de modèles sont : SMV (*Symbolic Model Checker for CTL*), Spin, CADP (Construction and Analysis of Distributed Processes), UPPAAL, CBMC (*Bounded Model Checker for ANSI-C programs*), BLAST (Berkeley Lazy Abstraction Software Verification Tool), etc.

Les travaux actuels se font autour des évaluateurs symboliques [**BCM$^+$92, CMCHG96**] et la parallélisation de l'exploration [**GHS01, ML04, JM04, JM06, MPS$^+$06**].

2.3. Raffinement. Le raffinement est fondamentalement une relation entre un objet abstrait et un autre moins abstrait ; l'un raffine l'autre. Une spécification S' raffine (peut ainsi raffiner) une autre spécification S : on note $S \sqsubseteq S'$.

Cette relation s'étend à l'obtention de code à partir de spécification abstraite. La transformation (systématique) d'une spécification abstraite en un objet concret tel un code concret exécutable est un processus de raffinement (voire de programmation ou d'implantation).

$$S_0 \sqsubseteq S_1 \sqsubseteq S_2 \sqsubseteq \cdots \sqsubseteq \mathit{Code}$$

Dans la pratique, on raffine une structure abstraite (décrite à l'aide d'objets mathématiques) de différentes manières : en choisissant et en remplaçant des structures abstraites par d'autres moins abstraites, en ajoutant des détails initialement écartés du cahier de charges, en ajoutant/précisant des contraintes, en levant l'indéterminisme des niveaux abstraits de spécification, en ajoutant des caractéristiques de l'exécution, etc

Une préoccupation importante ici est la question de savoir si un raffinement est correct/cohérent : si le concret correspond/equivaut bien à l'abstrait. La théorie du raffinement étudiée par de nombreux auteurs (DIJKSTRA [**Dij68**], WIRTH [**Wir71b**], BACK[**Bac80**], BACK et WRIGHT [**BW98**]) offre un cadre de raisonnement et d'analyse pour le raffinement de spécifications en codes exécutables.

Des techniques de construction rigoureuse de programmes trouvent leur fondement dans les travaux sur le raffinement.

Deuxième partie

Intégration de méthodes formelles

Table des Matières

CHAPITRE 3

Intégration de méthodes : quoi, pourquoi, comment ?

1. Motivations

Dans cette section nous explicitons quelques motivations des recherches en intégration de méthodes formelles. La question est celle de savoir comment spécifier et développer des systèmes complexes rigoureusement (sinon formellement) en utilisant conjointement plusieurs méthodes formelles appropriées à différentes facettes du système. En pratique il s'agit de trouver des méthodes formelles et des outils efficaces appropriés à de tels développements. Rappelons qu'un système est complexe non pas seulement du fait de sa taille mais essentiellement parce que son développement n'est pas trivial, il présente par exemple plusieurs caractéristiques à prendre en compte lors de son développement.

En effet, les systèmes réels (industriels ou non) présentent plusieurs caractéristiques ; or la plupart des méthodes (et outils associés) disponibles aujourdhui sont soit très générales (Z, Coq, Isabelle, ...) soit elles se focalisent sur des caractéristiques précises (CSP, LOTOS, Esterel, ...). Elles sont donc en inadéquation avec les besoins en ingénierie du logiciel.

D'où le recours, lors de l'étude et du développement d'un système donné, à l'intégration des méthodes formelles adaptées à différentes facettes ou caractéristiques identifiées. Les caractéristiques principales qui apparaissent le plus souvent dans l'expression du cahier de charges d'un système sont : la complexité des données (structure, quantité), le contrôle et l'ordonnancement des traitements sur les données, la réaction par rapport à l'environnement du système, la concurrence de déroulement de certains traitements, les contraintes de temps, la répartition sur différents processeurs ou équipements physiques, la mobilité dans le temps, etc.

2. Problématique

Aucune méthode ne peut prendre en compte simultanément les différentes facettes d'un système à développer ou analyser. De plus, l'isolation des problèmes permet de mieux les caractériser et les appréhender. Ainsi de nombreux langages et outils existent pour traiter telle ou telle autre caractéristique d'un système à un certain niveau d'abstraction.

En effet toutes les caractéristiques ne peuvent pas être positionnées sur le même plan d'abstraction. Par exemple certaines méthodes considèrent que les contraintes de répartition sur plusieurs processeurs ou les contraintes de temps relèvent du niveau implantation ; d'autres au contraire abordent le développement en insistant sur la répartition ou sur les contraintes de temps dans les phases préliminaires.

Il convient de distinguer et de clarifier les niveaux d'abstraction par rapport aux caractéristiques des cahiers des charges et aussi par rapport aux langages et méthodes de développement.

De multiples concepts et modèles sont utilisés par les langages et méthodes de spécification. Nous évoquons ci-dessous en guise d'illustration une petite liste de logiques/théories à la base de certaines méthodes formelles : théorie des ensembles (Z, B, VDM), logique classique (Otter, ...), logique intuitionniste (Coq), logique modale (TLA), logique d'ordre supérieur (HOL), théorie des types (Coq), algèbre universelle (CASL), théorie des catégories (ML), théorie des automates (expressions rationnelles), etc

Il s'en suit que pour la globalité d'un système à modéliser ou à développer, les interactions entre les spécifications exprimées dans différents langages ne sont pas triviales ; l'exploitation des raisonnements d'une

logique/théorie à une autre ne le sont pas non plus (c'est le *Transfer Problem* connu en logique [**BCLE04**] : une proriété vraie dans une logique donnée ne l'est pas toujours dans une autre logique).

De même, pour l'expression de propriétés d'un système à multiples facettes, différentes logiques peuvent être appropriées ; cependant elles n'ont pas toutes ni la même expressivité ni les mêmes contraintes. Par exemple les logiques LTL (*Linear Temporal Logic*) et CTL (*Computational Tree Logic*) n'ont pas le même pouvoir expressif. Toutes les deux exigent un automate à état comme modèle d'entrée. LTL permet d'expimer les propriétés sur les états (et les suites d'états). CTL permet d'exprmier les propriétés sur les transitions entre états. Le choix de l'une ou de l'autre dépend donc des besoins d'analyse, et moins des outils dont on dispose.

Les verrous : l'intégration de méthodes pose fondamentalement le problème (théorique et pratique) de l'interopérabilité des modèles sémantiques et des systèmes de raisonnement (sémantique, analyse de propriétés). Ce problème se manisfete pratiquement à trois niveaux principaux : l'hétérogénéité syntaxique, l'hétérogénéité sémantique et enfin l'hétérogénéité (des outils) de raisonnement. Le problème de l'interopérabilité des modèles est la conséquence de l'hétérogénéité sémantique. L'hétérogénéité syntaxique est un problème de moindre ampleur ; on peut étudier le problème sous l'angle des grammaires et langages formels ; il est en effet possible de désambiguiser des langages à peu de frais.

3. Caractérisations et solutions à l'intégration de méthodes

3.1. Compatibilité entre méthodes. Les méthodes formelles étant bâties sur le socle constitué de la syntaxe (formalisme) et de la sémantique, la résolution des problèmes d'incompatibilités présentés comme verrous est la base d'une solution à l'intégration de méthodes formelles. Lorsqu'on intègre deux ou plusieurs méthodes, chacune ayant ses caractéristiques, sa syntaxe, sa sémantique, ses outils, on doit s'assurer de la compatibilité à trois niveaux idoines. Lorsque des méthodes ne sont pas *compatibles*, on ne peut pas les intégrer.

Nous donnons en conséquence une place importante dans nos travaux à la notion de compatibilité entre méthodes.

DÉFINITION 1 (Methodes compatibles). *Deux méthodes M_1 et M_2 sont compatibles si elles ont les trois niveaux de compatibilité : compatibilité syntaxique, compatibilité sémantique (par rapport à un modèle sémantique), et compatibilité de raisonnement formel (par rapport à une logique).*

Nous traitons la compatibilité syntaxique par le biais des grammaires et langages. La compatibilité sémantique est traitée par l'utilisation de modèles sémantiques : nous utilisons les modèles sémantiques communs aux méthodes à intégrer comme base de l'intégration. Quant au niveau de raisonnement, le modèle sémantique base de l'intégration sert aussi de socle pour les outils de raisonnement.

Il n'est pas possible d'intégrer des méthodes qui ne sont pas compatibles ou alors le coût pour le faire serait prohibitif. La plus simple façon est de baser l'intégration sur des méthodes compatibles selon les trois niveaux.

Dans nos travaux, nous avons étudié les questions de l'intégration sous différents autres angles (concepts, modèles, outils) qui rejoignent celui de la syntaxe, de la sémantique et du raisonnement formel. Nous en donnons une synthèse avec une structure en couches comme indiquée dans la figure 1.

3.2. Stratégies d'intégration et de combinaison de langages et méthodes. Sur les aspects pratiques, nous avons aussi exploré les propositions ou solutions existantes rentrant dans le cadre de l'intégration de méthodes via des paradigmes des langages de spécification. Parmi ces langages et techniques de spécification existantes, on trouve :

– les approches qui combinent deux paradigmes ou plus (souvent il s'agit du paradigme des *données* et du paradigme des *comportements*), par exemple RAISE (Projet Esprit, 1988), LOTOS (1987), μ-CRL, PSF ([**MV90**]) qui combine l'algèbre de processus ACP et les spécifications algébriques écrites en ASF.
– les approches qui étendent un langage par d'autres aspects présents dans d'autres langages par exemple : Z+CSP (1990), TLA+Z (1994), Z+CCS(1996), TAA+RdP ([**Rei91, BG00**]), CASL+CCS(2000), Timed-CSP ([**RR99**]), etc
  Nous avons aussi proposé dans [**APS07**] une généralisation des approches intégrant des données formelles aux systèmes de transitions.
– les approches effectuant une intégration forte au sens où la sémantique de la combinaison des langages est homogène ; elles sont étudiées par exemple par ZAVE et JACKSON (1990+), à travers les travaux sur le multiformalisme par exemple le *Event Calculus* (1992), les machines à configurations (1998), ...
  Nous classons aussi dans cette catégorie les travaux de BOUTE [**Bou04**] sur des abstractions unificatrices qui permettraient d'intégrer plusieurs paradigmes.

Nous avons contribué à ces techniques d'intégration de méthodes en étudiant la combinaison des systèmes de transition avec des données [**Att00b, Att02c**], la combinaison des algèbres de processus avec des données, notament CCS et Casl [**SA04, SAA02a, SAA02b**] ; la combinaison de diagrammes d'état avec des données [**APS03, Chr03**].

De plus, il résulte de nos travaux, une organisation des savoir-faire en méthodes combinant des paradigmes et des techniques d'intégration. En ce qui concerne les paradigmes, il s'agit des paradigmes *données, comportement, temps*. Le paradigme des comportements est aussi invariablement appelé paradigme des *processus* (comme dans les algèbres de processus).

Dans [**Att00a**] nous avons identifié deux principales stratégies d'intégration. D'une part, une démarche selon laquelle, on s'appuie sur les langages supports des méthodes et où un des langages sert de *langage principal* et un autre sert de *langage secondaire* ; par exemple on privilégie les aspects dynamiques (ou comportementaux) et c'est une algèbre de processus qui est le langage principal, puis les aspects données sont traités dans le langage secondaire. D'autre part, une démarche selon laquelle, les langages à intégrer sont mis sur le même plan, et on effectue une extension des constructions de l'un par celles de l'autre ou bien on se sert de l'un comme un langage de référence et on y traduit l'(les) autre(s).

Nous avions de la même façon identifié trois différents principaux cas d'intégration de méthodes basés sur l'intégration deux à deux des paradigmes de données et de dynamique : *données-données*, *comportement-données*, *comportement-comportement*.

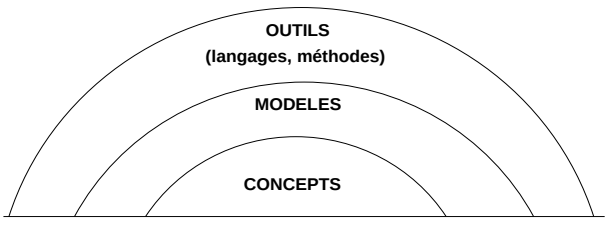

FIG. 1. Concepts, Modèles, Outils

Le principal cas est la combinaison des aspects données (via les langages comme Z, B, ASL[1]) avec les aspects dynamiques (via les LTS[2], Process Algebra, Pétri Nets, etc). Les autres types d'intégration impliquant uniquement des méthodes orientées données ou des méthodes uniquement orientée dynamiques, sont solubles par des extensions.

La prise en compte du temps est considérée comme transversale.

L'approche UML (*Unified Modeling Language*). UML [**RJB99, Obj01**] est un langage semiformel doté de représentations graphiques pour modéliser des systèmes. Les modèles sont ici un ensemble de diagrammes consacrés chacun à un aspect du système modélisé. On distingue par exemple des diagrammes de classes (comme modèle central), des diagrammes d'objets (instances des classes), des diagrammes de séquences qui donnent le comportement des objects modélisés en termes d'interaction dans le temps entre des objets donnés, des diagrammes de collaboration qui donnent le comportement entre objets mais d'un point de vue un peu plus haut que celui des diagrammes de séquences. Il y a en tout neuf types de diagrammes UML.

Du point de vue des motivations, nous partageons avec UML l'idée de spécifier différents aspects d'un système avec les formalismes appropriés et ensuite d'intégrer ces différents aspects. Mais à la différence de UML, nous mettons l'accent sur la possibilité de raisonner de façon globale sur un système intégré.

UML n'est pas initialement basée sur une sémantique formelle qui permettrait de fonder le raisonnement et l'analyse de propriétés. De nombreux travaux sont faits aussi bien sur la formalisation de UML [**LMM99, LDSM01, BLMF00, DdB00, RCA01**] que sur le développement d'outils autour de UML [**Rob99, HJGP99**]. Les travaux sur la formalisation ont notament relevé de nombreux points d'imperfection [**SG99, Sch00**] liés par exemple à la confusion entre différents niveaux d'abstraction.

La plupart de ces travaux ne considère qu'une partie des diagrammes proposés dans le cadre d'une modélisation en UML (les diagrammes de classes et les diagrammes de séquences ont ainsi été largement étudiés).

Les travaux sur les outils (ArgoUML[**Rob99, LSC03**], UMLaut[**HJGP99**], U2B[**SB06**], etc), ont donné des résultats pratiques sous forme d'outils d'édition de diagrammes ou de traduction de certains diagrammes dans des formalismes divers et de pouvoir y analyser des propriétés.

Cependant des incompatibilités sont à noter — par rapport aux sous-ensembles de diagrammes considérés, aux sémantiques considérées pour les diagrammes — et rendent impossible l'interopérabilité entre les outils et par conséquent les modèles UML sont de ce point de vue inexploitables.

Dans la suite de ce chapitre, consacré à l'intégration de méthodes formelles, nous ne considérons plus la notation UML, du fait de l'impossibilité d'y faire du raisonnement formel de façon globale.

3.3. Les techniques de plongement (*Embedding Techniques*). Les techniques de plongement (*embedding techniques*) ont été introduites en 1992 [**BGG$^+$92**] pour (ré)utiliser des plateformes logiques existant à des fins d'analyse formelle. Le principe général est de traduire un langage ou une logique source dans un langage ou une logique cible. Ces techniques ont été intensivement utilisées pour l'intégration de méthodes formelles et l'outillage de formalismes [**BG95, GP98, MR99, Att02c**].

On distingue deux principales techniques de plongement :

– Plongement superficiel (*shallow embedding*). Cette technique consiste à fixer des notations conventionnelles pour traduire les expressions du langage ou de la logique source en des termes du langage

[1] Langage de spécification algébrique

[2] Labelled Transition Systems

ou de la logique cible. Les constructions d'un langage sont aussi traduites en des constructions sémantiquement équivalentes dans le langage cible. En conséquence, ici on n'a pas besoin de (toute) la syntaxe du langage à traduire mais plutôt de la sémantique à traduire.

Cependant, avec le *shallow embedding*, au niveau raisonnement, seuls les propriétés/théorèmes exprimés dans le langage plongé sont prouvables (mais pas les propriétés/théorèmes concernant le langage lui-même).

– Plongement profond ou sémantique (*deep embedding* ou *semantic embedding*). La technique consiste à traduire la syntaxe et la sémantique du langage/logique source dans le langage/logique cible. La traduction des constructions du langage source vers le langage cible est ainsi plutôt faite dans le langage cible. Tout le langage source est traduit dans le langage cible.

  L'avantage du *plongement profond* est que, du point de vue des raisonnements, les propriétés/théorèmes concernant le/la langage/logique plongé/e peuvent être prouvés/ées dans le/la langage/logique cible.

Nous nous sommes appuyés dans nos travaux sur ces techniques pour la traduction rigoureuse des spécifications formelles et de façon générale des méthodes formelles (formalisme et systèmes de raisonnement) [**Att02c, Att05c**]. Cela permet de ramener la problématique de l'intégration de méthode à une approche (combinaison de) logique.

3.4. La pratique : une stratégie d'intégration de méthodes. L'intégration de méthodes serait systématique si une solution existait pour chacun des verrous énoncés dans la section 1. Nous avons montré qu'il est possible de définir une approche d'intégration systématique. En voici les principales étapes :

– Choisir le cas d'intégration en fonction du cahier de charges (par exemple la combinaison des paradigmes *donnée* et *dynamique*)
– Fixer une stratégie d'intégration. Par exemple le plongement dans un langage ou utilisation d'un langage comme langage principal (une algèbre de processus par exemple) et d'un autre langage comme langage secondaire. Ceci est fait en étudiant comment les logiques impliquées interagissent et comment elles peuvent être combinées.
– Décrire et préciser les compatibilités syntaxiques. Identifier les opérateurs communs et lever les ambigüités par renommage.
– Préciser la compatibilité sémantique, en faisant des choix par rapport à la stratégie d'intégration fixée ; par exemple, associer une plus grande priorité à la méthode principale par rapport à la méthode secondaire.
– Définir une stratégie de raisonnement formel ; par exemple l'utilisation d'une sémantique particulière adaptée au modèle, l'utilisation une axiomatisation dans une logique d'ordre supérieur ou la traduction systématique dans une logique tierce équipée de système de raisonnement.

Nous montrons ci-après comment notre caractérisation de l'intégration de méthodes et la stratégie proposée permettent de donner une présentation systématique des solutions qui existent dans la pratique sous l'angle de la combinaison de langages ou de paradigmes. Plusieurs illustrations sont considérées : LOTOS, ZCCS, Csp2B, Machines à Configurations, ObjectZ+CSP, Circus. Les tables 1, 2, 3, 4, 5 montrent comment les stratégies décrites sont utilisées dans la pratique.

L'étude des différents cas d'intégration montre que l'intégration des paradigmes de processus et données est le cas le plus répandu dans la pratique. On utilise un langage ou une méthode à base d'un des paradigmes comme principal et un langage ou une autre méthode ayant pour fondement l'autre paradigme comme langage ou méthode secondaire. On procède alors soit par traduction de l'une (secondaire) vers

LOTOS	
Cas d'intégration	Processus *avec* Donnée
Compatibilité syntaxique	-
Compatibilité sémantique	LTS, Sémantique opérationnelle
Stratégie employée	un langage principal (Algèbre de Processus) un secondaire (ACT ONE)
Raisonnement formel	FSM, Sémantique Opérationnelle
Analyse de propriétés	*Model checking*

TAB. 1. Positionnement de LOTOS

ZCCS	
Cas d'intégration	Processus *avec* Donnée
Compatibilité syntaxique	-
Compatibilité sémantique	LTS
Stratégie employée	un langage principal (CCS), un secondaire (Z)
Raisonnement formel	Sémantique opérationnelle
Analyse de propriétés	preuve déductive

TAB. 2. Positionnement de ZCCS

CSP-B	
Cas d'intégration	Donnée *avec* Processus
Compatibilité syntaxique	-
Compatibilité sémantique	plus faible pré-condition, LTS
Stratégie employée	un langage principal (B), un secondaire (CSP) plongement de CSP dans B
Raisonnement formel	Sémantique opérationnelle
Analyse de propriétés	preuve déductive, raffinement

TAB. 3. Positionnement de CSP-B

Machines à Configurations	
Cas d'intégration	Processus *avec* Donnée
Compatibilité syntaxique	-
Compatibilité sémantique	LTS
Stratégie employée	un langage principal (LTS/Process Algebra) un secondaire (Logique, Z)
Raisonnement formel	Sémantique opérationnelle
Analyse de propriétés	preuve déductive

TAB. 4. Positionnement des *Configuration Machines*

l'autre (principale), soit par augmentation de l'une (principale) par l'autre (secondaire).

En conclusion, il est possible de définir un cadre systématique général pour l'intégration de méthodes.

4. Généralisation

Nous avons élargi le problème de l'intégration au cadre plus général énoncé précédemment pour le développement de systèmes complexes. En dehors des aspects liés à l'intégration de méthodes, le principe de travail qui sous-tend certaines des approches actuelles de développement peut être caractérisé comme suit :

(1) étant donné un cahier de charges,

(2) on considère une méthode ou un langage puis,

(3) on spécifie et on développe le système.

Cette approche marche bien dans les cas où le cahier de charges est précisément ciblé, la nature du système à développer est bien identifiée et la méthode (avec leurs outils associés) choisie correspond bien à ce type de système.

Nous avons étudié et proposé une autre approche de travail pour faire face aux besoins d'intégration de méthodes ; en voici les étapes :

(1) faire ressortir du cahier des charges, les caractéristiques du système : les paradigmes nécessaires (abstractions, structures, propriétés) ;

(2) choisir les langages appropriés aux paradigmes ou,

(3) élaborer l'environnement de développement approprié en intégrant les paradigmes identifiés ;

(4) spécifier puis développer dans l'environnement ainsi élaboré.

CSP-OZ	
Cas d'intégration	Processus *avec* Donnée
Compatibilité syntaxique	-
Compatibilité sémantique	"Failure-Divergence Semantics"
Stratégie employée	un langage principal (Object-Z) un secondaire (CSP)
Raisonnement formel	Sémantique opérationnelle
Analyse de propriétés	*Model-checking*

TAB. 5. Positionnement de CSP-OZ

CIRCUS	
Cas d'intégration	Processus *avec* Donnée
Compatibilité syntaxique	-
Compatibilité sémantique	LTS
Stratégie employée	Traduction dans un langage principal(Z)
Raisonnement formel	Sémantique Opérationnelle, raffinement
Analyse de propriétés	-

TAB. 6. Positionnement de Circus

Types de systèmes	Méthodes appropriées
Systèmes transformationnels	Formalismes *state-oriented* (Z, B, VDM, Object-Z, etc), Formalismes *property-oriented* (Larch/LP, CASL, etc)
Systèmes réactifs	Algèbres de processus, SCR, TLA, Z, B, Langages synchrones, π-calculus I/O Automata
Systèmes distribués	Algèbres de processus, LOTOS, ODP, B
Systèmes concurrents	Algèbres de processus, LOTOS, Z, TLA
Systèmes temps-réels	Automate temporisé, Estelle, Algèbre de processus avec temps
Protocoles	SDL, LOTOS

TAB. 7. Adéquation des méthodes aux types de systèmes

4.1. Un cadre générique pour l'intégration de méthodes. Nous présentons un cadre et une *méthode d'intégration multiparadigme* basée sur la stratégie décrite ci-dessus. L'idée principale est la suivante : au lieu de définir une méthode intégrée très générale, nous construisons une méthode intégrée spécifique aux exigences particulières ou à des classes de systèmes. Cela consiste à choisir des méthodes ou formalismes adaptés aux paradigmes identifiés dans le cahier des charges. Puisque les systèmes sont différents selon leurs besoins, une approche convenable est d'élaborer la méthode intégrée associée à chaque système ou classe de système ; cela justifie l'idée de la génération de méthode intégrée à partir des besoins exprimés dans le cahier de charges.

La figure 2 (où les flêches indiquent les plongements) résume le principe du cadre générique proposé.

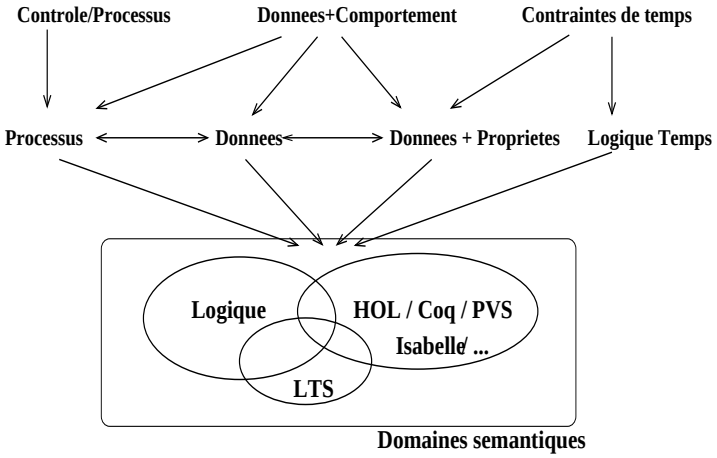

FIG. 2. Un cadre d'intégration générique

Le choix des méthodes à intégrer dépend maintenant des besoins. Cela veut dire qu'à partir des besoins ou des exigences, des caractéristiques sont identifiées et elles conduisent vers les paradigmes fournis par certaines méthodes qui seront intégrées. Les paradigmes guident ainsi l'intégration des méthodes. L'intégration est accomplie après l'analyse du système à étudier/développer.

Dans le cas spécifique de classe de systèmes avec des caractéristiques et besoins communs, il est possible d'engendrer à l'avance une méthode intégrée appropriée qui peut être paramétrée. Une piste explorée dans cette direction est la considération des classes de méthodes traitant le même type de système comme illustrée par les tableau 7.

Une méthode d'intégration générique peut être élaborée pour chaque classe de systèmes en considérant les méthodes de la classe. Par exemple comment intégrer les méthodes (Algèbre de processus, TLA, Z, B,

langages synchrones) qui sont dans la même classe "Systèmes réactifs" ? Nous présentons une telle méthode d'intégration dans la suite.

L'approche présentée dans la figure 3 qui caractérise la plupart des travaux existant en intégration de méthodes doit être confrontée avec celle de la figure 2 qui illustre notre proposition basée sur les besoins et les paradigmes alors que les autres approches bâtissent l'intégration sur les méthodes plutôt que sur les besoins.

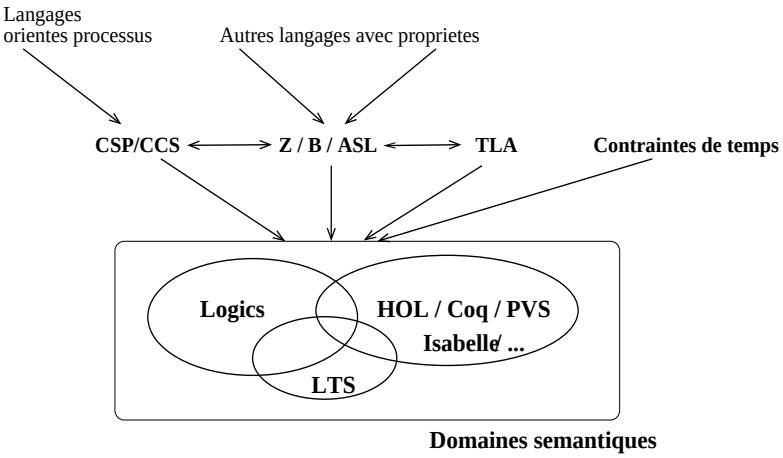

FIG. 3. Un cadre d'intégration classique

4.2. Nouvelle stratégie d'intégration des méthodes. Les *bases d'intégration* et la compatibilité peuvent aider à l'intégration de méthodes. Nous avons proposé une approche d'intégration basée sur la considération des *bases sémantiques* des méthodes.

4.2.1. *Bases formelles pour l'intégration.* Lorsque l'intégration est possible (selon la compatibilité), une autre difficulté est de trouver une base homogène convenable afin de raisonner sur des objets venant de différentes méthodes ; nous considérons à cet effet les *bases d'intégration* et les *domaines d'intégration*. C'est une réponse à la question de trouver un niveau de raisonnement convenant au raisonnement par rapport à différentes logiques.

DÉFINITION 2 (Bases d'intégration sémantique). *Une base d'intégration sémantique pour des méthodes données est un modèle sémantique qui permet le raisonnement formel sur les (parties des) méthodes formelles : elle peut être vue comme un niveau de compatibilité commun ou bien une logique abstraite dans laquelle on peut raisonner de façon cohérente sur les méthodes considérées.*

Une base d'intégration sémantique peut être incluse dans une autre et elle peut partager des caractéristiques avec d'autres.

Exemple de base d'intégration sémantique : la logique du premier ordre. En effet la logique du premier ordre permet le raisonnement sur plusieurs formalismes qui l'utilisent pour décrire les objets. Par exemple si on considère les descriptions de deux objets (issus de différents langages) puis leurs traductions en logique du premier ordre sous forme de prédicats, alors les prédicats résultants peuvent être combinés et manipulés avec les opérateurs de la logique.

Lorsqu'une base d'intégration existe, il est toujours possible de traduire la sémantique des objets (d'un formalisme et d'un paradigme) dans cette base, de les intégrer et d'y raisonner (éventuellement de traduire

les résultats dans des formalismes cibles) comme illustré par la figure 4.

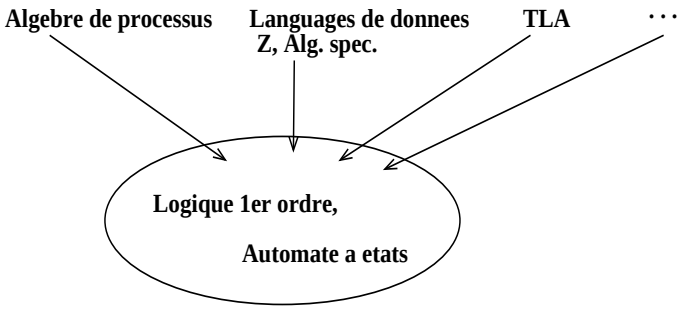

FIG. 4. Une base d'intégration simple

Bases d'intégration élémentaires. La figure 4 illustre l'emploi de bases d'intégration élémentaires afin d'intégrer diverses méthodes ou formalismes. Les bases d'intégration considérées sont la logique du premier ordre, les systèmes de transition et les logiques d'ordre supérieur.

- la logique du premier ordre (FOL), comme indiqué précédemment, est une base d'intégration formelle.
- les systèmes de transition constituent une base d'intégration. En effet, il est bien connu que les systèmes de transition (avec leurs différentes variantes) peuvent être utilisées pour traiter des systèmes dynamiques complexes. Les théories sous-jacentes sont bien étudiées et l'état de l'art montre que plusieurs systèmes opérationnels s'en servent comme format de représentation interne (utilisation des systèmes à états finis) et comme support de raisonnement.
- les logiques d'ordre supérieur (HOL), sont de bons candidats comme bases d'intégration. Considérons comme illustration : HOL [**Gor93**], Isabelle [**Pau93a**],[**Pau93b**], théorie des types [**GLT89**], la théorie des catégories [**BW90, AG91**]. En effet ces plate-formes logiques fournissent des structures puissantes pour la description formelle d'objets et le raisonnement formel.

Il est donc possible de construire des méthodes intégrées qui peuvent couvrir plusieurs paradigmes avec une base de raisonnement homogène.

Pratiquement, nous avons besoin de la définition de passerelles entre les méthodes aux niveaux syntaxique et sémantique. Nous avons introduit [**Att00a**] la notion de passerelle de spécifications (*specification bridges*) à cet effet. La passerelle est définie selon les trois niveaux de compatibilité.

Le principe d'une passerelle est de traduire (ou plonger) le modèle qui sous-tend une technique/méthode dans un modèle ou formalisme donné qui peut être utilisé au niveau de l'intégration. La notion de bases d'intégration est élargie à celle de *domaine de compatibilité*.

4.2.2. *Domaine de compatibilité.* Dans la suite la notion de compatibilité donne un aboutissement à l'idée de classes de méthodes partageant la même logique ou partageant les mêmes paradigmes.
Les domaines de compatibilité généralisent les bases d'intégration vues précédemment.

DÉFINITION 3 (Domaine de compatibilité). *Un Domaine de compatibilité est défini comme une catégorie de caractéristiques de méthodes telles que, tout couple de méthodes considérées dans la catégorie, est compatible par rapport à une logique ou un paradigme.*

Exemple de domaines de compatibilité élémentaires

- *modèles sémantiques* : logique, systèmes de transition étiquetées (*labelled transition systems*) sémantiques de traces, logique temporelle, plus faibles pré-conditions, modèle de KRIPKE, algèbre ; etc

Domaine de compatibilité : *(Labelled) Transition Systems*			
Méthodes / Formalismes	Caractéristiques (Données, Processus)	Type de systèmes	Modèles de description
MEC	Automate à états	dynamique, Communicant	Automate
Process Algebra	Process, Communication	dynamique, concurrence, interaction	Processus

TAB. 8. Système de transition comme domaine de compatibilité

Domaine de compatibilité : *Algèbre de Processus*			
Méthodes / Formalismes	Caractéristiques (Donnée, Processus)	Type de systèmes	Modèles de description
CCS, CSP, ACP, LOTOS	Processus, Communication	dynamique, concurrence	Processus communicants
π-calculus	Processus, Mobilité	Concurrence, Réactif, Mobile	Processus communicants

TAB. 9. Algèbre de processus comme domaine de compatibilité

Domaine de compatibilité : *Bisimulation*			
Méthodes/ Formalismes	Caractéristiques (Données, Processus)	Type de systèmes	Modèles de description
CCS	Processus communic.	Concurrent, Réactif	Algèbre de processus
π-calculus	Processus, Mobilité	Concurrence, Réactif	Algèbre de processus
LOTOS	Donnée, Processus	Concurrence, Réactif, Protocoles	Algèbre de processus

TAB. 10. Bisimulation comme domaine de compatibilité

– *descriptions syntaxiques* : grammaires, logiques, théorie des ensembles ; etc
– *domaines d'application* : systèmes de transformation, systèmes séquentiels, systèmes répartis, systèmes temps-réels, systèmes réactifs, systèmes concurrents, etc.

Les tables 8, 9, suivantes 10 illustratent l'emploi des domaines de compatibilité.

L'idée concrète est que les domaines d'intégration pour l'intégration de méthodes doivent être plus généraux que les domaines élémentaires listés précédemment : ils doivent fournir des cadres homogènes pour le développement formel ou le raisonnement formel sur les méthodes intégrées.

Par exemple, en considérant les modèles sémantiques comme domaines de compatibilité, une logique d'ordre supérieur est un domaine d'intégration. Les algèbres de processus comme modèles de description constituent un autre exemple de domaine d'intégration. En effet on a dans les deux cas, en plus de la compatibilité, la possibilité de raisonnement.

Des relations entre domaines sont nécessaires afin de traiter les approches multiparadigmes.
– est-ce qu'une méthode peut appartenir à plus d'un domaine d'intégration ?
– si oui, qu'en est-il de la transitivité (cf. Table 11) ?

Domaines	Méthodes dans le domaine
D_1	M_1, M_2, M_4
D_2	M_3, M_5
D_3	M_1, M_6

TAB. 11. Relation entre domaines d'intégration

M_1 est compatible avec M_4 parce qu'elles sont dans le même domaine D_1 ; M_6 est compatible avec M_1 ; est-ce que M_6 et M_4 sont compatibles ?

Un exemple illustratif est donné dans la table 12.

Domaines	Méthodes dans le domaine
Logiques temporelles	TLA, Unity
Plus faibles pré-conditions	B, *Action Systems*
Théorie du raffinement	B, *Action Systems*, CSP
Algèbre de Processus	CSP, CCS, ACP, LOTOS

TAB. 12. Exemples de méthodes domaine-compatibles

CSP est compatible avec B, CSP est compatible avec LOTOS, par conséquent, la question est : est-ce que B et LOTOS sont compatibles ?

La réponse peut être donnée selon les domaines de compatibilité. En considérant ici le raffinement et les algèbres de processus, il y a compatibilité puisque les algèbres de processus permettent aussi le raffinement. Cependant la technique de raffinement (bissimulation) utilisée pour les algèbres de processus n'est pas la même que celle utilisée pour les spécifications B ; elles préservent cependant la correction fonctionnelle et par conséquent une certaine compatibilité.

Cependant les autres niveaux de compatibilité doivent être assurés (syntaxe, sémantique).

Domaine de compatibilité syntaxique.

PROPOSITION 1. *domaines de compatibilité syntaxique*
Tous les langages qui utilisent la même logique pour décrire leurs structures sont dans le même domaine.

La *logique du premier ordre* est un bon exemple de domaine de compatibilité syntaxique.

Domaine de compatibilité sémantique. La finalité ici est d'assurer la compatibilité (sémantique) selon les modèles sémantiques.

PROPOSITION 2. *(Domaines de compatibilité sémantique)*
Toutes les méthodes qui emploient les mêmes modèles sémantiques sont dans le même domaine sémantique.

En corollaire, deux méthodes prises dans un domaine de compatibilité sont compatibles et peuvent être intégrées.

Par exemple *B, CSP, Action Systems* partagent comme base sémantique des systèmes de transitions (gardées), ils sont de plus compatibles par rapport au raffinement.

Ce travail sur la compatibilité nous a permis d'élaborer un système d'information d'aide aux méthodes formelles et de développer une bases de données pour l'aide à l'analyse de cahier de charges et à l'intégration de méthodes (C'est le projet NatIF/Oryx [**AS02, SA04**]).

4.2.3. *Compatibilité de méthodes.* La relation de compatibilité est étendue à d'autres aspects du développement : raffinement de données, raffinement d'opérations.

Le raffinement et la synthèse sont les principales techniques utilisées en développement ; nous avons introduit les relations *FR-compatibilité* et de "raffino-compatibilité" *refinement-compatibility* entre les méthodes.

DÉFINITION 4. *(Compatibilité par rapport au raisonnement formel (FR))*
Deux ou plusieurs techniques/méthodes sont FR-compatibles si elles utilisent la même technique de raisonnement formel (systeme formel, techniques de preuve, système de raffinement, etc).

PROPOSITION 3. *(Refinement-Compatibility — rafino-compatibilité)*
Deux ou plusieurs méthodes sont refino-compatibles si elles permettent un raffinement des spécifications par la même technique (de raffinement).

La *Méthode B* [**Abr96a**] et les *Action Systems* [**BKS83**] sont *raffino-compatibles*. En effet, par rapport aux opérations, ces méthodes sont compatibles (systèmes de transition et utilisation des plus faibles préconditions pour modéliser les opérations) et utilisent aussi les mêmes techniques de raffinement.

Compte-tenu de ce qui précède, on a les éléments pour : analyser une approche d'intégration donnée, donner des recommandations pour certaines approches, proscrire certaines autres, et finalement mettre en place des guides généraux pour l'intégration de méthodes.

Voici sous forme d'étapes un exemple de guide élaboré pour l'intégration de méthodes :

(1) Analyse orientée caractéristiques des méthodes concernées : spécificités, modèles de description, types de systèmes pour lesquels elles sont appropriées. En conséquence, identification des bases d'intégration.

(2) Choix de base d'intégration commune le cas échéant en fonction des caractéristiques précédemment identifiées.

(3) Définition de passerelles entre spécifications et leur sémantique en plongeant les méthodes dans la base d'intégration choisie.

(4) Plongement dans les logiques d'ordre supérieur pour les cas non triviaux.

Un usage pratique de la compatibilité est la définition de passerelle (figure 5) entre spécifications et leurs sémantiques.

4.2.4. *Passerelle de spécification et modèles sémantiques.* La construction de passerelle entre spécifications, comme c'est illustré en Figure 5, est une application directe de la notion de compatibilité.

En guise d'illustration, considérons *CSP* et *B* ; elles sont *refinement-compatible* et de plus elles utilisent toutes les deux les descriptions en forme de systèmes de transitions. Ainsi une passerelle peut être

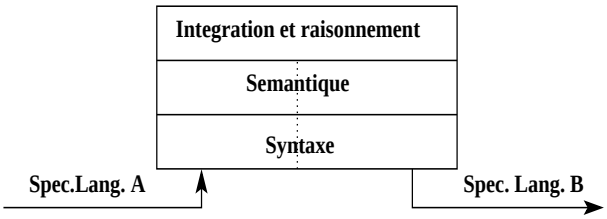

FIG. 5. Principe des passerelles entre spécifications

construite entre *CSP* et *B* par traduction des constructions de la première en celles de la seconde. Les systèmes de transitions étiquetées servent de support de la traduction ; ils permettent de capturer la sémantique des spécifications *CSP* afin de construire les spécifications *B* correctes. Comme exemple, prenons le travail de BUTLER [**But99**] qui décrit la traduction des spécifications *CSP* en machines abstraites *B* afin d'exploiter les outils de *B*.

On peut généraliser le principe de plongement d'un formalisme dans un autre à celui du plongement d'un modèle sémantique (support des paradigmes de spécification) dans un autre. L'intérêt de cette démarche est que le raisonnement formel peut être effectué de façon homogène pour l'intégration multiparadigme.

4.3. Liens avec d'autres approches.

4.3.1. *Composition de modules algébriques.* MICHEL et WIELS [**MW97**], poursuivant les travaux présentés dans [**SW96**], ont proposé un cadre basé sur la logique et la théorie des catégories, pour la composition de modules de spécifications. Leur approche, une extension du calcul de modules[3] à la EHRIG et MAHR [**EM90**], consiste à spécifier un système comme l'interconnexion de modules. Chaque module est constitué de quatre parties décrites par des spécifications combinant signature et propriétés logiques associées aux données et au comportement. L'interconnexion des modules est décrite par des morphismes entre les quatre parties composant les différents modules.

Cette approche est une intégration de modules dans un cadre homogène fixé ; tous les modules composés sont décrits dans le même contexte, avec le même formalisme et avec le même cadre sémantique.

Cependant, il peut servir de support pour une intégration de méthodes (hétérogènes) si on peut ramener, moyennant la compatibilité sémantique, les spécifications des modèles à composer au module au sens de WIELS et al. Cela revient à reconstruire tous les modules à intégrer dans le cadre logique et catégorique ciblé. Le coût de cette reconstruction peut être considérable ; il s'exprime par le plongement sémantique (ie au niveau des langages et leur sémantiques) des formalismes supports des modèles dans le cadre logique ciblé.

Les travaux présentés dans [**ADW04**] se situent dans ce dernier contexte ; les auteurs composent des modules ou composants décrivant diverses vues d'un même système en les ramenant à des institutions (pour obtenir un cadre de composition catégorique homogène) puis en utilisant les relations de synchronisation entre les institutions pour lier les différents composants.

4.3.2. *L'approche HetCASL.* HetCASL (Heterogenoeous CASL [**Mos03, Mos05**]) est une extension de CASL [**ABK$^+$02**] qui permet la combinaison de différentes spécifications CASL (*Common Algebraic Specification Language* du projet CoFI [4]).

CASL est développé sur la base des *institutions* proposées par GOGUEN et BURSTALL comme cadre catégorique pour s'abstraire de différents systèmes logiques. Une institution est vue comme une catégorie

[3] ici les modules sont décrits par des types abstraits algébriques

[4] Common Framework Initiative for Algebraic Specification and Development

de signatures, des foncteurs de signatures et de modèles et une relation de satisfaction qui permet d'établir qu'une expression satisfait un modèle ou non.

Une spécification élémentaire CASL est définie par une classe de signatures, une classe de modèles, un ensemble d'axiomes pour chaque signature, et une relation de satisfaction entre les modèles et les signatures.

Plusieurs extensions ou des variantes de CASL ont été proposées : CASL-LTL, HaCASL, CSP-CASL, CoCASL, etc ; se pose alors la question de l'intégration de différentes spécifications CASL. HetCASL apporte une réponse à cette question.

L'extension HetCASL est faite par rapport à la structuration des spécifications CASL. Ainsi HetCASL combine différentes spécifications en indiquant les logiques utilisées, les liens entre elles (inclusion) et les relations de traduction entre les différentes logiques.

HetCASL est un cadre pour raisonner sur des spécifications "CASL-compatibles" (extensions ou sous-langages de CASL) mais pouvant avoir des logiques différentes. Pour ce faire HetCASL permet de construire un graphe de logiques (qui peuvent être des institutions, morphismes, comorphismes ou transformations de morphismes d'institutions) dans lequel toute logique peut être traduite dans une autre logique qui dispose d'un système de raisonnement.

Par rapport à nos travaux, nous situons l'approche de HetCASL dans le cadre de l'intégration générique restreinte à un domaine de compatibilité qui est CASL ; en effet HetCASL traite d'une même famille de langage autour de CASL ; les langages intégrés doivent être des extensions ou des sous-langages de CASL.

4.4. Sur l'*intégrabilité*. Nous proposons de baser l'*intégrabilité des modèles* sur l'interopérabilité sémantique entre les modèles : en effet, lorsque les modèles sémantiques utilisés sont compatibles, il est possible d'encoder logiquement un modèle par un autre et de pouvoir y raisonner. C'est là une condition nécessaire, "vérifiable" avant d'envisager l'intégration de méthodes. La vérification peut se faire dans un premier temps par rapport à une base de compatibilité. Au delà de cette solution, les approches logiques de combinaison de systèmes de raisonnement et la théorie des modèles peuvent servir de cadre de raisonnement.

Dans cette direction nous travaillons sur un concept d'*unification généralisée de modèles*, semblable à l'unification de termes en logique.

PROPOSITION 4 (Unification généralisée de modèles). *L'unification généralisée $\mathcal{U}$ de modèles, consiste à trouver sous les contraintes de compatibilité, un modèle unificateur et des contextes logiques tels qu'on puisse passer du modèle unificateur aux modèles de départ en prenant en compte les contextes.*

Un exemple d'illustration de ce concept est celui des systèmes de transitions. Etant donnés deux systèmes de transitions étiquettées S_1 et S_2, est-il possible de trouver un système de transition S_u et un contexte (morphisme de systèmes : renommage d'états, de transitions, identification de transitions) tels que S_1 et S_2 soient réductibles à S_u ?

Exemple.
Soient
m_1 un LTS : $\{(e_0, l_1, e_1), (e_1, l2, e_3), (e_1, l_3, e_2)\}$ et
m_2 un LTS : $\{(f_0, a_1, f_1), (f_1, a_2, f_2), (f_2, a_3, f_3), (f_2, a_4, f_4), (f_4, a_5, f_4)\}$;
$\mathcal{U}(m_1, m2)$ est la substitution d'états $\{e_0 = f_0, e_1 = f_2, e_3 = f_3, e_2 = f_4\}$ avec la substitution d'étiquettes $\{l_1 = a_1, a_2 = \epsilon, a_3 = l_2, l_3 = a_4, a_5 = \epsilon\}$.
On peut par conséquent considérer 4 états ($q_0 = e_0 = f_0$, $q_1 = e_1 = f_2$, $q_2 = e_3 = f_3$, $q_4 = e_2 = f_4$) et 3 transitions pour les transition non-ϵ ($t_1 = l_1 = a_1$, $t_2 = a_3 = l_2$, $t_3 = l_3 = a_4$) de telle sorte que un modèle unificateur soit : $\{(q_0, t_1, q_1), (q_1, \epsilon, q_2), (q_2, t_2, q_3), (q_2, t_3, q_4), (q_4, \epsilon, q_4)\}$.

4.5. L'approche UTP de Hoare et He. UTP (*Unifying Theories of Programming*) est un cadre formel, proposé par HOARE et HE [**HH98**], pour

- unifier les inombrables théories de spécification/programmation couvrant différents paradigmes (concurrence, communication, composition séquentielle, non-déterminisme, terminaison, raffinement), différents syles de programmation (impératif, fonctionnel, logique) et pour
- raisonner sur différents paradigmes de programmation en leur donnant une sémantique dénotationnelle dans un modèle relationnel. En fait UTP relie de façon homogène les trois principales approches de raisonnement sémantique sur les spécifications et programmes : l'approche dénotationnelle, l'approche opérationnelle et l'approche algébrique.

Un des principaux objectifs de UTP est de modéliser[5] tout programme (mais aussi la spécification ou la conception (*design*)) de façon homogène comme un ensemble de relations appelées *design* ; une relation est ici un prédicat sur une collection de variables (*alphabet* du prédicat). De plus les relations sont reliées entre elles par une variable spécifique qui traite du démarrage et de la terminaison des programmes impliquées.

Il y a actuellement de nombreux travaux autour de UTP [**DS06**] aussi bien sur les aspects théoriques (formalisation de sémantiques de différents formalismes dans le cadre UTP) que sur la définition d'outils.

Le cadre UTP nous semble un cadre cible pour expérimenter les idées présentées dans ce chapitre ; d'une part on peut envisager le plongement de spécifications ou de sémantiques (de différentes méthodes) dans une base à la UTP afin de raisonner sur ces spécifications ; d'autre part on peut envisager une base UTP pour l'intégration générique basée sur les paradigmes qui sous-tendent les langages de spécification ; il s'agit de formaliser le noyau regroupant les différents paradigmes dans le cadre UTP.

5. Autres caractérisations logiques et catégoriques

En considérant une méthode comme constituée de concepts, langages, techniques et outils, l'intégration est vue sous différents angles relatifs aux constituants.

Les bases sont par conséquent l'intégration des concepts et des langages. Plus spécifiquement, il s'agit, hormis les aspects syntaxiques, de l'intégration de logiques et de sémantiques (donc des modèles sous-jacents).

5.1. Approche logique.

PROPOSITION 5. *L'intégration de méthodes/modèles est réductible en la combinaison de systèmes logiques.*

Considérons la décomposition de modèles M_i selon les trois niveaux présentés précédemment : $M_i = \langle L_i, R_i \rangle$ avec L_i le langage correspondant aux deux niveaux syntaxique et sémantique et R_i correspondant au troisième niveau (du raisonnement formel).

L'intégration des méthodes M_i et M_j définie avec $L_i \uplus L_j$ la fusion des langages et $R_i \uplus R_j$ la combinaison des systèmes de raisonnement est exactement le fibrage (*fibring*, [**CSS99**]) de M_i et M_j (considérés comme modèles logiques).

Combinaison de logiques.
La combinaison de systèmes (de raisonnement) logiques a été longuement étudié dans le cadre de la logique formelle [**BdR97, Gab96a, dCH96, CSS99, KR00**].

Nous avons mis en évidence ci-dessus le lien entre l'intégration de méthodes et la combinaison de logiques. Nous donnons ici quelques définitions utiles à l'approfondissement de ce lien.

[5] *predicative model* : une approche relationnelle utilisant des prédicats

Il existe plusieurs techniques de combinaisons de logiques : la fusion, le fibrage de logiques (*fibring*), la paramétrisation (*parametrization*) de logiques et la synchronisation de logiques (*synchronization*) [**SSC97, CSS99, CSS98, CSS05**].

La *fusion* de logiques (modales) a d'abord étudié [**MKa91, KW97, Kra99**] avant les techniques plus générales comme celle de fibrage (*fibring* par Gabbay[**Gab96b**]).

La *synchronisation* de deux systèmes logiques (chacun composé d'un langage, d'un système d'inférence, d'un ensemble de modèles et d'une relation de satisfaction) est un système logique dont le langage est l'union disjointe des deux langages, le système d'inférence est l'union disjointe des deux systèmes d'inférence et dont les modèles sont les paires constituées d'un modèle d'un système et d'un modèle de l'autre système. La relation de satisfaction est l'extension des relations de satisfaction des deux systèmes.

La *fusion* de deux systèmes (logiques) modaux est une logique bimodale contenant les deux opérateurs modaux initiaux et les connecteurs propositionnels.

Le *fibrage* de deux systèmes logiques est un système logique dont le langage est l'union des constructeurs des deux langages, le système d'inférences est l'union des règles d'inférence des deux systèmes, les modèles sont les paires constituées d'un modèle d'un système et d'un modèle de l'autre système. La relation de satisfaction (ou d'interprétation) est l'extension des relations de satisfaction des deux systèmes.

La *paramétrisation* consiste à remplacer une partie d'un système logique donné par une autre logique. Le système d'inférence du système paramétré est ainsi instancié à certains endroits par le système d'inférence du paramètre, on a ainsi deux niveaux distincts de raisonnement, l'un (le système paramétré) primant sur l'autre.

Ainsi l'augmentation d'un langage de spécification par un autre (avec ou sans la relation de méthode principale ou secondaire), est vue comme un fibrage de logiques (celles sous-jacentes aux langages de spécifications) ou une paramétrisation de logiques.

5.2. Raisonnement multi-systèmes. Dans la pratique l'intégration de méthodes, se traduit par l'intégration de différents modèles correspondants à des sous-systèmes d'un même système général ou à différents aspects (modèles) des sous-systèmes.

Soient n sous-systèmes $S_1, S_2, S_3, \cdots S_n$, composants d'un système S et chacun vérifiant les propriétés $P_1, P_2, P_3, \cdots$: on note $\{\langle S_i, P_i\rangle\}$.

Différents cas sont envisageables pour raisonner globalement sur S.

- Soit S a des propriétés globales énoncées P (on exige $S \models P$) et dans ce cas les sous-systèmes doivent préserver P ;
- Soit S n'a pas de propriétés globales imposées aux sous-systèmes.

Cas 1 : S a des propriétés globales que devraient garantir ses sous-systèmes ; les propriétés sont exprimées par des variables logiques opérant sur l'espace d'état de S et par conséquent sur les variables d'état de S.
Dans ce cas, les sous-systèmes S_i composant S sont

- soit indépendants de S et par conséquent n'utilisent aucune variable de S (ou bien des S_i indépendants entre eux) ; par conséquent on ne peut rien conclure sur la garantie par S_i des propriétés globales P ; dans ce cas les sous-systèmes doivent être revus afin d'employer des variables de S permettant ainsi de les étudier par rapport à P et aussi entre les S_i ;
- soit dépendants totalement ou partiellement de S et par conséquent utilisent des variables d'état de S ; $S_i \models P_i$ et la question est $S_i \models P$?
  considérant la portée globale des variables (*vg*) de $\langle S, P\rangle$ et la portée locale des variables (*vl*) de $\langle S_i, P_i\rangle$, l'intersection (vg_i) entre *vg* et *vl* est la partie de S concernée par S_i ; on devrait prouver que

$S_i \models P_{vg_i}$ si P_{vg_i} exprime les propriétés relatives à vg_i ; de même les sous-systèmes peuvent avoir des recouvrements entre leurs variables, dans ce cas, les propriétés à l'intersection doivent être aussi préservées : si $vl_i \cap vl_j = vl_{ij}$ alors on exige que $S_i \models P_{ij}$ (idem pour S_j) ; mais cela découle de $S_i \models P_i$;
Cas 2 : S n'a pas de propriétés globales ; les composants S_i de S sont soit indépendants entre eux soit dépendants entre eux.

Obligations de preuve selon les cas.
Relation entre les S_i et S

Variables utilisées	Expression des propriétés
$S_i : vg, vl_i$	$P_i(vg) \wedge P_i(vl_i) \wedge P_i(vg, vl_i)$

Les obligations de preuve sont :

$$(OP_{gl}) \begin{cases} P_i(vg) \wedge P(vg) \\ P_i(vl_i) \\ P_i(vg, vl_i) \wedge P(vg) \end{cases}$$

Relation entre les S_i :

Variables utilisées	Expression des propriétés
$S_i : vg, vl_i$	$P_i(vg) \wedge P_i(vl_i) \wedge P_i(vg, vl_i)$
$S_j : vg, vl_j$	$P_j(vg) \wedge P_j(vl_j) \wedge P_j(vg, vl_j)$

Soit $vl_{ij} = vl_i \cap vl_j$,
– soit $vl_{ij} = \{\}$ et il ne reste que les obligations OP_{gl}
– soit $vl_{ij} \neq \{\}$ et en plus des obligations OP_{gl}, on prouve

$$(OP_{ll}) \begin{cases} P_i(vl_i) \wedge P_j(vl_j) \text{ par conséquent } P_i(vl_{ij}) \text{ et } P_j(vl_{ij}) \\ P_i(vg, vl_i) \wedge P_j(vg, vl_j) \end{cases}$$

En fait, on peut voir à travers ces obligations de preuve la conservation (ou transfert) de propriétés entre différents modèles ou systèmes logiques.

5.3. Approche catégorique. Nous conjecturons que sous la contrainte de compatibilité, un cadre catégorique permettra d'exprimer l'intégration entre méthodes, en termes de plongement de modèles dans un autre modèles (à l'aide de *pullback*) ou comme nous le verrons plus tard, en termes de dérivation de modèles (*pushout*) à partir d'autres modèles.

Il existe plusieurs travaux qui ont exploré le cadre catégorique pour la spécification de systèmes sur lesquels nous pouvons nous appuyer.

A la suite des travaux pionniers de EHRIG et MAHR [**EM90**], GOGUEN et BURSTALL [**GB92**], FIADERO et MAIBAUM [**FM92, FM93, Fia06**] ont proposé la gestion de modules et leur composition parallèle en utilisant la théorie des catégories.

De même, T. MOSSAKOWSKI a proposé un HetCASL cadre catégorique [**Mos03, Mos05**] pour combiner des spécifications écrites avec des variantes de Casl.

Ces travaux considèrent à priori un cadre algébrique homogène et non des spécifications hétérogènes. Cependant, ces travaux sont fondateurs et peuvent être mis à profit pour les contextes hétérogènes.

SEGUIN et WIELS [**SW96**] ont utilisé une approche catégorique pour la vérification de systèmes tolérents aux pannes. Dans [**WE98**] les auteurs ont exploré l'utilisation d'une approche catégorique pour gérer

l'évolution des spécifications ainsi que leurs relations pendant le processus de développement. Une difficulté que nous relevons ici est la lourde charge laissée au spécifieur/développeur ; ce dernier doit (ré)écrire les traductions non triviales des spécifications dans d'autres formalismes.

Par rapport à ces travaux, qui imposent un contexte (algébrique) précis aux modules ou composants à intégrer, l'approche catégorique que nous souhaitons explorer en complément de l'approche logique présentée précédemment, ne part pas de cette hypothèse d'un contexte précis, mais plutôt de partir d'un modèle pour en construire/dériver d'autres dans des contextes différents mais compatibles. Par exemple on voudrait construire à partir d'une spécification à l'aide d'automates à états finis, d'autres spécifications en B, réseaux de Petri, algèbre de processus, systèmes de transition étendus, etc. Partant d'un cadre formel où ces différents modèles sont reliés, il serait plus aisé (du fait d'une assistance technique pré-établie) d'intégrer et d'analyser des spécifications adhoc. L'idée est que, à l'instar du plongement sémantique (voir section 3.3 du chapitre 3), les structures et les sémantiques des formalismes considérés sont précédemment reliées.

CHAPITRE 4

Intégration de B et des réseaux de Petri

Nous présentons dans cette section une illustration des méthodes d'intégration : le plongement des réseaux de PETRI dans les systèmes abstraits B. Nous adoptons la technique de plongement sémantique (*deep embedding*, cf Section 3.3) en traduisant en B les structures statiques et aussi la sémantique d'évolution des réseaux de PETRI. La structure statique d'un réseau de PETRI est capturée dans un système abstrait via une structure de graphe. Ce système abstrait est alors inclus dans un autre système abstrait qui capture la sémantique d'évolution des réseaux de PETRI. La sémantique d'évolution se résume en une collection d'événements *B* selon les stratégies choisies : réseaux élémentaires ou réseaux à haut niveau. Le plongement permet d'utiliser conjointement les réseaux de PETRI et les systèmes abstraits B dans un même développement, mais à différentes étapes et pour des analyses diverses.

1. Motivations

Le développement de systèmes sûrs exige l'emploi de concepts, langages, méthodes et outils qui sont fournis par les approches formelles. Il existe de nombreuses méthodes ; cependant elles sont souvent mono-paradigmes et, très souvent l'appréhension des systèmes réels dépassent le cadre couvert par les techniques de spécification mono-paradigmes ; de plus la complexité de ces systèmes exige une intégration adéquate de techniques et méthodes appropriées aussi bien pour l'analyse formelle que pour le développement.

Les efforts de recherche se concentrent sur la combinaison de diverses approches et des outils spécifiques associés afin de renforcer leur impact sur le traitement des systèmes industriels.

Il est nécessaire de rendre les méthodes formelles plus pratiques et efficaces dans leur usage : *i)* elles doivent être reliées aux techniques et pratiques d'ingénierie ; *ii)* elles doivent être mieux outillées en fournissant des plate-formes de développement opérationnelles et puissantes. Ces points sont encore des défis pour la communauté de recherche en méthodes formelles et ils motivent en partie nos travaux.

L'intégration de diverses méthodes formelles peut être motivée par différents types de combinaisons : la complémentarité des méthodes afin de couvrir les différentes facettes du système à développer, le besoin de techniques spécifiques telles que le raffinement et la composition, ou des techniques de raisonnement spécifiques tels que la preuve de théorème ou l'évaluation de modèle (*model checking*), ou des considérations pratiques telles que des formalismes graphiques et l'interopérabilité entre des outils supports.

Dans ce travail nous avons étudié l'intégration des réseaux de PETRI et de B afin d'utiliser conjointement les deux approches complémentaires dans le même développement. Le formalisme des réseaux de PETRI peut être utilisé comme une interface d'entrée graphique pour les projets de développement en B. La méthode B suivrait pour compléter l'analyse formelle du système modélisé à l'aide de réseaux de PETRI.

Le formalisme des réseaux de PETRI est largement utilisé [**Mur89, RR98, Rei98, KJ04**] par des ingénieurs et aussi dans des projets académiques. Il dispose d'outils graphiques, d'outils de simulation et de vérification de propriétés de vivacité utilisant des techniques d'évaluation de modèle. . B est une approche orientée modèle qui permet le développement correct par raffinement à partir de spécifications abstraites vers des codes concrets exécutables ; elle est basée sur la preuve de théorème et offre essentiellement des techniques de vérification de propriétés.

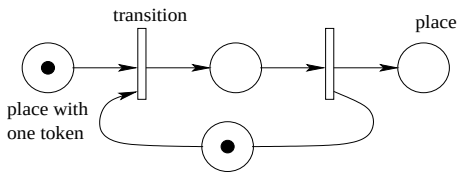

FIG. 1. Exemple de réseau Petri

Notre travail dans ce contexte a consisté en : *i)* la définition d'une structure *B* générique qui capture les modèles de réseaux de PETRI et leurs sémantiques ; *ii)* la systématisation du plongement des structures de réseaux de PETRI et leur sémantique d'évolution en B événementiel.

Ce travail a conduit au développement d'une passerelle entre les réseaux de PETRI et B. Le reste de ce chapitre est organisé comme suit : nous rappelons brièvement le formalisme des réseaux de PETRI, puis nous présentons le plongement des réseaux de PETRI dans B en considérant d'abord les réseaux élémentaires puis en généralisant aux réseaux de haut niveau. Ensuite nous abordons l'aspect analyse de propriétés.

2. Réseaux de Petri et systèmes abstraits B

2.1. Un aperçu des réseaux de PETRI (RdP). Formellement, un RdP est un quadruplet $(P,\ T,\ Pre,\ Post)$ où :

- P est un ensemble fini de places, (avec $\mid P \mid = m$, le cardinal de P) ;
- T est un ensemble fini de transitions, (avec $\mid T \mid = n$, le cardinal de T) ;
- P et T sont des ensembles disjoints ($P \cap T = \{\}$) ;
- $Pre\ :\ P \times T \rightarrow \mathbb{N}$ est une fonction d'entrée, $Pre(p,t)$ dénote le nombre d'arcs allant de la place p vers la transition t ;
- $Post\ :\ P \times T \rightarrow \mathbb{N}$ est une fonction, $Post(p,t)$ dénote le nombre d'arcs allant de la transition t vers la place p.

Pratiquement, un RdP est un graphe orienté bipartite dont les arcs connectent les nœuds de deux différents ensembles : l'ensemble des places et l'ensemble de transitions. Les réseaux de PETRI disposent d'un formalisme graphique où les places sont connectées aux transitions par des arcs orientés.

Graphe associé à un RdP. Le graphe associé au réseau N est décrit par :

- Γ_p les transitions atteignables à partir de chaque place :
  $\forall p \in P\,.\,\Gamma_p(p) = \{t \in T \mid Pre(p,\ t) > 0\}$
- Γ_t les places atteignables à partir de chaque transition :
  $\forall t \in T\,.\,\Gamma_t(t) = \{p \in P \mid Post(p,\ t) > 0\}$
- W_{in} le poids de chaque arc d'entrée : $\forall\ p \in P, \forall\ t \in T\,.\,W_{in}(p,\ t) = Pre(p,t)$ et
- W_{out} le poids de chaque arc de sortie : $\forall\ p \in P, \forall\ t \in T\,.\,W_{out}(p,\ t) = Post(p,t)$

Le graphe associé au RdP est la représentation abstraite qui est utilisée pour manipuler le graphe.

Les places connectées à une transition avec un arc venant de chaque place sont les places d'entrée de la transition. Les places connectées à une transition avec un arc allant de la transition vers chaque place sont les places en sortie de la transition.

Marquage d'un réseau de PETRI. Un réseau marqué $M_N = (N,\ \mu)$ est constitué d'un réseau N et d'une fonction $\mu\ :\ P \rightarrow \mathbb{N}$.
$\mu(p)$ est le nombre de jetons dans la place p ; c'est le *marquage* de la place p. Le marquage initial M_0 d'un réseau est le n-uplet constitué du marquage initial de toutes les places p_i du réseau : $M_0 = (\mu(p_1), \cdots, \mu(p_m))$

où m est le nombre de places.

Comportement d'un RdP. Un RdP évolue en franchissant des transitions *autorisées*. Une transition est *autorisée* si toutes ses places d'entrée contiennent au moins assez de jetons que le nombre indiqué sur les transitions allant de ces places vers la transition. En l'absence d'indication ce nombre est 1. Une transition autorisée peut être franchie ; elle autorise en conséquence les actions associées aux places de sortie de la transition. Il y a un choix non-déterministe entre les transitions autorisées.

Le franchissement d'une transition modifie les marquages des places en entrée et en sortie. Cela peut autoriser ou non d'autres transitions. Toutes les transitions autorisées peuvent être franchies. Par conséquent l'évolution du réseau décrit un marquage du réseau qui peut être infini. Quand une transition est franchie, un jeton est enlevée de chaque place en entrée de la transition et un jeton est ajouté à chaque place en sortie de la transition.

Ceci est généralisé en enlevant (resp. ajoutant) le nombre de jetons correspondant au poids des arcs en entrée de la transition (resp. au poids des arcs de la transition vers les places de sortie).

2.2. Un aperçu des systèmes abstraits B. Les *systèmes abstraits* sont les structures de base de B événementiel. Ils remplacent ici les *machines abstraites* qui sont les structures de base de la méthode B classique [**Abr96a**] qui est orientée opération. Un système abstrait définit un modèle mathématique du comportement[1] d'un système. Ce modèle est essentiellement constitué d'une description d'états (à l'aide de constantes, variables et invariant) et de plusieurs descriptions d'événements. Un système abstrait est comme une machine abstraite où les événements remplacent les opérations. Les systèmes abstraits sont comparables aux *Action Systems* de Back [**BKS83**] ; ils décrivent une évolution indéterministe d'un système à l'aide d'actions gardées. Des contraintes dynamiques peuvent être exprimées sur les systèmes abstraits pour spécifier diverses propriétés de vivacité [**AM98**]. Les systèmes abstraits peuvent être raffinés comme les machines abstraites.

Les données d'un système abstrait. On adopte ici une vision globale d'un système. Les données contenues dans un système abstrait sont communes à un modèle tout entier, distribué ou non. Les données qui y sont modélisées correspondent ainsi à tous les sous-systèmes du système distribué. Les systèmes abstraits ont été utilisés pour spécifier le comportement de systèmes distribués [**Abr96b, BW96**].

Les événements d'un système abstrait. En B, un événement correspond, comme dans l'approche des *action systems*, à l'observation d'une transition du système. Les événements sont spontanés et montrent comment le système évolue. Un événement possède une garde (*guard*) et une action. Un événement se produit uniquement lorsque sa garde est vraie. L'action d'un événement décrit, à l'aide de substitutions généralisées, la manière dont l'état du système évolue lorsque l'événement se produit. Les gardes de plusieurs événements peuvent être vraies simultanément ; dans ce cas un seul d'entre ces événements se produit. Le système effectue un choix indéterministe interne. Lorsqu'aucune garde n'est vraie, le système est bloqué (*deadlock*).

Un événement a une des formes générales décrites dans la figure 2 où bv dénote les variables locales de l'événement ; gcv dénote les constantes et variables globales du système abstrait contenant l'événement. $P(bv, gcv)$ est un prédicat ; $GS(bv, gcv)$ est une substitution généralisée qui modélise l'action de l'événement.
La garde d'un événement avec la forme ANY est $\exists(bv).P(bv, gcv)$. La forme SELECT est un cas particulier de la précédente. La garde d'un événement ayant cette forme est $P(gcv)$.

[1] Le comportement d'un système est l'ensemble des suites de transitions possibles à partir de son état initial.

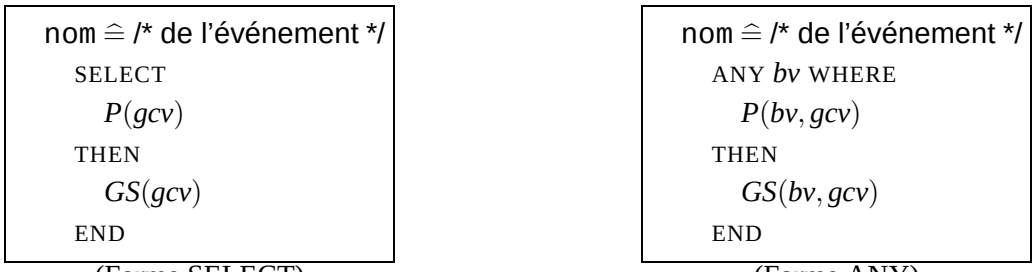

FIG. 2. Formes générales des événements

Sémantique et cohérence (consistency). Un système abstrait décrit un modèle mathématique qui simule le comportement d'un système. Sa sémantique est induite par l'invariant et est établie par des obligations de preuve. La cohérence du modèle est alors établie par des obligations de preuve ; on doit prouver que : l'initialisation doit établir l'invariant et chaque événement du système abstrait doit préserver l'invariant du modèle . Dans le cas d'un événement avec la forme ANY, l'obligation de preuve est :

$$I(gcv) \wedge P(bv, gcv) \wedge \mathsf{term}(GS(bv, gcv)) \Rightarrow [GS(bv, gcv)]I(gcv)$$

où $I(gcv)$ est l'invariant du système abstrait.Le prédicat $\mathsf{term}(S)$ exprime que la substitution S termine.

3. Plongement des réseaux en B événementiel

En choisissant le plongement sémantique (cf. section 3.3), on a non seulement besoin de la traduction syntaxique du langage source (en préservant la sémantique) mais aussi, sa sémantique doit être traduite dans la cible.

3.1. Plongement de la structure des réseaux de PETRI en B. Au niveau syntaxique, nous faisons abstraction du format graphique des réseaux de PETRI et nous considérons plutôt la structure abstraite sous-jacente (un graphe).

Du point de vue sémantique, le plongement de la structure du réseau de PETRI en B (Fig. 3) consiste à décrire le graphe associé au réseau. Le quadruplet qui décrit un réseau N est encodé avec l'ensemble des places (*places*), l'ensemble des transitions (*transitions*), et les deux relations entre places et transitions (*placesBefore*, *placesAfter*).

De plus, nous avons la fonction de marquage des places : *mu*, et des fonctions de pondération des arcs. Les poids des arcs sont des entiers naturels supérieurs ou égaux à un. Les poids des arcs d'entrée sont décrits par la fonction *weightBefore*. Les poids des arcs en sortie sont décrits par la fonction *weightAfter*. Des propriétés invariantes peuvent être ajoutées.

On obtient un système abstrait sans événements (Fig. 3) qui capture uniquement la structure de graphe d'un réseau marqué (N, mu). Ce système abstrait est progressivement complété : il reste à traiter la sémantique comportementale du réseau. Nous nous basons sur le marquage du réseau et sur les transitions.

3.2. Plongement de la sémantique d'évolution des réseaux de PETRI en B. Un réseau évolue en franchissant les transitions autorisées. A partir d'un marquage, le franchissement d'une transition autorisée, aboutit à un nouveau marquage du réseau et ainsi de suite. Cela se traduit en Event-B par un système abstrait dont les événements correspondent aux franchissements des transitions.

Une transition d'un réseau peut être formalisée, en première approximation, avec un événement B (voir Fig. 4) dont la garde exprime que toutes les places d'entrée ont le nombre de jetons requis et dont le corps

SYSTEM *PetriNet*
SETS
PLACE /* ens. des places */
; TRANSITION /* ens. des transitions */
VARIABLES
places, *transitions*, *placesBefore*, *placesAfter*, *weightBefore*, *weightAfter*, *mu*,
$\cdots$ /* autres variables (coupées ici) */
INVARIANT
places $\subseteq$ PLACE
$\wedge$ *transitions* $\subseteq$ TRANSITION
$\wedge$ *placesBefore* $\in$ *transitions* $\leftrightarrow$ *places* /* *placesBefore*$^{-1} = \Gamma_p$ */
$\wedge$ *placesAfter* $\in$ *transitions* $\leftrightarrow$ *places* /* *placesAfter* $= \Gamma_t$ */
$\wedge$ *placesBefore* $= \mathrm{dom}($*weightBefore*$)$
$\wedge$ *placesAfter* $= \mathrm{dom}($*weightAfter*$)$
$\wedge$ *weightBefore* $\in$ *transitions* $\times$ *places* $\nrightarrow \mathbb{N}$
$\wedge$ $\mathrm{dom}($*weightBefore*$)$ $=$ *placesBefore*
$\wedge$ *weightAfter* $\in$ *transitions* $\times$ *places* $\nrightarrow \mathbb{N}$
$\wedge$ $\mathrm{dom}($*weightAfter*$)$ $=$ *placesAfter*
$\wedge$ *mu* : *places* $\rightarrow \mathbb{N}$
$\wedge \cdots$

FIG. 3. Système abstrait B (partiel) encodant un réseau de Petri

(une substitution généralisée) exprime la mise à jour des places en entrée (en enlevant les jetons nécessaires) et la mise à jour des places en sortie (en ajoutant le nombre de jetons nécessaires) Les événements d'un système *B* sont instantanés et leur effet peut provoquer l'occurrence d'autres événements. Cela correspond bien avec la sémantique des réseaux de PETRI : le franchissement d'une transition t_i est instantanée et peut conduire au franchissement d'autres transitions qui ont les places de sortie de t_i parmi leurs places d'entrée.

3.2.1. *Réseaux de Petri élémentaires.* Ici réseau *élémentaire* signifie que les actions (données+opérations) ne sont associées ni aux places ni aux transitions. Le poids d'un arc peut être supérieur ou égal à 1. La garde pour le franchissement d'une transition *ti* est que toutes les places *pp* en entrée de *ti* aient le nombre requis de jetons :

$$ti \in transitions \ \wedge$$
$$\forall pp.(pp \in placesBefore[\{ti\}] \Rightarrow \mu(pp) \geq weightBefore(ti, pp))$$

L'effet de base du franchissement d'une transition est la mise à jour, via la fonction μ, des marquages des places d'entrée et de sortie par rapport aux poids des arcs entrants et sortants :

Soit $pbef = placesBefore[\{ti\}]$ les places en entrée et $paft = placesAfter[\{ti\}]$ les places en sortie de la transition *ti* ; la mise à jour des marquages des places est faite de la façon suivante :

$$\mu := \mu <+ \{pp, vv \mid pp \in pbef \wedge vv = mu(pp) - weightBefore(ti, pp)\}$$
$$<+ \{pp, uu \mid pp \in paft \wedge uu = mu(pp) + weightAfter(ti, pp)\}$$

Notons qu'il peut y avoir pour un réseau des places qui sont à la fois entrée et sortie, pour ces places, la mise à jour doit être cumulative ; pour cela il faut traiter distinctement les places seulement en entrée, les places seulement en sortie puis les places en entrée et en sortie. Soit alors

$pbef = placesBefore[\{ti\}] - placesAfter[\{ti\}]$ les places en entrée seulement,
$paft = placesAfter[\{ti\}] - placesBefore[\{ti\}]$ les places en sortie seulement et
$pcom = placesBefore[\{ti\}] \cap placesAfter[\{ti\}]$ les places en entrée et en sortie.

La mise à jour due à l'effet d'une transition est par conséquent :

$$\begin{aligned}\mu := \mu &<+ \{pp, vv \mid pp \in pbef \wedge vv = mu(pp) - weightBefore(ti, pp)\} \\ &<+ \{pp, uu \mid pp \in paft \wedge uu = mu(pp) + weightAfter(ti, pp)\} \\ &<+ \{pp, mm \mid pp \in pcom \wedge mm = mu(pp) - weightBefore(ti, pp) + weightAfter(ti, pp)\}\end{aligned}$$

De là, le franchissement d'une transition est traduite par un simple événement[2] *B* **`event_tr`** (Fig. 4) qui fonctionne pour n'importe quelle transition t_i, de façon non-déterministe.

Les variables *mupbef* et *mupaft* modélisent avec des substitutions généralisées la mise à jour de la fonction μ comme décrit au dessus. La notation $s \lhd r$ exprime la restriction du domaine de la relation r aux éléments de l'ensemble s. La notation $r_1 <+ r_2$ correspond à l'écrasement d'une relation r_1 par une autre r_2.

Pour faciliter la lecture, nous prenons délibérément un peu de liberté dans la présentation des systèmes abstraits donnés dans la suite.

`event_tr` $\widehat{=}$ /* franchissement d'une transition quelconque t_i */
ANY t_i WHERE
$t_i \in transitions$
$\wedge\ \forall pp.(pp \in placesBefore[\{ti\}] \Rightarrow \mu(pp) \geq weightBefore(t_i, pp))$
THEN
LET *pbef*, *paft*, *pcom* BE
$pbef = placesBefore[\{t_i\}] - placesAfter[\{t_i\}]$
$\wedge\ paft = placesAfter[\{t_i\}] - placesBefore[\{t_i\}]$
$\wedge\ pcom = placesAfter[\{t_i\}] \cap placesBefore[\{t_i\}]$
IN
/* *maj des places avant et après la transition t_i* */
$mu := mu <+ \{pp, vv \mid pp \in pbef \wedge vv \in NAT \wedge vv = mu(pp) - weightBefore(ti, pp)\}$
$<+ \{pp, uu \mid pp \in paft \wedge uu \in NAT \wedge uu = mu(pp) + weightAfter(ti, pp)\}$
$<+ \{pp, mm \mid pp \in pcom \wedge mm \in NAT \wedge mm = mu(pp) - weightBefore(ti, pp) + weightAfter(ti, pp)\}$
END
END

FIG. 4. Schéma d'un système abstrait modélisant l'évolution d'un réseau élémentaire

Nous formalisons la sémantique comportementale d'un réseau de PETRI élémentaire par un système abstrait avec un seul événement représentant les transitions du réseau. Ce système abstrait simule l'évolution du réseau.

Le fait d'utiliser un seul événement pour toutes les transitions plutôt qu'un événement par transition simplifie la généralisation et le raisonnement sur le plongement ; en effet, seule la structure du réseau de PETRI objet du plongement serait traduite pour chaque nouveau projet ; la sémantique globale d'évolution elle, étant déjà faite une fois pour toute.

[2]La simplicité de la spécification actuelle par rapport à une version précédente est due à D. Bert

3.2.2. *Structure générique du plongement.* Nous donnons dans la figure 5 la structure B générique qui correspond à tout modèle de réseau ; c'est un système abstrait nommé *EmbeddedPN*. Nous séparons l'encodage de la sémantique (*EmbeddedPN*) qui vaut pour n'importe quel réseau de celui de la partie statique (*PetriNet*) qui est spécifique à un système donné.

La partie statique (contenue dans le système *PetriNet*) est complétée avec des variables :

– la variable *pl_actions* est l'ensemble des actions associées aux places,

– la fonction injective[3] $pl\_treatment \in places \rightarrowtail pl\_actions$ garde l'action (ou le traitement) attachée à chaque place ; un élément spécifique *nullaction* est utilisé pour l'initialisation et pour les places sans actions.

Le système abstrait *EmbeddedPN* possède deux nouvelles variables pour la relation *trans_places* et la fonction *guard_P_actions* :

– la relation *trans_places* contient pour les transitions franchies en cours, les places de sortie qui ne sont pas encore traitées ;

– la fonction *guard_P_actions* est utilisée pour récupérer la garde du traitement de chaque place.

Par conséquent l'événement **`event_tr`** qui gère le franchissement des transitions et donc l'évolution du réseau considéré est amélioré et remplacé dans le cas du traitement de données (on parle alors de réseaux de PETRI avancés) par deux ou plusieurs événements selon la stratégie considérée pour les réseaux de PETRI ; en effet on est là dans le cadre des réseaux de PETRI avancés (voir [**Mur89, GL81**]). Nous présentons le traitement des cas relatifs dans la section 3.3.

SYSTEM *EmbeddedPN*
INCLUDES
  PetriNet /* paramètre : un RdP quelconque */
VARIABLES
  trans_places /* places de sorties des transitions franchies */
, *guard_P_actions* /* garde des actions assocées aux places */
INVARIANT
  $guard\_P\_actions \in pl\_actions \rightarrow BOOL$
$\wedge$ $trans\_places \in transitions \leftrightarrow places$
INITIALISATION
  $guard\_P\_actions := ((pl\_actions - \{nullaction\}) \times \{FALSE\})$
    $\cup\{(nullaction, TRUE)\}$
|| $trans\_places := \{\}$
EVENTS
  `event_tr` $\widehat{=}$ $\cdots$ /* franchissement de n'importe quelle ti */
END

FIG. 5. Structure générique du plongement

Nous avons étendu le plongement aux cas plus complexes : gestion des traitements associés au réseau. En effet, selon leur type (réseaux place/transition, condition/événement, ressources, etc), les réseaux de PETRI peuvent traiter des données et des actions de différentes manières. Dans certains réseaux les places avec des jetons peuvent modéliser la disponibilité de données ; dans ce cas un traitement peut être associé

[3]Elle est injective parce que nous voulons avoir une fonction inverse.

aux transitions. Dans d'autres modèles, certaines places peuvent avoir un traitement qui est ainsi gardé par une ou plusieurs transitions. C'est par exemple le cas dans un réseau modélisant un processus qui écrit des données de façon exclusive avec d'autres processus qui peuvent écrire aussi. Dans ces situations, une place spécifique est souvent utilisée (voir Fig. 6 (a)) pour gérer l'exclusion entre les processus ; seul le processus ayant franchit la transition `start` à un moment pourra déclencher l'action effectuée dans la place `writing`.

Ainsi, il n'y a pas une seule façon de plonger les réseaux de PETRI dans Event-B ; cela dépend des modèles de réseaux considérés. Nous avons exploré différents cas concernant la prise en compte des traitements associés au réseau : le cas où les traitements sont associés aux places et le cas où les traitements sont associés aux transitions.

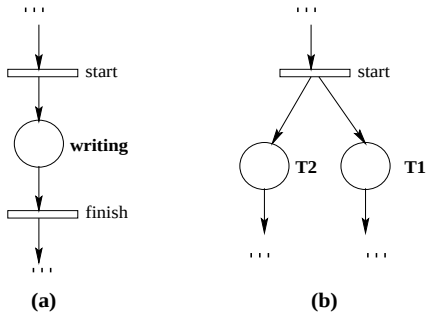

FIG. 6. Cas de traitement associé aux places du RdP

3.3. Traitement des réseaux avancés. Dans la section précédente, nous avons pris en compte l'évolution d'un réseau élémentaire, mais nous n'avons pas considéré le cas où des traitements sont associés au réseau.

3.3.1. *Réseaux de* PETRI *de haut niveau.* Les réseaux de PETRI de haut niveau (*High Level* PETRI *Nets* : HLPN) ont été introduits pour attaquer le problème de l'explosion d'états liés à la modélisation et l'analyse des cas réels un peu complexes. Les réseaux HLPN utilisent *i)* des données structurées pour modéliser les jetons, et des expressions algébriques pour annoter les éléments du réseau ; *ii)* des modes de transition pour décrire des opérations ou traitement plus élaborés.

Dans le cas des HLPN l'autorisation d'une transition dépend non seulement de la disponibilité des jetons mais aussi de leur nature. Il y a plusieurs mises en œuvre des HLPN [**Jen96**] ; les réseaux *prédicat/transition* [**Gen87**] et les réseaux de PETRI colorés [**Jen86, KJJ04**] sont deux formes de HLPN.

Nous considérons une abstraction de l'idée des réseaux HLPN. Des traitements peuvent être associés aux places ou aux transitions. Cela correspond à l'idée de jetons structurés, de places et transitions typées et plus généralement de l'exécution de certaines opérations (ou traitements) associées aux places ou aux transitions d'un réseau. En conséquence nous avons proposé un traitement générique global. Nous avons fait une étude en plusieurs étapes : d'abord nous avons examiné la formalisation dans le cas où des traitements sont associés seulement aux places. Ensuite nous avons étudié les cas où les traitements sont associés aux transitions. Finalement nous avons traité les cas où les traitements sont associés aussi bien aux places qu'aux transitions.

Réseaux avec traitements associés aux places. Le traitement associé à une place doit être effectué lorsqu' une transition d'entrée, donc une garde, qui est associée à la place est franchie. Ainsi, *chaque traitement d'une place d'un réseau est traduit comme un événement gardé d'un système abstrait*. En pratique, les actions nécessitent du temps pour être effectuées. Par conséquent le franchissement d'une transition doit se faire en deux étapes : *i)* autorisation des gardes de tous les traitements associés aux places de sortie de la

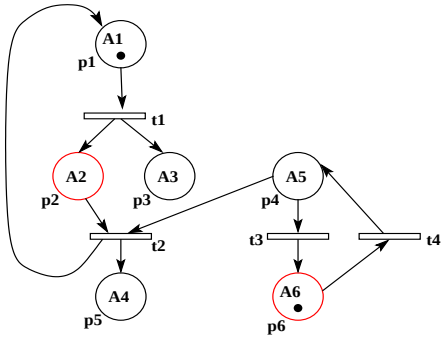

FIG. 7. Traitements interdépendants dans le RdP

transition ; *ii)* lancement de façon non-déterministe des traitements impliqués. Tous les traitements doivent être effectués.

Quelques questions se posent :

– comment régler la durée des traitements et l'autorisation des autres transitions et des autres traitements éventuellement activés ?
– doit-on attendre la fin d'un traitement avant la prise en compte d'un autre ?
– comment ordonnancer les transitions autorisées et les traitements autorisés ?

Examinons ces questions en considérant la figure 7 on s'aperçoit de la complexité de l'ordonnancement des traitements. La transition t1 autorise les traitements {A2, A3} ; t2 autorise les traitements {A4, A1} ; t3 autorise le traitement {A6} ; t4 autorise le traitement {A5}.

Ces traitements sont interdépendants parce que les places qui les contiennent sont soit en entrée soit en sortie des transitions franchies. Il y a des cycles ; par exemple le franchissement répété des transitions t3 et t4.

Pour maîtriser la situation nous utilisons les variables précédemment définies (voir la section 3.2) : *pl_treatment*, *pl_actions*, *guard_P_actions*. Le franchissement d'une transition *ti* est gérée avec deux événements qui correspondent aux deux étapes distinguées ci-dessus.

La première étape de la transition est formalisée par l'événement **fire_transition_tr** donné dans la figure 8. Les places en sortie d'une transition *ti* sont : $paft = placesAfter[\{ti\}]$. Les traitements associés à ces places sont $involved\_actions = paft \lhd pl\_treatment$.

Les gardes des traitements concernés (associés aux places de sortie des transitions franchies) sont autorisées ($\forall\, Ai \in placesAfter[\{ti\}].\, guard(Ai) := true$). La fonction *guard_P_actions* est mise à jour afin d'autoriser les gardes. Ceci est réalisé avec un produit cartésien : $ran(involved\_actions) \times \{TRUE\}$. Le marquage des places d'entrée est mis à jour. La transition franchie et ses places en sortie sont notées dans la relation *trans_places* ; cela est nécessaire pour l'ordonnancement des traitements impliqués. En effet *tous les traitements des places de sortie doivent être effectués* ***avant*** *les traitements des éventuelles transitions qu'ils ont autorisées*.

Puisque les événements B sont atomiques il n'est pas possible de mettre à jour les marquages des places de sortie durant la première étape ; elles auraient alors probablement autorisé d'autres transitions. De plus pour coller au mieux aux applications des réseaux de PETRI, nous devons considérer l'exécution complète des divers traitements pendant leur ordonnancement.

La seconde étape du franchissement est formalisée avec l'événement **`action_Ak`** (voir Fig. 9).

Un événement B est décrit pour chaque traitement associé à une place. Cela permet de gérer l'aspect haut niveau du réseau. En effet, les traitements dépendent des jetons et des transitions. La garde de chaque

fire_transition_tr $\widehat{=}$ /* franchissement de n'importe quelle ti */
ANY t_i WHERE
$t_i \in transitions$
$\wedge\ \forall pp.(pp \in placesBefore[\{ti\}] \Rightarrow \mu(pp) \geq weightBefore(t_i, pp))$
THEN
LET $pbef$, $paft$, $involved\_actions$ BE
$pbef = placesBefore[\{t_i\}] \wedge paft = placesAfter[\{t_i\}]$
$\wedge\ involved\_actions = paft \lhd pl\_treatment$
IN
/* autorisation des gardes des actions impliquées */
$guard\_P\_actions := ran(involved\_actions) \times \{TRUE\}$
/* maj des places d'entrée de t_i, */
/* les places de sortie de t_i seront maj apres les actions */
$\|\ mu := mu \lhd\!\!+ \{pp, vv \mid pp \in pbef \wedge vv \in NAT \wedge vv = mu(pp) - weightBefore(ti, pp)\}$
/* maj des places à traiter pour la transition franchie */
$\|\ trans\_places := trans\_places \cup (\{t_i\} \times paft)$
END
END

FIG. 8. Partie dynamique de la structure générique (a)

traitement est maintenue (à vraie) jusqu'à ce que le traitement soit décenché et effectué. Les traitements associés aux places de sortie qui sont couramment autorisés sont effectués de façon non-déterministe ; ils sont notés dans (le codomaine de) la fonction *trans_places*. Cependant les traitements dans les places contenues dans *trans_places* peuvent être effectués à tout moment (à cause du non-déterminisme de l'occurrence des événements).

Quand l'événement associé à un traitement est achevé sa garde est mise à faux et le nombre de jetons de la place est mis à jour : la fonction *trans_places* est modifiée, la fonction *mu* est modifiée pour refléter la mise à jour des marquages des places de sortie.

Cependant il y a quelques limites au cas courant : il y a comme une perte de priorité entre les traitements. Si l'effet d'un des traitements autorisés contribue à autoriser le franchissement d'une autre transition, les traitements autorisés par cette dernière transition peuvent s'exécuter avant les traitements déjà autorisés (ceci vient fatalement du fait de la substitution $trans\_places := trans\_places \cup (\{t_i\} \times paft)$).

Une autre limite est la suivante : lorsqu'il y a des cycles, une garde autorisée (d'un traitement) peut être surchargée (par exemple, mise à vrai de nouveau) ; c'est-à-dire que, la condition qui la valide peut être de nouveau vraie, alors que le traitement autorisé auparavant n'est pas encore effectué.

Nous résolvons ces problèmes dans le cas général en utilisant les priorités.

Réseaux de PETRI *avec traitements associés aux transitions.* De la même manière que précédemment avec le cas des places, une fonction totale $tr\_treatment \in transitions \rightarrow tr\_actions$ est utilisée pour noter le traitement associé à chaque transition. *tr_actions* est l'ensemble des traitements associés à toutes les transitions ; cette variable est définie dans la structure statique *PetriNet*. Lorsqu'une transition autorisée est franchie, le traitement qui lui est associé doit être effectué avant la mise à jour du marquage des places en sortie, sinon une autre transition pourrait prendre la priorité (c'est de la préemption) sur le traitement courant. Plusieurs transitions peuvent partager les mêmes places. Cependant, lorsque celles ci ont le nombre de jetons nécessaires pour le franchissement des transitions qui les partagent, une seule de ces transitions

action_Ak $\widehat{=}$ /* pour une action Ak (associée a la place pp) */
ANY Ak WHERE
$Ak \in actions \wedge$
$\wedge\ guard\_P\_actions(Ak) = TRUE$ /* une des actions autorisées */
THEN
LET $pp, tr, weiga, \cdots$ BE
$\wedge\ pp = pl\_treatment^{-1}(Ak)$ /* la place associée à Ak */
$\wedge\ tr = trans\_places^{-1}(pp)$ /* la transition avant pp */
$\wedge\ weiga = weightAfter(tr, pp)$ /* poids de l'arc */
$\wedge\ \cdots$ /* partie coupée */
IN
$guard\_P\_actions(Ak) := FALSE$
$\|\ mu(pp) := mu(pp) + weiga$
$\|\ trans\_places := trans\_places - \{(tr, pp)\}$
$\|\ \cdots$ /* place pour une action effective Ak */
END
END

FIG. 9. Extrait de la partie dynamique de la structure générique (b)

autorisées sera franchie. Ainsi, deux étapes sont nécessaires pour gérer le franchissement d'une transition. Dans une première étape, une des transitions autorisées est choisie de façon non-déterministe ; la garde du traitement qui lui est associé est autorisée. Le marquage de toutes les places d'entrée de la transition est mis à jour. Cela est presque similaire à l'événement **fire_transition_tr**. Dans une seconde étape, le traitement associé à la transition est effectué ; sa garde est rendue fausse puis, le marquage des places de sortie de la transition est mis à jour. Ces places peuvent autoriser d'autres transitions et ainsi de suite.

Nous obtenons deux événements B correspondant aux étapes décrites : *i)* un événement de franchissement qui est utilisé pour sélectionner une transition et pour mettre à jour les places d'entrée ; cet événement traite toutes les transitions autorisées. *ii)* le traitement associé à chaque transition a un événement dont la garde dépend du marquage des places d'entrée.

Réseaux avec traitements associés aux places et aux transitions. Dans le cas courant, lorsqu'une transition est franchie ; le traitement qui lui est associé est autorisé et le marquage des places de sortie de la transition est mis en place. Ces places de sortie ont des traitements qui doivent être autorisés. Après cela, le traitement de la transition est effectué, il autorise les traitements liés aux places de sortie. De plus, les traitements liés aux places doivent être effectués avant d'autoriser les transitions qui leur sont liées. Afin d'intégrer cette sémantique, nous utilisons en plus des variables précédentes,

- la fonction *enabled_P_actions* pour les traitements (des places) qui viennent d'être autorisés et
- la fonction *enabled_T_actions* pour les traitements (associés aux transitions) qui viennent d'être autorisés. Rappelons que *trans_places* permet de noter les places de sortie qui ne sont pas encore traitées dans le cadre d'une transition qui vient d'être franchie.

Le plongement est effectué avec des règles de priorité. La priorité entre les traitements est établie de la façon suivante. Une transition est franchie si :

- *i)* les places d'entrée ont le nombre de jetons requis,
- *ii)* il n'y a pas de traitement, lié à une place, non encore effectué (testé avec $(trans\_places = \{\})$).

En effet quand une transition est franchie, son traitement est autorisé et il autorise en conséquence des traitements associés aux places de sortie. Ils doivent tous être effectués avant le franchissement d'une autre transition. Cette stratégie résout le problème de la surcharge (ou écrasement) des gardes.

En conséquence l'évenement **fire_transition_tr** est modifié comme décrit dans la figure 10.

fire_transition_tr $\widehat{=}$ /* pour toute transition ti */
ANY t_i WHERE
$t_i \in transitions$
$\wedge\ \forall pp.(pp \in placesBefore[\{ti\}] \Rightarrow \mu(pp) \geq weightBefore(t_i, pp))$
/* et pas d'actions non effectuées (gestion de priorité) */
$\wedge$ **trans_places = {}** $\wedge$ **(enabled_P_actions** $\rhd$ **{ TRUE }) ={}**
THEN
LET $pbef, paft$, *involved_actions* BE
$pbef = placesBefore[\{t_i\}] \wedge paft = placesAfter[\{ti\}]$
$\wedge\ involved\_actions = paft \lhd pl\_treatment$
$\wedge \cdots$ /* partie coupée */
IN
$enabled\_T\_actions(t_i) := TRUE$ /* autoriser l'action de la transition */
/* autoriser la garde des actions des places concernées */
$\|\ enabled\_P\_actions := ran(involved\_actions) \times \{TRUE\}$
/* maj des places d'entrée de t_i */
/* les places de sorties de t_i ; elles seront mises à jour plus tard */
$\|\ mu := mu \Leftarrow\!+ \{pp, vv \mid pp \in pbef \wedge vv \in NAT \wedge vv = mu(pp) - weightBefore(ti, pp)\}$
$\|\ trans\_places := \{t_i\} \times paft$
END
END

FIG. 10. Extrait de la partie dynamique d'un réseau de PETRI avec des actions sur places et transitions

Le reste des événements (non détaillés ici) est comme suit :
enable_transition_action_guard ; il positionne à vraie la garde d'une transition, puis il inhibe la garde de la transition.
enable_place_action_guard ; il positionne à vraie la garde du traitement d'une place autorisée, il met à jour la fonction *mu* et la variable *trans_places* en y enlevant les places déjà traitées ;
launch_transition_action_aj ; il lance un des traitements (des transitions) dont la garde est vraie et puis il positionne à faux la garde ;
launch_place_action_ak ; cet événement lance le traitement d'une place dont la garde est vraie, puis la garde est inhibée.

Tous ces cinq événements (du système abstrait *EmbeddedPN*) simulent un déroulement entrelacé des 'exécutions' des traitements associés aux places et transitions, mais nous utilisons les priorités pour éviter les comportements anormaux des traitements.

Nous avons mené des expérimentations avec l'Atelier B ; la cohérence du système complet, représentant le réseau de PETRI, est analysée ; nous nous sommes appuyés sur des études de cas.

4. Analyse de propriétés

Deux classes de propriétés sont souvent analysées sur les modèles de réseaux de PETRI : l'une concerne la *bornitude* (le caractère borné) du réseau. Par exemple l'accumulation de jetons dans une place est un signe de mauvais fonctionnement du réseau. La seconde classe concerne la vivacité des réseaux. En étudiant l'atteignabilité de certain marquage, on peut par exemple détecter l'absence de blocage ou d'interblocage. Dans tous ces cas, le graphe de marquage (l'ensemble des marquages accessibles) doit être calculé. Cet aspect de l'analyse peut poser des problèmes. Le graphe peut avoir une taille très grande pour être analysé en un temps raisonnable ; le graphe peut aussi être infini. Lorsque le graphe est infini, un graphe de couverture est utilisé. Il permet d'analyser une partie des propriétés désirées.

La principale classes de techniques d'analyse [**Mur89, RR98**] des réseaux est : l'*analyse d'atteignabilité*. Elle est basée sur l'exploration ou la réduction de l'espace d'états en utilisant l'évaluation de modèle (*model checking*). La principale idée est de construire un graphe d'occurrence (un graphe orienté) qui possède un nœud pour chaque marquage (c'est un état atteignable du système modélisé) et un arc pour chaque transition entre marquages. L'analyse est ainsi basée sur ce graphe. L'atteignabilité est comme une simulation de l'exécution du système modélisé. Il permet une analyse rapide du système pour tester ses fonctionnalités.

D'autres classes d'analyse sont : l'*analyse structurelle* ; ici c'est l'analyse algébrique qui est employée ; l'*analyse d'invariant*, consiste à vérifier que certaines propriétés associées aux places sont satisfaites pour tous les états atteignables (les marquages) du système modélisé.

Les avantages de la première classe de techniques sont : la construction d'un graphe et son analyse systématique ; le graphe peut être très grand mais il existe maintenant des techniques qui permettent de travailler avec des graphes minimisés. Cependant le principal inconvénient est que, un tel graphe peut devenir très grand, même pour de petits systèmes, et donc rendre l'analyse impraticable à cause du problème d'explosion combinatoire.

Un des aspects sur lesquels nous avons contribué est la définition de la base d'utilisation conjointe des techniques et outils d'analyse. Les outils B disponibles peuvent être utilisés pour analyser les propriétés de sûreté (*safety*) des systèmes modélisés avec les réseaux de PETRI. Dans ce cadre nous avons mené diverses expérimentations ; par exemple le problème générique du Producteur-Consommateur avec sémaphore. Nous

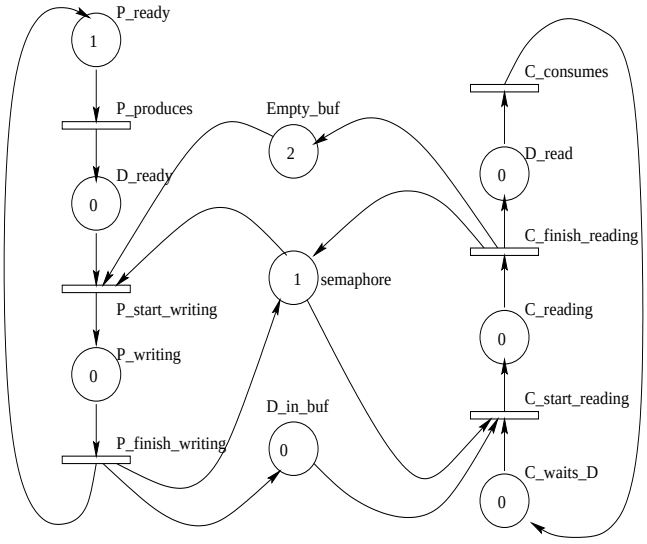

FIG. 11. Un exemple de producteur-consommateur

avons décrit et analysé le système de producteur-consommateur présenté dans la figure 11 en utilisant notre approche d'intégration de B et des réseaux de PETRI. Seule la description du système abstrait *PetriNet* est fournie ; il est inclus dans le système abstrait *EmbeddedPN* qui n'a pas changé (il modélise la sémantique des réseaux). En plus des propriétés qui peuvent être analysées avec les outils standard des réseaux de PETRI, des propriétés de sûreté qui peuvent être analysées en utilisant les outils B sont par exemple :

– la bornitude de certaines places : les places *Empty_buf* et *D_in_buf* (voir Fig. 11) sont bornées. Cela est formalisé avec le prédicat suivant qui est ajouté à l'invariant :

$$mu(Empty\_buf) \leq 2 \wedge mu(D\_in\_buf) \leq 2$$

– Il n'y a pas une mauvaise utilisation des ressources (ici le buffer) :

$$mu(Empty\_buf) + mu(D\_in\_buf) = 2$$

– Le système est vivace ; cela veut dire qu'il y a toujours au moins une transition franchissable ; ceci est formalisé par :

$$placesBefore^{\sim}[dom(nmu \rhd \{ii \mid ii \in \mathbb{N} \wedge ii > 0\})] \neq \{\}$$

Nous avons ainsi illustré comment modéliser et analyser conjointement avec les réseaux de PETRI et B.

5. Bilan de cette étude

Nous avons présenté dans ce chapitre un plongement des réseaux de PETRI en B événementiel. Le plongement est systématique et couvre aussi bien les réseaux de PETRI élémentaires que les réseaux de PETRI haut niveau. Nous avons élaboré une infrastructure à deux niveaux constituée d'un système abstrait B générique qui permet de décrire n'importe quel réseau de PETRI et, un système abstrait qui inclut (généricité) le premier et dont les événements capturent la sémantique de l'évolution des réseaux.

Concernant les réseaux de PETRI à haut niveau, plusieurs politiques ont été considérées. Concrètement nous pouvons combiner l'emploi des réseaux de PETRI et de la méthode B dans le même projet. Par exemple on peut commencer la modélisation avec un outil graphique spécifique aux réseaux de PETRI et puis poursuivre avec la méthode B pour étudier certaines propriétés.

Ce travail est lié aux travaux sur les techniques de plongement mais il est plus particulièrement lié aux travaux de SEKERINSKI et ZUROB [**SZ02**] sur les Statecharts et B.

Plus généralement ce travail contribue à combler le fossé entre la technologie des réseaux de PETRI largement utilisée en entreprises et les approches basées sur les preuves, telle que la méthode B qui utilise les machines abstraites, les systèmes abstraits, le raffinement et la preuve de théorèmes représentant des propriétés invariantes ou spécifiques. Cela constitue une étape vers une plate-forme d'analyse multifacette pour relier les techniques de modélisation des systèmes à événements discrets.

Nous poursuivons des travaux sur l'automatisation de l'interaction entre les outils dédiés aux réseaux de PETRI et ceux dédiés à B. Nous avons exploré la transformation en machine B, des sorties XML des outils tels que PEP tool[4]. Mais de nombreuses expérimentations de taille variable sont nécessaires pour la mise à l'échelle de notre processus de plongement/traduction. Des interfaces conviviales pour faciliter la combinaison des outils sont aussi nécessaires.

[4] `sourceforge.net/projects/peptool`

Troisième partie

Contributions à la méthode B

Table des Matières

CHAPITRE 5

Systèmes B communicants

Nous avons étudié et proposé une approche de composition parallèle de systèmes abstraits B [**Att05a**]. Elle a pour but de permettre une démarche ascendante, à la manière des algèbres de processus, lors de développement de systèmes concurrents ou distribués en B. Cette démarche est complémentaire à la démarche traditionnelle de B événementiel qui est descendante. Le besoin d'une approche ascendante complémentaire se justifie sur plusieurs plans : *i)* elle permet plus de souplesse dans les phases initiales, *ii)* elle facilite la structuration en sous-systèmes parallèles communicants lors des spécifications et *iii)* elle permet un interfaçage vers B pour de nombreux formalismes utilisant une approche ascendante. Nous avons montré comment spécifier sans modifier B, des systèmes communicants en explicitant les interactions entre des sous-systèmes. La communication peut se faire avec variables partagées, avec échange de messages, en mode asynchrone ou en mode synchrone. La composition parallèle produit un système abstrait B standard. Nous présentons ici quelques résultats obtenus dans le cadre de cette étude [**Att05a**].

1. Motivations

Le développement rigoureux de systèmes de grande taille requiert l'emploi de méthodes formelles, compositionnelles et outillées. Ces dernières permettent de construire des systèmes corrects à partir de sous-systèmes ou composants corrects. Dans cette étude nous proposons une approche qui permet la composition parallèle de systèmes abstraits (*abstract systems*) B [**AM98**]. Elle est justifiée sur plusieurs plans. Primo, l'approche proposée intègre un type de compositionnalité dans le cadre de la méthode B. En effet la composition est une caractéristique importante lorsqu'on développe des systèmes complexes ou de grande taille. Secondo, le développement peut ainsi être structuré avec des composants plus simples. Enfin, l'approche fournit les moyens pour interfacer avec B, de nombreux formalismes et techniques basés sur les systèmes de transitions. Ces derniers utilisent souvent des opérateurs de composition parallèle pour structurer les spécifications. En conséquence, la motivation et l'intérêt pour cette étude sont multiples comme nous l'indiquons ci-après.

La méthode B [**Abr96a**] propose un cadre pratique pour le développement formel de systèmes. Une extension dite *B événementiel* [**Abr96b, AM98**] est proposée pour traiter les systèmes distribués et plus généralement des systèmes à événements discrets[**AM98, Abr01**]. Cependant, avec cette approche orientée événement, seule une démarche descendante est privilégiée : un système donné est analysé et formalisé comme un tout. Ensuite interviennent les processus de raffinement et de décomposition [**Abr01**]. Le processus de raffinement va de l'abstrait au concret en introduisant progressivement des détails de conception. Par ailleurs, des techniques bien établies et utilisées telles que les systèmes de transitions, adoptent une approche ascendante durant l'analyse et la spécification. C'est le cas des algèbres de processus comme CCS [**Mil80**], CSP [**Hoa85**] et leurs nombreuses extensions, LOTOS [**LOT88**], AltaRica [**APGR00**], machines à configurations [**Att02c**], etc. Dans ces approches, les sous-systèmes sont identifiés, spécifiés puis composés de proche en proche pour construire des systèmes plus grands. Dans les algèbres de processus par exemple, les détails du comportement des systèmes composés peuvent être masqués dans le processus résultat qui est par conséquent plus abstrait. C'est une approche ascendante qui favorise la compositionnalité. On voudrait bien utiliser conjointement ces deux approches pour tirer profit des possibilités de raffinement offertes par

la méthode B ; cela fait partie de nos motivations. D'une part, le raffinement correct (accompagné par des preuves) jusqu'au code exécutable est un trait important pour l'ingénierie logicielle. Cependant les propriétés d'un système entier doivent être exprimées dans une spécification globale et garanties par tous ses sous-systèmes. D'autre part, la composition de parties pour en faire un tout interactif répond à un souci de pragmatisme. Notre approche est aussi motivée par le besoin de mécanismes de structuration dans les étapes initiales des spécifications en B événementiel. Ceci est utile pour l'emploi de B comme un support de développement pour des approches basées sur des systèmes communicants. Actuellement, la méthode B procède par une modélisation mathématique globale (décrite par un système abstrait) puis son raffinement et sa décomposition graduels. Elle ne permet pas de composer parallèlement des sous-systèmes. Pour franchir cet obstacle, nous étudions des moyens de composer les systèmes abstraits pour faire des systèmes interactifs plus grands. Ceci est fait sans modifier les fondements de la méthode B. Dans notre étude nous avons traité la composition de systèmes abstraits avec l'explicitation des interactions pour la communication. La communication peut se faire avec des variables partagées ou l'échange de messages entre sous-systèmes. Le mode de communication lui, peut être asynchrone ou synchrone. Deux principaux opérateurs de composition parallèle sont introduits à cet effet : l'un permet la composition parallèle avec un mode de communication asynchrone et l'emploi de variables partagées ou l'échange de messages. L'autre, permet la composition parallèle avec le mode de communication synchrone.

Ce chapitre est consacré à la composition parallèle avec variables partagées et la communication asynchrone. Dans la section 2 nous présentons notre proposition de composition de systèmes abstraits ; quelques travaux connexes et futurs sont évoqués dans la section 3. Rappelons que nous avions donné une introduction à B événementiel dans la section 2.2 du chapitre 4.

2. Systèmes abstraits B communicants (CBS)

Nous commençons par la présentation des hypothèses de travail puis nous explicitons notre approche à travers un des opérateurs de composition introduits pour structurer et faire communiquer les systèmes abstraits.

2.1. Préliminaires fondamentaux.

PROPOSITION 6. *Un système abstrait impliquant plusieurs événements coopérant pour effectuer une même tâche peut être décomposé en plusieurs sous-systèmes abstraits sur la base des variables d'état globales et locales utilisées par ces événements.*

Nous prenons l'exemple classique du producteur-consommateur pour illustrer l'approche. C'est un exemple simple mais intéressant qui est largement utilisé pour illustrer les concepts de concurrence et de synchronisation. L'idée principale est l'exécution concurrente d'un processus *consommateur* qui prélève une donnée d'une zone mémoire (tampon) à une place. Cette dernière est remplie par un autre processus *producteur*. Le consommateur ne peut pas consommer un tampon vide et le producteur ne peut pas remplir un tampon plein. Le producteur et le consommateur doivent donc alterner leurs tâches. Soit un modèle à événement du système (Fig. 1). Dans la suite nous montrons comment un tel système peut être construit en composant deux sous-systèmes distincts.

Répartition des variables et de l'invariant. Les variables et l'invariant du système abstrait *ProdCons* peuvent être répartis sur la base des variables utilisées par les événements `produce` et `consume`. Certaines variables sont communes aux deux événements. Une partie commune du système abstrait est partagée par les deux événements ; nous l'avons isolé et présenté dans la figure 2.

```
SYSTEM ProdCons
SETS
   DATA ;   STATE = {empty, full}
VARIABLES
   buffer, bufferstate, bufferc
INVARIANT
   bufferstate ∈ STATE ∧ buffer ∈ DATA ∧ bufferc ∈ DATA
INITIALIZATION
   bufferstate := empty || buffer :∈ DATA || bufferc :∈ DATA
EVENTS
   produce ≙ /* si buffer vide */
      ANY dd WHERE dd ∈ DATA ∧ bufferstate = empty
      THEN
         buffer := dd || bufferstate := full
      END ;
   consume ≙ /* si buffer plein */
      SELECT bufferstate = full
      THEN
         bufferc := buffer || bufferstate := empty
      END
END
```

FIG. 1. Un système abstrait du producteur-consommateur

```
SYSTEM ProdCons
SETS
   DATA ;   STATE = {empty, full}
VARIABLES
   buffer, bufferstate
INVARIANT
   bufferstate ∈ STATE ∧ buffer ∈ DATA
INITIALIZATION
   bufferstate := empty || buffer :∈ DATA
```

FIG. 2. Variables et invariant partagés

Cela constitue une *stratégie de répartition* qui est une hypothèse importante pour la suite de l'approche. Sur la base de cette stratégie, considérons deux sous-systèmes abstraits distincts *Producer* et *Consumer* qui seront composés parallèlement.

Nous définissons un *opérateur de composition* spécifique qui compose les systèmes abstraits de telle sorte que le résultat soit un système abstrait.

Variables partagées et substitutions multiples. La composition simultanée des substitutions généralisées ($S||T$) est initialement définie lorsque les substitutions S et T ont des espaces de variables disjoints [**Abr96a**]. Ici, pour la composition de systèmes abstraits nous avons besoin de lever cette restriction. Plusieurs auteurs ont traité ce problème [**BPR96, Dun99, Dun02**]. Nous utilisons l'extension proposée par

DUNNE [**Dun99, Dun02**]. DUNNE [**Dun99**] étend le domaine de l'opérateur de composition multiple $\|$ et le nomme *composition parallèle de substitutions*. Il ajoute la règle suivante aux règles de réécriture initiales de ABRIAL [**Abr96a**] :

$$x := E \,||\, x := F \;\;\widehat{=}\;\; E = F \Rightarrow x := E$$

DUNNE a noté aussi que lorsque les substitutions partagent le même espace de variables[1], la composition parallèle $\|$ correspond à l'opérateur plus général[2] de *fusion* de BACK et BUTLER[**BB98**]. De plus, la composition parallèle (notée $||$) de DUNNE peut avoir un nombre arbitraire de recouvrement entre les variables. Ce n'est pas le cas de l'opérateur de fusion.

Structure de travail pour les systèmes abstraits. Nous suivons l'approche présentée dans [**BPR96**] pour les machines abstraites en considérant la *signature* et le *corps* d'un système abstrait. La signature $\mathit{signature}(\mathcal{S})$ d'un système abstrait $\mathcal{S}$ est l'ensemble des identificateurs apparaissant dans sa partie statique (constantes, variables) et dans sa partie dynamique (noms des événements). Par conséquent les identificateurs sont groupés par catégorie (constantes, variables, événements). Une forme concrète d'une signature avec ces caractéristiques est :

$$\{\langle \texttt{constant}, \{\mathit{consId\_list}\}\rangle, \langle \texttt{variable}, \{\mathit{varId\_list}\}\rangle, \langle \texttt{event}, \{\mathit{evtId\_list}\}\rangle\}$$

avec *consId_list*, *varId_list* et *evtId_list* qui sont respectivement la liste des identificateurs de constantes, la liste des identificateurs de variables et la liste des identificateurs d'événements.
La signature est utile pour des raisons pratiques : c'est l'interface pour le renommage et la comparaison de systèmes. Le corps $\mathit{body}(\mathcal{S})$ est composé des variables (V), de l'invariant (I), de l'initialisation (U) et de l'ensemble des événements (E) du système abstrait.

Nous introduisons les fonctions auxilliaires $\mathit{sets}(\mathcal{S}_i)$, $\mathit{var}(\mathcal{S}_i)$, $\mathit{inv}(\mathcal{S}_i)$, $\mathit{init}(\mathcal{S}_i)$, $\mathit{events}(\mathcal{S}_i)$ pour dénoter respectivement l'ensemble des noms d'ensembles apparaissant dans la clause SETS, l'ensemble des variables, l'invariant, l'initialisation et l'ensemble des événements d'un système abstrait $\mathcal{S}_i$.
Nous simplifions les constituants[3] et la notation en considérant $\mathcal{S} = \langle \Sigma, B\rangle$ avec Σ représentant $\mathit{signature}(\mathcal{S})$ et $B = \langle V, I, U, E\rangle$ représentant $\mathit{body}(\mathcal{S}) = \langle \mathit{var}(\mathcal{S}), \mathit{inv}(\mathcal{S}), \mathit{init}(\mathcal{S}), \mathit{events}(\mathcal{S})\rangle$.

Ainsi, un système abstrait $\mathcal{S}_i = \langle \mathit{signature}(\mathcal{S}_i), \mathit{body}(\mathcal{S}_i)\rangle$ est simplement décrit par $\langle \Sigma_i, B_i\rangle$ ou de façon équivalente par $\langle \Sigma_i, \langle V_i, I_i, U_i, E_i\rangle\rangle$. De plus, pour chaque événement *ee* membre de $\mathit{events}(\mathcal{S}_i)$, $\mathit{guard}(\mathit{ee})$ dénote la garde de *ee* et $\mathit{subst}(\mathit{ee})$ dénote la substitution généralisée qui décrit l'action de *ee*.

Renommage d'un système abstrait. Le renommage d'un système abstrait $\mathcal{S} = \langle \Sigma, B\rangle$ est un remplacement syntaxique cohérent de certains identificateurs utilisés dans $\mathcal{S}$ par d'autres identificateurs donnés à cet effet. Par conséquent, le renommage est défini sur la signature Σ et étendu à B. Soient les signatures Σ_i et Σ_j (avec $\Sigma_i \subseteq \Sigma$) et soit $\alpha \in \Sigma_i \rightarrow \Sigma_j$ une application injective de telle sorte que les types des identificateurs soient préservés ; α peut être facilement étendue à B de sorte que chaque identificateur *idt* dans Σ_i utilisé dans B est remplacé par sa valeur $\alpha(\mathit{idt})$ dans $\alpha(B)$.

$$\texttt{rename}(\langle \Sigma, B\rangle, \alpha) \mathrel{\widehat{=}} \langle \alpha(\Sigma), \alpha(B)\rangle$$

Sur cette base et suivant l'approche présentée dans [**BPR96**], d'autres opérations auxilliaires peuvent être décrites sur les systèmes abstraits.

---

[1] appelé *frame* par DUNNE.

[2] défini dans le contexte général de tous les transformateurs de prédicats monotones.

[3] Nous ne considérons pas toutes les clauses d'un système abstrait mais l'extension est triviale.

Communication asynchrone et communication synchronisée. La communication implique en premier lieu l'évolution simultanée de deux ou plusieurs systèmes puis l'échange de données. Pour ce faire, nous avons besoin de mécanismes de composition et de communication pour les systèmes abstraits. Tout d'abord, les systèmes impliqués sont asynchrones ; il n'y a pas d'horloge globale. D'un point de vue pratique, une simple communication implique un récepteur et un émetteur. Deux points de vue sont généralement considérés pour les mécanismes de communication. La communication peut être synchrone ou synchronisée : c'est le paradigme de rendez-vous à la CSP (HOARE) où on considère l'acte final de communication et le fait que les systèmes impliqués sont tous arrivés au point de rendez-vous. Du point de vue de la communication asynchrone, une durée quelconque peut s'écouler entre le début de la communication (initiée par un des systèmes impliqués) et son achèvement (les autres systèmes participent). Cela veut dire que tous les systèmes ne sont pas bloqués jusqu'à l'achèvement de la communication. Dans le cadre de la méthode B, les événements sont atomiques ; leur occurrence est asynchrone et ils ne consomment pas de temps. Ils peuvent se synchroniser si l'effet des uns affecte les gardes des autres.

2.2. Composition et communication avec variables partagées. Nous commençons par la définition d'un opérateur de composition qui permet à des systèmes abstraits de communiquer en utilisant des variables partagées. L'hypothèse de travail est que cette composition doit être compatible avec l'approche descendante. On doit pouvoir, en partant du résultat de la composition, procéder au raffinement et à la décomposition [**Abr01**]. Les sous-systèmes à composer peuvent partager des variables globales *gv* et les propriétés globales associées *I(gv)*. Cependant si les sous-systèmes n'ont pas de variables en commun, la composition résulte en un entrelacement pur. Additionnellement les sous-systèmes peuvent avoir des variables locales.

Dans la suite nous utilisons $\mathcal{S}_1$ et $\mathcal{S}_2$ comme illustration (Fig. 3). La clause GLOBAL VARIABLES est utilisée ici pour indiquer quelles variables sont globales. Elle disparait dans le résultat de la composition parce que toutes les variables sont fusionnées. Notons que l'invariant I_i de $\mathcal{S}_i$ est réécrit avec les variables locales et globales de $\mathcal{S}_i$ comme : $I_i(gv) \wedge J_i(vi_j) \wedge K_i(gv, vi_j)$ avec vi_j qui représente les variables locales de $\mathcal{S}_i$.

$J_i(vi_j)$ traite des propriétés locales ; $K_i(gv, vi_j)$ relie les variables locales et globales avec leurs propriétés. Rappelons que $I_i(gv)$ est la partie commune de l'invariant partagée par les systèmes abstraits considérés. Si $K_i(gv, vi_j)$ n'est pas explicité dans un invariant donné, alors il est interpreté comme le booléen *true*.

La forme des événements dans la figure 3 est utilisée comme forme canonique des événements. La garde *guard*(ee) est composée de deux parties (prédicats). La première est exprimée avec les variables locales (liées à ANY, *bv*) et globales (*gv*) : $P_1(v1_i, bv, gv)$. La seconde est uniquement décrite avec les variables globales : $P_2(gv)$. Les gardes des événements dépendent des variables d'état. Un événement est autorisé si sa garde est vraie ; autrement, il est interdit. Un événement ee_i autorise un autre événement ee_j si l'action de ee_i contribue à rendre vraie la garde de ee_j.

Ces événements qui dépendent les uns des autres sont appelés *événements dépendants*. Par ailleurs, les *événements indépendants* sont les événements dont les gardes ne dépendent pas des actions d'autres événements.

Composition parallèle avec communication asynchrone. La communication asynchrone se rapporte la plupart du temps à la communication dans laquelle une durée arbitraire peut s'écouler entre le début de la communication initiée par un sous-système qui poursuit son évolution, et la fin de la communication où d'autres sous-systèmes achèvent la communication. C'est en opposition avec la communication synchrone par rendez-vous où les sous-systèmes s'attendent tous pour achever l'acte de communication.

La composition parallèle asynchrone de deux sous-sysèmes $\mathcal{S}_1$ et $\mathcal{S}_2$ est dénotée par $\mathcal{S}_1]|[\mathcal{S}_2$. Cette composition définit un système abstrait $\mathcal{AS}$ obtenu en calculant sa partie statique et sa partie d'événements

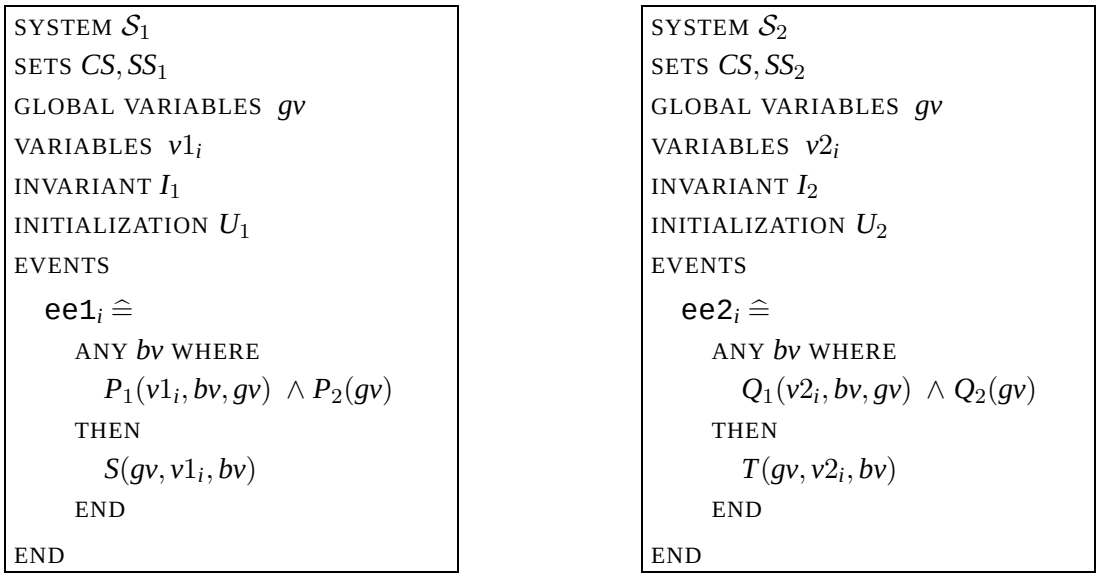

FIG. 3. Systèmes abstraits $\mathcal{S}_1$ et $\mathcal{S}_2$

en partant de celles de $\mathcal{S}_1$ et $\mathcal{S}_2$. La notation $\mathcal{AS} \mathrel{\widehat{=}} \mathcal{S}_1 ||[\, \mathcal{S}_2$ est alors utilisée. Une procédure *asynchronousMerging* est utilisée pour le calcul du résultat de la composition. La procédure est décrite par les règles d'inférence suivantes qui formalisent le calcul de chaque clause du système abstrait résultant. Pour la partie statique, la clause SETS du résultat est obtenue par la fusion (union d'ensemble) des clauses SETS des systèmes composés : $\{CS, SS_1\} \cup \{CS, SS_2\}$.
Ceci est formalisé par la règle **AsyncSetsRule**.

$$\frac{\begin{array}{c}\mathcal{AS} = \mathcal{S}_1 ||[\, \mathcal{S}_2 \\ s\mathcal{S}_1 = sets(\mathcal{S}_1) \quad s\mathcal{S}_2 = sets(\mathcal{S}_2) \quad s\mathcal{S} = s\mathcal{S}_1 \cup s\mathcal{S}_2\end{array}}{sets(\mathcal{AS}) = s\mathcal{S}}\ \textbf{AsyncSetsRule}$$

De la même façon, nous formalisons en utilisant des règles d'inférence le calcul des autres clauses du système abstrait composé. Les clauses VARIABLES de $\mathcal{S}_1$ et $\mathcal{S}_2$ sont fusionnées (avec une union disjointe) pour former les variables de $\mathcal{S}$: $\{gv, v1_i\} \cup \{gv, v2_i\}$.

L'initialisation de $\mathcal{AS}$ est définie par la fusion (avec la substitution parallèle à la DUNNE) des initialisations de $\mathcal{S}_1$ et $\mathcal{S}_2$: $U_1 \| U_2$.

L'invariant du système résultant $\mathcal{AS}$ est (par la règle **AsyncInvRule**) la conjonction des invariants de $\mathcal{S}_1$ et $\mathcal{S}_2$: $I_1 \wedge I_2$. Rappelons qu'il y a une partie commune aux deux composants. Il est nécessaire que les invariants des sous-systèmes n'expriment pas des exigences contradictoires[4].

Nous exigeons donc comme hypothèse de travail que
$\neg\, (K_1(gv, v1_j) \Rightarrow \neg\, K_2(gv, v2_j)$ et réciproquement
$\neg\, (K_2(gv, v2_j) \Rightarrow \neg\, K_1(gv, v1_j)$
ce qui se simplifie en $K_1(gv, v1_j) \wedge K_2(gv, v2_j)$.

[4]Cette hypothèse de travail est forte, en effet dans le cas général, on n'a pas cette garantie.

Nous notons $notContradict(I_1, I_2)$ pour $K_1(gv, v1_j) \wedge K_2(gv, v2_j)$.

$$\frac{\begin{array}{c}\mathcal{AS} = \mathcal{S}_1]|[\,\mathcal{S}_2 \\ I_1 = inv(\mathcal{S}_1) \quad I_2 = inv(\mathcal{S}_2) \quad notContradict(I_1, I_2) \\ i\mathcal{S} = I_1 \wedge I_2\end{array}}{inv(\mathcal{AS}) = i\mathcal{S}}\textbf{AsyncInvRule}$$

Le résultat à ce stade de la procédure est présenté dans la figure 4 (a). En ce qui converne la clause EVENTS, les événements de $\mathcal{S}_1]|[\,\mathcal{S}_2$ sont obtenus par l'union (l'opérateur $\uplus$ dénote ici l'union des ensembles d'événements) de tous les événements des systèmes abstraits $\mathcal{S}_1$ et $\mathcal{S}_2$ (**AsyncEvtRule**). En cas de conflit de noms, un renommage des sous-systèmes est fait avant leur composition.

$$\frac{\begin{array}{c}\mathcal{AS} = \mathcal{S}_1]|[\,\mathcal{S}_2 \\ events(\mathcal{S}_1) \cap events(\mathcal{S}_2) = \emptyset \\ e\mathcal{AS} = events(\mathcal{S}_1) \uplus events(\mathcal{S}_2)\end{array}}{events(\mathcal{AS}) = e\mathcal{AS}}\textbf{AsyncEvtRule}$$

Les événements de $\mathcal{S}_1$ (resp. $\mathcal{S}_2$) préservent la partie de l'invariant impliquant les variables libres utilisées dans $\mathcal{S}_1$ (resp. $\mathcal{S}_2$) à cause de la répartition des variables. Ainsi, le système abstrait résultant $\mathcal{AS}$ évolue par les transitions dénotées par les événements de $\mathcal{S}_1$ ou par les événements de $\mathcal{S}_2$. La partie événements du système résultant a la forme donnée dans la figure 4 (b).

Du point de vue opérationnel, le comportement de $\mathcal{AS}$ est un entrelacement indéterministe des événements de $\mathcal{S}_1$ et $\mathcal{S}_2$. Puisque $\mathcal{S}_1$ et $\mathcal{S}_2$ partagent des variables globales, ils ont probablement des événements dépendants. Une occurrence d'un événement est donc suivie de façon indéterministe par n'importe quel événement (de $\mathcal{S}_1$ ou de $\mathcal{S}_2$) dont la garde est vraie. Il y a un choix interne indéterministe lorsque plusieurs gardes sont vraies.

Etant donnés $\mathcal{S}_1 = \langle \Sigma_1, \langle V_1, I_1, U_1, E_1 \rangle \rangle$ et $\mathcal{S}_2 = \langle \Sigma_2, \langle V_2, I_2, U_2, E_2 \rangle \rangle$, la composition parallèle de $\mathcal{S}_1$ et $\mathcal{S}_2$ est définie comme suit :

$$\mathcal{S}_1]|[\,\mathcal{S}_2 \mathrel{\widehat{=}} \langle \Sigma_1 \cup \Sigma_2, \langle V_1 \cup V_2, I_1 \wedge I_2, U_1 \| U_2, E_1 \uplus E_2 \rangle \rangle$$

Cela formalise la procédure appelée *asynchronousMerging* qui consiste en une utilisation combinée des règles calculant chaque clause.

Par rapport au résultat de la composition, notament $\Sigma_1 \cup \Sigma_2$ et $I_1 \wedge I_2$, l'initialisation et les événements de chacun des systèmes abstraits doivent interpreter les variables locales provenant de l'autre système dans leur contexte de provenance pour éviter le *frame* problème noté ci-dessus (section 2.1). les variables communes ont les même valeurs dans les systèmes abstraits.

Soulignons que notre approche de composition est complètement intégrée dans le cadre initial de la méthode B. Nous avons temporairement introduit la composition au niveau de la spécification abstraite mais la démarche standard de B peut être poursuivie avec les raffinements successifs.

Illustration de la composition avec communication asynchrone. Soient les deux sous-systèmes producteur (*Producer*) et consommateur (*Consumer*) décrits dans la figure 5. Les variables partagées et la partie commune de l'invariant étaient données dans la figure 2.
En appliquant la procédure *asynchronousMerging* à la composition suivante :

$$AsyncProdCons \mathrel{\widehat{=}} Producer]|[\,Consumer$$

nous obtenons exactement le système abstrait décrit dans la figure 1.

SYSTEM $\mathcal{S}_1]|[\,\mathcal{S}_2$
SETS
CS, SS_1, SS_2
VARIABLES
$gv, v1_i, v2_i$
INVARIANT
$I_1 \wedge I_2$
INITIALIZATION
$U_1 \| U_2$

(a) State Part

SYSTEM $\mathcal{S}_1]|[\,\mathcal{S}_2$ (Cont'd)
...
EVENTS
$\texttt{ee1}_i \mathrel{\widehat{=}}$
ANY bv WHERE
$P_1(v1_i, bv, gv) \wedge P_2(gv)$
THEN
$S(gv, v1_i, bv)$
END
;
$\texttt{ee2}_i \mathrel{\widehat{=}}$
ANY bv WHERE
$Q_1(v2_i, bv, gv) \wedge Q_2(gv)$
THEN
$T(gv, v2_i, bv)$
END
END

(b) Event Part

FIG. 4. Système abstrait correspondant à $\mathcal{S}_1]|[\,\mathcal{S}_2$

SYSTEM *Producer*
SETS
DATA ; STATE = {empty, full}
VARIABLES
buffer, bufferstate
INVARIANT
bufferstate ∈ STATE
∧ buffer ∈ DATA
INITIALIZATION
bufferstate := empty
|| buffer :∈ DATA
EVENTS
produce ≙ /* si buffer est vide */
ANY dd WHERE
dd ∈ DATA ∧
bufferstate = empty
THEN
buffer := dd ||
bufferstate := full
END
END

SYSTEM *Consumer*
SETS
DATA ; STATE = {empty, full}
VARIABLES
buffer, bufferstate, bufferc
INVARIANT
bufferstate ∈ STATE ∧ buffer ∈ DATA
∧ bufferc ∈ DATA
INITIALIZATION
bufferstate := empty || buffer :∈ DATA
|| bufferc :∈ DATA
EVENTS
consume ≙ /* si buffer est plein */
SELECT
bufferstate = full
THEN
bufferc := buffer ||
bufferstate := empty
END
END

FIG. 5. Sous-systèmes producteur et consommateur

Conditions de cohérence. Il est nécessaire que la composition de systèmes abstraits cohérents $\mathcal{S}_1$ et $\mathcal{S}_2$ soit un autre système abstrait cohérent (voir section 2.2 du chapitre 4). Nous avons donc une obligation de preuve de cohérence pour le système abstrait composé. La composition parallèle asynchrone de systèmes abstraits est un processus d'abstraction qui construit un système abstrait. Le comportement de ce dernier est équivalent à l'observation des comportements combinés des systèmes composés. Aucune restriction particulière n'est faite pour le système résultant. Il conserve les mêmes fondements et la même sémantique que les systèmes abstraits standards. Les preuves d'obligation sont les mêmes : l'initialisation doit établir l'invariant et les événements doivent le préserver. Cependant, nous avons l'obligation de preuve suivante (ici pour deux sous-systèmes, mais cela est extensible à plusieurs sous-systèmes) pour éviter les exigences contradictoires sur les variables globales : $K_1(gv, v1_j) \wedge K_2(gv, v2_j)$. Rappelons que *gv* représente les variables globales et que $I_i(gv)$ exprime les propriétés communes.
Le système abstrait $\mathcal{AS} = \mathcal{S}_1][\mathcal{S}_2$ est alors cohérent (nous notons *consistent*$(\mathcal{AS})$) si :

- ses composants n'expriment pas des propriétés contradictoires :

$$K_1(gv, v1_j) \wedge K_2(gv, v2_j)$$

- son initialisation $U_1 \| U_2$ établit l'invariant I (avec $I = I_1(gv) \wedge J_1(v1_j) \wedge K_1(gv, v1_j) \wedge I_2(gv) \wedge J_2(v2_j) \wedge K_2(gv, v2_j)$) :

$$[U_1 \| U_2] I$$

- chaque événement *ee* membre de *events*$(\mathcal{S})$ (se termine et) préserve l'invariant I :

$$I \wedge guard(ee) \Rightarrow [subst(ee)] I$$

  pour la terminaison des *ee*, on a $I \wedge guard(ee) \Rightarrow \mathsf{term}(subst(ee))$ où le prédicat $\mathsf{term}(S)$ exprime que S termine.
  Par conséquent, l'obligation de preuve s'écrit plus simplement

$$I \wedge guard(ee) \wedge \mathsf{term}(subst(ee)) \Rightarrow [subst(ee)] I$$

Dans le cas général, ces obligations de preuve doivent être déchargées pour la composition. Notons que l'invariant I est simplifié puisque $I_1(gv) = I_2(gv)$. Cette égalité peut être assouplie en considérant que $I_1(gv) \Leftrightarrow I_2(gv)$. Aussi, $K_1(gv, v1_j) \wedge K_2(gv, v2_j)$ est établie.

Maintenant, si on considère la composition de composants déjà (montrés) cohérents, nous voulons que le résultat le soit aussi. Les conditions de cohérence sont sensiblement assouplies. Une partie de l'invariant est déjà déchargée grâce à la cohérence des composants. Aussi la composition a du sens si $K_1(gv, v1_j) \wedge K_2(gv, v2_j)$.

Par conséquent, le système abstrait $\mathcal{AS} = \mathcal{S}_1][\mathcal{S}_2$ est cohérent si :

- son initialisation $U_1 \| U_2$ établit l'invariant $I = I_1(gv) \wedge J_1(v1_j) \wedge K_1(gv, v1_j) \wedge I_2(gv) \wedge J_2(v2_j) \wedge K_2(gv, v2_j)$:

$$[U_1 \| U_2] I$$

  $[U_1 \| U_2] I$ est vrai par définition de $\|$ et aussi parce que U_1 (resp. U_2) établit déjà $I_1(gv) \wedge J_1(v1_j) \wedge K_1(gv, v1_j)$ (resp. $I_2(gv) \wedge J_2(v2_j) \wedge K_2(gv, v2_j)$).
- chaque événement *ee* membre de *events*$(\mathcal{S})$ termine et préserve l'invariant I :

$$I \wedge guard(ee) \wedge \mathsf{term}(subst(ee)) \Rightarrow [subst(ee)] I$$

Sur la compositionnalité. La composition de deux systèmes abstraits cohérents $\mathcal{S}_1$ et $\mathcal{S}_2$ est cohérente si :

- soit les systèmes ne décrivent aucune relation entre les variables locales et les variables globales, cela veut dire que $K_k(gv, vk_i)$ est réduit à *true* (c'est le cas avec l'exemple du producteur-consommateur),

– soit on effectue la preuve que les $K_k(gv, vk_i)$ ne sont pas contradictoires. Or, c'est le cas par construction de la composition (voir la règle *AsyncInvRule*).

$$\frac{\mathit{consistent}(\mathcal{S}_1) \quad \mathit{consistent}(\mathcal{S}_2) \quad \mathit{notContradict}(\mathit{inv}(\mathcal{S}_1), \mathit{inv}(\mathcal{S}_2)) \quad \mathcal{AS} = \mathcal{S}_1]|[\,\mathcal{S}_2}{\mathit{consistent}(\mathcal{AS})}$$

Preuve. Étant donné un événement ee_1 de $\mathcal{S}_1 = \langle \Sigma_1, \langle V_1, I_1, U_1, E_1 \rangle \rangle$,

$$I_1 \wedge \mathit{guard}(ee_1) \wedge \mathsf{term}(\mathit{subst}(ee_1)) \Rightarrow [\mathit{subst}(ee_1)](I_1) \quad \text{car } \mathit{consistent}(\mathcal{S}_1)$$

$\Leftrightarrow I_1(gv) \wedge J_1(v1_i) \wedge K_1(gv, vk_1) \wedge P_1(v1_i, bv) \wedge P_2(gv) \Rightarrow [\mathit{subst}(ee_1)](I_1(gv) \wedge J_1(v1_i)) \wedge K_1(gv, vk_1)$
en dépliant I_1 et $\mathit{guard}(ee_1)$

$\Leftrightarrow I_1(gv) \wedge J_1(v1_i) \wedge K_1(gv, vk_1) \wedge I_2 \wedge P_1(v1_i, bv) \wedge P_2(gv) \Rightarrow [\mathit{subst}(ee_1)](I_1(gv) \wedge J_1(v1_i) \wedge I_2)$
introduction de I_2 car $\mathit{subst}(ee_1)$ ne modifie éventuellement que $J_1(v1_i)$ et $I_1(gv)$ est égal à $I_2(gv)$.

$\Leftrightarrow I_1 \wedge I_2 \wedge \mathit{guard}(ee_1) \Rightarrow [\mathit{subst}(ee_1)](I_1 \wedge I_2)$
simplification avec l'hypothèse $\mathit{notContradict}(\mathit{inv}(\mathcal{S}_1), \mathit{inv}(\mathcal{S}_2))$

Ainsi ee_1 préserve l'invariant global □

De la même manière, ee_2, (pour les événements de $\mathcal{S}_2$), préserve l'invariant global.

Notons cependant que ce résultat dépend d'une hypothèse forte (en prémisse de la règle *AsyncInvRule*) qui est la non-contradiction entre les $K_k(gv, vk_i)$. C'est l'hypothèse que chaque (sous-)système fait à propos de l'autre (sous-)système.
Dans le cas général, il n'est pas possible d'établir la non-contradiction entre les invariants $K_k(gv, vk_i)$; et donc les preuves de préservation de l'invariant global doivent être refaites dans ce cas général.

Propriétés algébriques de la composition.

$$\mathcal{S}_1]|[\,\mathcal{S}_2 \equiv \mathcal{S}_2]|[\,\mathcal{S}_1$$

$$(\mathcal{S}_1]|[\,\mathcal{S}_2)]|[\,\mathcal{S}_3 \equiv \mathcal{S}_1]|[\,(\mathcal{S}_2]|[\,\mathcal{S}_3)$$

La composition parallèle est commutative et associative. En effet, premièrement l'invariant de la composition est la conjonction des invariants des composants ; deuxièmement, la composition parallèle de substitutions $\|$ est utilisée pour l'initialisation ; finalement, en ce qui concerne les événements, la composition résulte en un ensemble d'événements.

Grâce à ces deux propriétés, la composition parallèle d'un ensemble fini de systèmes abstraits $\mathcal{S}_i$ est écrite $]|[_{i\in 1\cdots n}\mathcal{S}_i$. Le résultat est l'application successive (par paire) de $]|[$. Par conséquent la notation est généralisée comme suit :

$$]|[_{i\in 1\cdots n}\mathcal{S}_i \mathrel{\widehat{=}} \langle \cup_{i\in 1\cdots n}\Sigma_i,\ \langle \cup_{i\in 1\cdots n}V_i,\ \wedge_{i\in 1\cdots n}\, I_i,\ \|_{i\in 1\cdots n}U_i,\ \biguplus_{i\in 1\cdots n} E_i\rangle\rangle$$

Raffinement. Bien que nous nous focalisons dans cette section sur la composition au niveau de la spécification, il est utile d'évoquer les aspects liés au raffinement. La composition peut être effectuée pourvu que les sous-systèmes aient toujours des variables communes, déjà raffinées ou non. Raffiner un système abstrait consiste à raffiner son état et ses événements. Le raffinement de l'état consiste à introduire des variables concrètes liées aux abstraites avec un invariant. Le raffinement des événements consiste à *i)* renforcer les

gardes des événements et *ii)* à introduire de nouveaux événements qui raffinent la substitution *skip*. Un des avantages de notre approche est qu'elle préserve ce processus de raffinement standard de B. Le raffinement $\mathcal{S}'$ d'une composition $\mathcal{S}_1]|[\,\mathcal{S}_2$ est le raffinement du système abstrait systématiquement calculé à partir de $\mathcal{S}_1$ et $\mathcal{S}_2$ en utilisant la procédure *asynchronousMerging*. De plus, sous certaines conditions décrites ci-après, la composition est monotone par rapport au raffinement (l'invariant de liaison reste inchangé). Cependant, la composition de systèmes raffinés a du sens si : *i)* le raffinement des variables locales abstraites n'introduit pas de conflits avec les variables concrètes ; *ii)* soit les variables partagées ne sont pas raffinées dans l'étape courante soit elles le sont de la même façon dans les systèmes concernés.

$$\frac{\mathcal{S}_1{}' \sqsubseteq \mathcal{S}_1 \qquad \mathcal{S}_2{}' \sqsubseteq \mathcal{S}_2}{\mathcal{S}_1{}']|[\,\mathcal{S}_2{}' \;\sqsubseteq\; \mathcal{S}_1]|[\,\mathcal{S}_2}$$

En effet, les nouveaux événements raffinent toujours *skip* ; les variables concrètes introduites sont liées aux variables abstraites dans l'invariant final ; la partie commune de l'invariant est raffinée de la même façon. En ce qui concerne le renforcement des gardes, un événement concret de $\mathcal{S}_i{}'$ qui raffine un événement de $\mathcal{S}_i$, raffine toujours le même événement abstrait dans la composition.

3. Discussions

Nous avons présenté une technique générale pour structurer des systèmes abstraits interactifs en les composant parallèlement. Notre étude [**Att02b**] traite plus généralement les cas de variables partagées et d'échange de messages mais seul le premier cas est présenté ici. Avant tout, notre étude conserve le cadre théorique de la méthode B. Ainsi, le résultat d'une composition parallèle est un autre système abstrait standard. Nous évoquons ci-après quelques travaux connexes.

D'abord, considérons le sujet général de la composition de systèmes. L'approche *Assumption-Commitment* (aussi appelée *Rely-Guarantee*) [**Jon83, XS98, XdRH97**] a été proposée pour la composition de systèmes concurrents partageant des variables d'état. Brièvement, elle consiste pour chaque système impliqué dans la composition, à établir les propriétés de correction en faisant des hypothèses sur les autres systèmes qui forment son environnement. Par conséquent, la conception des systèmes composants n'est par réellement indépendante et cela rend difficile la structuration des systèmes. Cependant la propriété de compositionalité est intéressante pour la preuve de propriétés du système global. En effet les composants du système peuvent être prouvés séparément puis composés sans avoir à prouver de nouveau tout le système. L'approche *Assumption-Commitment* ne permet pas le raffinement séparé. Notre approche de composition n'impose pas aux sous-systèmes de raisonner sur leur environnement. Ils sont indépendants mais des propriétés de correction sont traités par les obligations de preuve pendant la composition. Cela simplifie largement la structuration du système global et aussi le raffinement séparé des sous-systèmes. En effet l'interférence entre variables globales est considérée uniquement pendant la composition.

Dans [**But96**], BUTLER traite le raffinement de systèmes à actions communicants. Nous partageons l'approche compositionnelle avec son travail. Mais la principale différence est que son approche est basée sur la communication avec les noms d'actions identiques (occurrences des actions de mêmes noms) alors que nous utilisons des variables partagées et des gardes.

En ce qui concerne la méthode B, il y a d'autres travaux sur la composition et l'interaction entre spécifications. Le travail présenté dans [**ST02**] a les mêmes motivations que le nôtre sur l'interaction entre systèmes. Cependant, cette approche n'est pas vraiment comparable à la nôtre. La démarche des auteurs est différente : ils utilisent les machines B et ils leur ajoutent une *partie contrôle* avec des processus contrôleurs écrits en CSP [**Hoa85**]. Les deux parties B et CSP étant développés séparément. Dans notre approche le contrôle est directement intégré dans les gardes des événements de systèmes abstraits B. Notre approche

semble plus intéressante et supporte mieux le passage à l'échelle ainsi que l'outillage nécessaire pour analyser les spécifications.

Dans [**BJK02**], les auteurs définissent une composition parallèle de systèmes abstraits B. Leur composition couvre les phases de spécification et de raffinement. Cependant, des différences avec notre travail peuvent être soulignées. L'approche courante ne traite que le cas des communications synchronisées. Un prédicat est explicitement ajouté pour contraindre la synchronisation entre événements : c'est une forme de garde sur les événements (en utilisant leurs noms). De plus, la compositionnalité du raffinement est restreinte à un nombre fini de systèmes qui sont redéfinis par des systèmes de transitions étiquetées. Dans notre approche il n'y a aucun mécanisme additionnel et aucune restriction n'est imposée, c'est l'intérêt de rester dans le cadre standard de B.

Dans [**PS02**], les auteurs traitent aussi l'interaction entre systèmes mais contrairement à notre approche, les auteurs utilisent des machines à états [**SDGS98**] qui partagent (et communiquent ainsi) des événements. Les machines à états participant à l'interaction, évoluent ensemble lorsqu'elles sont en mesure d'effectuer le même événement. Cette approche diffère de la notre : elle adopte une approche de conception spécifique des sous-systèmes où un événement est partiellement spécifié de façon distribuée dans plusieurs machines à états. Les machines sont ensuite traduites en des systèmes B particuliers qui ne rentrent pas dans le cadre actuel de B événementiel. Par conséquent, cette proposition a certaines limitations ; par exemple le raffinement tel que défini dans la méthode B ne peut pas être employé pour le reste du développement des machines ; il faudra un autre cadre approprié au raffinement.

Travaux en cours et futurs. Outre le développement d'outils en amont des outils standard de B, nous poursuivons actuellement d'autres études de cas pour valider notre technique d'échange de messages et des cas de communication particulière. D'autre mécanismes de structuration peuvent être simplement introduits sur la base de ce qui est présenté. Par exemple la communication de groupe est un modèle intéressant pour certains types d'applications [**Att06a**].

CHAPITRE 6

Méthode de spécification de systèmes multi-processus

1. Motivations

La spécification de systèmes asynchrones est difficile. Cette difficulté est accrue lorsqu'il s'agit de systèmes avec une structure ou architecture variable ou architecture dynamique. La bonne structuration de la spécification d'un système est importante non seulement pour la lisibilité mais également pour l'analyse formelle, le développement et la maintenance. La méthode B [**Abr96b, AM98**] est une des méthodes bien outillées qui couvre le processus de développement allant de la spécification à la génération de codes exécutables. Cependant il n'y a pas de guides spécifiques au niveau du travail de spécification. Il s'agit là d'une limitation qui est aussi partagée par les méthodes formelles généralistes comme PVS, Coq et Isabelle.

La motivation de cette partie de notre travail est la recherche de méthodes, techniques et outils pratiques pour aider les développeurs à spécifier et analyser leurs systèmes ; nous mettons l'accent sur les méthodes appropriées à des classes de systèmes à traiter.

La conception, l'analyse et l'implantation de systèmes distribués sont des tâches d'ingénierie difficiles. Ces tâches posent encore des problèmes difficiles (spécification et analyse). Pour maîtriser cette difficulté plusieurs couches de préoccupations sont distinguées. La couche la plus basse est celle des réseaux et des systèmes d'exploitation ; par dessus cette couche, on construit une couche de communication de groupe. Finalement la couche application est celle la plus proche de l'utilisateur.

La sûreté des applications distribuées requiert la preuve de correction à différents niveaux d'où la nécessité d'une *approche de vérification de propriétés en couches*. Pour contribuer à une telle approche nous avons entrepris d'élaborer une méthode de spécification et de vérification qui peut aider aussi bien dans la couche basse des applications distribuées que dans les applications distribuées elles mêmes.

Dans ce chapitre nous nous concentrons sur la spécification de systèmes asynchrones multi-processus avec une architecture dynamique. Nous présentons plus particulièrement la méthode que nous avons élaborée pour spécifier et analyser de manière systématique les systèmes en question. Nous préconisons la combinaison des techniques de preuve de théorème et d'évaluation de modèles pour effectuer l'*analyse multifacette* des systèmes. Les résultats de ces travaux sont publiés dans [**Att06a, Att06c**].

2. Spécification et analyse en B de systèmes de communication de groupes

L'interaction entre plusieurs processus qui coopèrent pour effectuer une tâche commune est souvent gérée en utilisant la communication entre les processus. Deux paradigmes principaux sous-tendent la communication : l'échange de messages et le partage de variables. Le premier est adapté aux systèmes distribués et par conséquent utilisés pour les systèmes de communication de groupe. L'architecture d'un système de communication de groupe est dynamique. Les processus impliqués dans un groupe interagissent en utilisant une structuration ad hoc des processus. Cette structuration varie pendant l'évolution des processus. Nous nous préoccupons ici de la spécification systématique des processus interagissants (communicants).

2.1. Architecture dynamique de processus. Les algèbres de processus (CCS [**Mil89**], CSP[**Ros98**], LOTOS[**LOT88**], etc) généralisent les approches de transition entre états et sont largement utilisés pour modéliser les systèmes communicants. Ici les comportements des processus élémentaires sont d'abord décrits

puis les opérateurs de composition parallèle sont utilisés pour combiner de proche en proche les processus. Ainsi l'architecture d'un système est une composition statique d'un nombre fini de processus. Le $\pi-$calculus de MILNER [**MPW92**] permet la description de structures dynamiques de processus mais il n'est pas bien outillé.

Dans plusieurs situations de spécification, on a à se préoccuper de la configuration dynamique de l'architecture des systèmes : par exemple un système d'allocation de ressources où le nombre de processus est variable. De telles situations sont souvent traitées en considérant un nombre arbitrairement élevé de processus ; il s'agit là d'une solution biaisée.

Nous avons proposé une démarche qui combine une approche orientée algèbre de processus et systèmes abstraits B. Elle convient bien à la spécification de processus communicants dynamiquement et pallie les limitations des deux approches considérées séparément.

Approche des transitions à états. La prise en compte du comportement d'un processus est intuitif mais les systèmes de transition manquent de structures de haut niveau pour les systèmes complexes. La gestion d'un nombre variable de processus n'est pas possible. Il n'est pas non plus possible de prendre en compte plusieurs instances du même processus. Les synchronisations entre processus doivent être explicites.

Approche des Systèmes B. Un point préoccupant est celui de la complétude de l'ordonnancement des événements d'une spécification : est-ce que la spécification couvre toutes les évolutions possibles (séquences d'événements) exprimées dans le cahier de charges ? En effet on peut avoir une spécification cohérente par rapport à l'invariant élaboré, mais qui ne soit pas conforme aux exigences comportementales attendues.

Des guides rigoureux peuvent aider à découvrir et exprimer les comportements désirés ; les propriétés de vivacité aident à couvrir l'aspect relevant de la complétude ; c'est la base de notre proposition.

2.2. Ordonnancement d'événements : causalité et vivacité. Nous expliquons ici notre stratégie pour contraindre les occurrences d'événements par rapport aux exigences du développeur. Nous explicitons l'ordonnancement des événements. Cela nous permet d'exprimer la causalité entre événements. Mais la propriété de vivacité *quelque chose de bien arrivera* peut aussi s'exprimer en considérant l'ordonnancement des événements. Puisque nous ne voulons pas modifier la méthode B et sa sémantique dans notre approche pour obtenir les modèles B pour les systèmes multiprocesseurs, nous évitons l'usage de la logique temporelle pour exprimer les propriétés de vivacité et vérifier (comme c'est le cas habituellement) les modèles obtenus par rapport à ces propriétés.

Par conséquent, au lieu de propriétés de vivacité exprimées en logique modale, nous considérons des causalités qui sont plus intuitives au niveau de la modélisation. Il est plus facile au développeur d'exprimer l'ordonnancement des événements de son système comme une relation de causalité ; par exemple une exigence telle que : "l'événement e_1 précède toujours l'événement e_2" est simplement capturée par une causalité : e_2 suit e_1. Implicitement chaque événement de l'alphabet du système en étude doit éventuellement pouvoir se produire (vivacité).

En utilisant une formule LTL (*Linear Temporal Logic*), le même exemple est exprimé comme suit :

$$\Box(e_1 \rightarrow \Diamond e_2)$$

Par conséquent les exigences du développeur relatives à l'ordonnancement des événements peuvent facilement être exprimées avec une collection de causalités entre événements comme ci-dessus. C'est le principe que nous utilisons dans notre proposition : le développeur explicite l'ordonnancement des événements de son système.

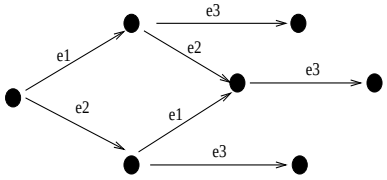

FIG. 1. Exemple de causalité

Techniquement, en considérant le comportement d'un processus, l'analyse des propriétés de vivacité ne peut être effectuée que sur des traces infinies (à partir de l'exploration du graphe de comportement du processus) ; c'est à dire en utilisant un automate de BÜCHI.

D'un point de vue méthodologique, nous utilisons un graphe orienté pour décrire les comportements des processus. Le graphe est décrit avec une relation sur les événements. Cette relation est appelée *follow* dans la suite et elle décrit une relation de transition ; les transitions sont ici étiquetées avec des événements. La relation *follow* est donnée par le spécifieur pour capturer les ordonnancements d'événements souhaités. La relation *follow* est suffisante pour décrire des situations compliquées ; par exemple une exigence telle que : "un événement e_3 est causé par la disjonction de e_1 et e_2" est modélisée par la relation $\{(e_1, \{e_2, e_3\}), (e_2, \{e_1, e_3\})\}$ qui est graphiquement présentée dans la figure 1.

Par rapport à un système abstrait B à construire, les exigences capturées par les ordonnancements d'événements (avec la relation *follow*) doivent apparaître dans les gardes des événements, ainsi le système abstrait sera correctement construit.

Traces des systèmes abstraits. On voit l'évolution d'un système abstrait comme la trace des enchaînements de ses événements à partir de l'état initial (qui est obligatoire pour un système abstrait) : un ensemble de séquences d'événements. En toute rigueur ces séquences peuvent être infinies. Comme dans le cas des automates de BÜCHI où les séquences infinies sont reconnues par rapport aux états finaux, on peut gérer les séquences infinies en considérant des états finaux des systèmes abstraits (ou des événements y conduisant) ; nous allons nous limiter à des séquences finies du fait des interactions avec l'utilisateur.

2.3. La méthode de spécification proposée. De façon générale il s'agit, étant donné un cahier de charges (un ensemble d'exigences utilisateur), de construire un système abstrait $\mathcal{A}$ qui modélise le comportement d'un ensemble de processus communicants $\mathcal{P}_r$. Ce comportement est décrit avec un ensemble d'événements qui sont observés quand le système évolue. Les systèmes communicants peuvent être de différents *types* ; ils peuvent utiliser des ressources ou données partagées ou non.

Par *type* de processus nous entendons la caractérisation d'un ensemble de processus ayant le même comportement ; par exemple dans un système producteur-consommateur, on peut avoir plusieurs processus producteurs (plusieurs instances d'un même comportement) et plusieurs processus consommateurs ; on dira du point de vue de la modélisation qu'il y a un type Producteur (qui décrit le comportement de tous les processus producteurs) et un type Consommateur.

La méthode proposée est une approche méthodologique pour construire le comportement du système envisagé, de telle sorte qu'il corresponde aux exigences énoncées dans le cahier de charges.

En conséquence le système construit doit garantir la sûreté et les ordonnancements d'événements doivent être ceux attendus. Nous distinguons plusieurs étapes :

Étape 1. Schéma général du système abstrait

• Commencer la construction d'un système abstrait $\mathcal{A}$ qui est le modèle formel du système à étudier. La sémantique de $\mathcal{A}$ n'est pas modifiée. $\mathcal{A}$ est constitué de *S*, *E*, *follow*, *Evts* où *S* est l'espace d'états qui sera décrit avec des variables et des prédicats, *E* est un alphabet d'événements, *follow* est une relation de transition sur *E* et *Evts* est un ensemble de spécifications d'événements.

La relation *follow* n'est ni réflexive, ni symétrique, ni transitive. De plus lorsque $(e_1, e_2) \in$ *follow*, chaque occurrence de e_2 doit être (immédiatement ou non, selon les autres éléments de *follow*) précédée d'une occurrence de e_1. Cette contrainte d'évolution est introduite plus tard (**Étape 5.**) à l'aide de garde d'événements.

Puisque $\mathcal{A}$ est un système multi-processus, plusieurs types de processus contribueront à définir son comportement. Ces types de processus sont décrits dans la suite.

• Identifier l'ensemble $\mathcal{P}_r$ des types de processus qui interagissent dans $\mathcal{A}$: $\mathcal{P}_r = \{P_1, P_2, \cdots\}$. Inclure dans la clause SETS de $\mathcal{A}$ un ensemble abstrait P_i pour chaque type de processus.

• Pour chaque type de processus $P \in \mathcal{P}_r$:

– considérer une nouvelle variable pour modéliser l'ensemble des processus de ce type ;

– Identifier les ressources et données du système et distinguer celles qui sont partagées ;

– Inclure dans la clause SETS de $\mathcal{A}$ les ensembles abstraits identifiés.

• Les ressources partagées nécessitent une stratégie particulière ;

Pour chaque type de ressources partagées :

– définir un événement B pour accéder/prendre/libérer la ressource. Typiquement un événement qui lit une ressource/donnée commune doit être différencié d'un événement qui écrit la donnée.

• Identifier l'ensemble des événements qui font évoluer le système : *E*. Ces événements peuvent être subdivisés en deux classes d'événements :

les *événements généraux* (GE) qui affectent le système entier et les *événements "process-specific"* (SE) qui correspondent à l'évolution des processus identifiés.

Cette répartition des événements en classes favorise la décomposition du système abstrait puisque les événements peuvent être répartis/partagés entre des sous-systèmes.

Étape 2. Modélisation de l'espace d'états : *S*

• Identifier les ressources et propriétés globales avec un prédicat (invariant) qui caractérise *S*.

• Compléter les clauses VARIABLES et INVARIANT de $\mathcal{A}$ avec les déclarations de ressources et les propriétés identifiées.

Étape 3. Description de l'ordonnancement des événements : la relation *follow*

• Identifier les propriétés du comportement du système global ; c'est-à-dire un ordonnancement spécifique des occurrences des événements en fonction des exigences (de vivacité) du cahier de charges.

La relation *follow* : $E \leftrightarrow E$ (voir Étape 1) capture l'ordonnancement requis pour les événements. Il reste à compléter *follow* en fonction des classes (GE, SE) d'événements dans *E*.

Étape 4. Définition des événements généraux : $Evts_{GE}$

• Compléter $\mathcal{A}$ avec les spécifications B des événements généraux.

Étape 5. Définition des comportements spécifiques aux processus : $Evts_{SE}$

• Pour chaque type de processus $P \in \mathcal{P}_r$, on complète $\mathcal{A}$ avec $\mathcal{A}_P$ qui est constitué de S_P, E_P, $follow_P$, $Evts_P$.

(1) E_P : identifier le sous-ensemble d'événements de *SE* qui soutient l'évolution de P ;

(2) $follow_P$: définir une relation d'ordonnancement sur ces événements en considérant les exigences spécifiques sur P ; cela consiste à décrire un sous-ensemble de *follow* relatif à P : $follow_P$. L'emploi d'une machine à états plutôt réduite est conseillé ici ; elle représente un graphe de comportement où les transitions sont étiquetées par des noms d'événements.

L'alphabet d'événements des processus doit être plus précisément défini que précédemment ; une analyse approfondie du comportement d'un processus est nécessaire pour découvrir l'alphabet d'événements des types de processus.

En ce qui concerne les processus du même type évoluant dans le système, tous les processus ayant atteint un même stade d'évolution (correspondant à un état du graphe de comportement) peuvent continuer à se dérouler de la même manière mais de façon non déterministe ; en effet, seul un des processus partageant le même état poursuit le déroulement. Cette situation est prise en compte avec des variables dénotant des ensembles de processus qui ont atteint un même stage ; les gardes des événements sont alors exprimées avec ces variables puis les corps des événements sont modifiés en conséquence (par exemple s'assurer que selon la spécification, le processus qui a quitté l'état en question, n'y est plus).

Par exemple une variable *siprocesses* dénote l'ensemble des processus dans un certain état s_i ;

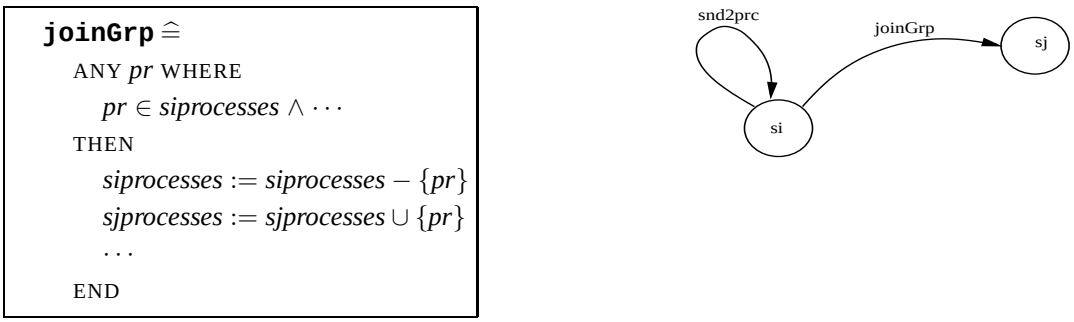

FIG. 2. Description des événements des processus à partir d'un état

Il s'en suit que seuls les processus dans l'état s_i peuvent évoluer (ou se comporter) comme décrit par les événements associés à cet état.

(3) $Evts_P$:

– Décrire chaque événement du processus courant P comme un événement B (garde et substitution).

– Compléter la clause EVENTS de $\mathcal{A}$ avec le nouvel événement décrit.

Les événements de l'alphabet E_P sont aussi décrits de la même façon.

Après l'étape courante, le système abstrait $\mathcal{A}$ en construction est garni avec les spécifications B de tous les événements.

Étape 6. Cohérence

• Compléter le système abstrait $\mathcal{A}$ avec les propriétés voulues ; compléter la clause INVARIANT avec les

prédicats associés puis
• prouver sa cohérence en utilisant des prouveurs B tels que AtelierB.

On doit se décharger de toutes les obligations. Cependant on n'a pas encore le moyen de garantir les exigences de vivacité capturées dans dans la relation *follow* ; On doit prouver que *traces*($\mathcal{A}$) coïncide avec la relation *follow*. C'est le rôle de l'étape suivante.

Étape 7. Complétude et vivacité
• Analyser et améliorer le système abstrait $\mathcal{A}$; ceci est fait à l'aide de l'évaluation de modèle (*model checking*) et d'animation (avec ProB par exemple).

D'abord il faut évaluer le système abstrait $\mathcal{A}$ pour détecter des blocages (*deadlocks*). Corriger la spécification en conséquence.

• Lorsque $\mathcal{A}$ est sans blocage, vérifier qu'il satisfait les exigences consignées dans *follow*. En utilisant ProB par exemple, il faut vérifier que tous les événements peuvent être déclenchés par rapport aux contraintes dans *follow*. Avec ProB, lorsqu'on sélectionne un événement, on a la liste des événements qui peuvent le suivre (ce sont les événemnts suivants dans le graphe de comportement).

De plus, il faut vérifier que chaque événement (*evt*) autorise les événements dans *follow*(*evt*). Cela est effectué en visualisant le graphe de comportement (par exemple ProB permet de visualiser un graphe réduit reduced visited states). Une autre façon de faire est par des animations successives. A partir d'un événement, on doit s'assurer que tous les autres événements suivants correspondent bien à ce qui est prévu dans *follow*. Le système abstrait $\mathcal{A}$ est mis à jour en conséquence.

Puisque la relation *follow* exprime l'ensemble de tous les ordonnancements d'événements possibles, après l'étape **Étape 7.**, l'analyse du comportement logique est achevée et on obtient une spécification *correcte* de $\mathcal{A}$ par rapport aux besoins.

Preuve : les occurrences des ordonnancements d'événements de $\mathcal{A}$ sont testées par rapport à *follow*.

Formellement, l'occurrence d'un événement $e_1 \mathrel{\widehat{=}} eG_1 \Longrightarrow eB_1$ est :

$$\exists\, v_i, v_{i+1}.[v := v_i]eG_1 \wedge [v, v' := v_i, v_{i+1}]\mathsf{prd}_v(eB_1)$$

L'occurrence d'une séquence de deux événements $e_1.e_2$ est :

$$\exists\, v_i, v_{i+1}, v_{i+2}.[v := v_i]eG_1 \wedge [v, v' := v_i, v_{i+1}]\mathsf{prd}_v(eB_1) \wedge$$
$$[v := v_{i+1}]eG_2 \wedge [v, v' := v_{i+1}, v_{i+2}]\mathsf{prd}_v(eB_2)$$

Ce qui se généralise facilement aux séquences de *n* événements et correspond à *traces*($\mathcal{A}$) ; notons que dans le cas général, les séquences devraient être infinies et non limitées à *n* événements.

$\mathsf{prd}_v(S)$ est le prédicat *before-after* de la substitution S ; il relie les valeurs de la variable d'état juste avant (*v*) et après (*v*') la substitution *S*. Nous notons $follow^{\circledast}$ la couverture de *follow*, c'est-à-dire toutes les possibilités de succession d'événements autorisées à travers la relation *follow* ; cette couverture donne les séquences des occurrences d'événements qui correspondent aux exigences consignées dans *follow*. En conséquence de ce qui précède on a

$$\boxed{follow^{\circledast} = traces(\mathcal{A})}$$

Cette relation est très contraignante ; dans la pratique, nous devons nous limiter à

$$\boxed{traces(\mathcal{A}) \subseteq follow^{\circledast}}$$

car toutes les séquences autorisées par *follow* ne sont pas nécessairement observées pour les évolution de $\mathcal{A}$.

Nous avons complété la méthode par des règles pratiques pour améliorer la description des comportements des processus.

La méthode est utilisée sur plusieurs études de cas et notamment pour la spécification et la vérification de système de communication de groupe [**Att06a**].

Les systèmes de communication de groupe (*Group Communication Systems* [**Edi96, CKV01**]) constituent la référence en ce qui concerne les couches basses dans les applications réparties. Nous avons alors utilisé les GCS comme banc de test de notre méthode. Le GCS concentre des difficultés importantes en terme de modélisation d'analyse et de mise en œuvre.

En somme notre proposition consiste à construire progressivement un système abstrait $\mathcal{A}$ tel que :

$$\mathcal{A} = \bigoplus_i \langle S_i,\, E_i,\, follow_i,\, Evts_i \rangle$$

où $\bigoplus$ représente l'opération de *fusion synchrone* des systèmes abstraits. Cette opération est plus générale comparée à l'opération de composition parallèle $]\!|\!|[$ que nous avions élaborée (section 2, chapitre 5) pour la composition parallèle. En effet avec l'opérateur $\bigoplus$ nous prenons en compte non seulement l'interaction entre les sous-systèmes mais aussi les contraintes de causalité liées aux sous-systèmes.

2.4. Synthèse. La méthode consiste à fusionner des processus types. L'interaction entre les types de processus est gérée avec des événements communs, l'accès à des ressources communes en utilisant des événements et les événements dépendants (ces derniers sont reliés par leur gardes et leur événements).

Construction progressive des systèmes multi-processus. La construction d'un système $\mathcal{A}$ commence avec sa partie commune $\mathcal{A}_g$ qui est identifiée dans le cahier de charges : $\mathcal{A}_g = \langle S_g,\, E_g,\, follow_g,\, Evts_g \rangle$. Cette phase est couverte par les **Etape 1** à **Etape 4** ci-dessus.

Ensuite, nous poursuivons avec l'**Etape 5** où on effectue la fusion des processus types.

$$\mathcal{A} = \mathcal{A}_g \Uparrow \bigoplus_i \langle S_i,\, E_i,\, follow_i,\, Evts_i \rangle$$

L'operateur $\Uparrow$ dénote la fusion de deux tuples ; $\bigoplus$ dénote la *fusion synchrone* des spécifications de processus types ; la fusion est synchrone parce que des variables et événements sont partagés et forcent la communication des processus.

La fusion résulte en une union disjointe, composant à composant, des éléments des tuples.

Hiérarchie. Chaque processus peut également être défini progressivement avec la même démarche de fusion.

On peut ainsi avoir un système $\mathcal{A}$ avec

$$\mathcal{A} = \mathcal{A}_g \Uparrow \bigoplus (P_1\, P_2\, \cdots P_n)$$

où certains P_u sont tels que $P_u = \langle S_u,\, E_u,\, follow_u,\, Evts_u \rangle$ et d'autres $P_{k \neq u} = \bigoplus_i \langle S_i,\, E_i,\, follow_i,\, Evts_i \rangle$

Analyse formelle. Il y a des obligations de preuve de cohérence exprimées à travers *Event B* ; c'est ce qui est fait dans l'étape **Etape 6**. Il y a d'autres obligations de preuve pour établir la correction par rapport aux exigences comportementales : le modèle construit doit être conforme aux comportements exprimés ; c'est la charge de l'**Etape 7** où on prouve

$$traces(\mathcal{A}) \subseteq follow^{\circledast}$$

3. Synthèse sur cette étude et perspectives

Ces travaux ont abouti à des résultats qui ont été publiés dans [**Att06a**], [**Att06c**]. Ils généralisent des travaux antérieurs sur la composition de systèmes abstraits [**Att05a**]. Nous nous appuyons sur ces travaux et expérimentations pour explorer de nouvelles pistes de recherche sur les systèmes distribués et à architectures ad hoc.

Quatrième partie

Analyse formelle de systèmes et développement

Table des Matières

CHAPITRE 7

Analyse multifacette

Dans ce chapitre nous abordons des travaux effectués sur l'analyse formelle multifacette.

En effet, nous avions souligné concernant les motivations sur l'intégration de méthodes, que le développement d'un système complexe requiert l'emploi de différentes méthodes, techniques et de divers outils là où et lorsqu'ils sont appropriés, afin de mieux appréhender les différentes facettes qui caractérisent le système. Très souvent les techniques et outils emploient des modèles ou langages d'entrée spécifiques. Un *modèle spécifique* est par conséquent un modèle qui traite ou prend en compte des caractéristiques particulières d'un problème, d'un système ou d'un formalisme et non pas la globalité des caractéristiques du problème ou du système.

L'emploi de différentes méthodes ou techniques pour appréhender les multiples facettes d'un système peut conduire à des résultats incohérents s'il n'y a aucune assurance que les modèles de départ sont cohérents voire équivalents. De plus c'est le développeur qui a la charge de s'assurer d'une telle cohérence entre les modèles qu'il utilise.

Il y a donc un réel défi qui consiste à élaborer des techniques d'analyse couvrant divers aspects d'un même modèle ou système.

L'approche que nous développons complète celle hétérogène, de l'analyse formelle dans le cadre de l'intégration de méthodes, où l'analyse est faite après avoir construit puis intégré différents modèles.

Ici nous envisageons une analyse multifacette qui consiste à partir dès le début d'un modèle formel intégrant des propriétés communes des différentes facettes à analyser. Par exemple, il est plus réasliste pour un projet de développement d'un système, que les différentes équipes devant travailler sur diverses facettes du système, partent d'un socle commun d'exigences, de contraintes et de propriétés à respecter afin d'assurer la cohérence entre les résultats des équipes, plutôt que de laisser chaque équipe travailler indépendemment sur sa partie puis de chercher à intgégrer les résultats.

1. Quelques limitations de l'analyse formelle monofacette

1.1. Approche classique de l'analyse formelle. On construit un modèle mathématique M d'un système S ayant des propriétés voulues P, puis on s'assure que le modèle vérifie bien les propriétés P.

Considérons comme cadre de présentation celui des systèmes à événements discrets. Il existe plusieurs méthodes de spécification de ces systèmes : les automates à états, les réseaux de Pétri, les systèmes à actions, la méthode B événementielle, etc.

Soit à spécifier et vérifier un système donné en utilisant une des méthodes. On est alors dans la situation où, selon les possibilités offertes par la méthode (et ses outils) on va pouvoir exprimer certaines propriétés et pas d'autres. Mais en fin de compte on aura vérifier : $M \vDash P$.

1.2. Limitations. Admettons qu'on ait besoin d'analyser certains aspects, formalisés par des propriétés P_i, du système à l'aide d'outils divers utilisant des formalismes ou modèles d'entrée différents ; on doit construire différents modèles M_1, M_2, M_3, etc puis montrer

$$M_1 \models P_1$$
$$M_2 \models P_2$$
$$M_3 \models P_3$$
$$\dots$$

Que peut-on conclure sur le système global, sur le rapport entre les modèles M_i et les propriétés P_i ? Rien !

1.3. Pistes de solutions. Pour tenter de répondre aux questions précédentes une piste de solution est celle d'examiner les rapports entre les modèles M_i et de conclure dans le cas idéal où ces modèles sont 'similaires' que le système global a bien les différentes propriétés analysées. Dans la pratique ce cas idéal, sera loin d'être trivial. Il faudra montrer l'équivalence entre les modèles ; problème difficile voire insoluble. Nous retombons alors sur les questions abordées auparavant sur la compatibilité entre modèles. Dans les cas favorables où les modèles sont compatibles, il est possible d'effectuer des plongements entre les modèles, et de cerner de proche en proche l'intervalle des solutions.

De façon générale, il s'agit de la question du transfert de propriétés entre systèmes logiques ; on se retrouve dans le cadre de la logique ou de la théorie des modèles, où les questions ne sont pas triviales. En toute généralité, il est difficile d'établir les relations entre les modèles d'entrée.

Par conséquent, dans un contexte d'analyse multifacette, la démarche classique ne convient pas. Nous proposons dans la suite, une démarche basée sur la *dérivation de modèles* à partir d'un modèle de référence. L'idée est de construire à priori des modèles reliés d'une certaine façon, au lieu de chercher à les relier à postériori.

2. Analyse multifacette

Nous avons étudié [**Att05b**] une approche pour l'analyse multifacette où nous proposons l'emploi d'un *modèle de référence* comme base de modèles spécifiques et d'analyses spécifiques même si les formats d'entrée de ces dernières sont différents.

2.1. Présentation de la méthode. Notre proposition consiste (cf. Figure 1) à

(1) construire un modèle général abstrait à partir du système à étudier,

(2) dériver ou traduire systématiquement à partir de ce modèle, des modèles spécifiques aux différentes techniques d'analyse,

(3) effectuer les analyses sur les modèles spécifiques ou sur leurs extensions,

(4) assurer la cohérence entre le modèle de référence et les spécifiques en répercutant les corrections de chaque modèle spécifique sur le modèle de référence puis en dérivant (systématiquement) tous les modèles spécifiques en cours. Le résultat de chaque facette analysée participe ainsi à l'amélioration du système global.

Dans la suite du processus de développement, un seul modèle peut être approprié à la génération de code, les autres ayant permis entre temps de s'assurer de certaines propriétés, d'effectuer des simulations, etc.

Par exemple un modèle en B peut être utilisé pour la génération de code par raffinements successifs alors qu'un modèle à base de réseaux de Pétri permettra la simulation et la preuve de certaines autres propriétés.

Le modèle de référence peut être un modèle mathématique très général ou un modèle exprimé dans un langage ad hoc. Cette dernière solution parait restrictive à cause de la syntaxe et de la sémantique du langage choisi. Cependant ces restrictions peuvent aussi faciliter la construction de modèles effectifs contrairement à ce que permettrait un modèle mathématique plus abstrait.

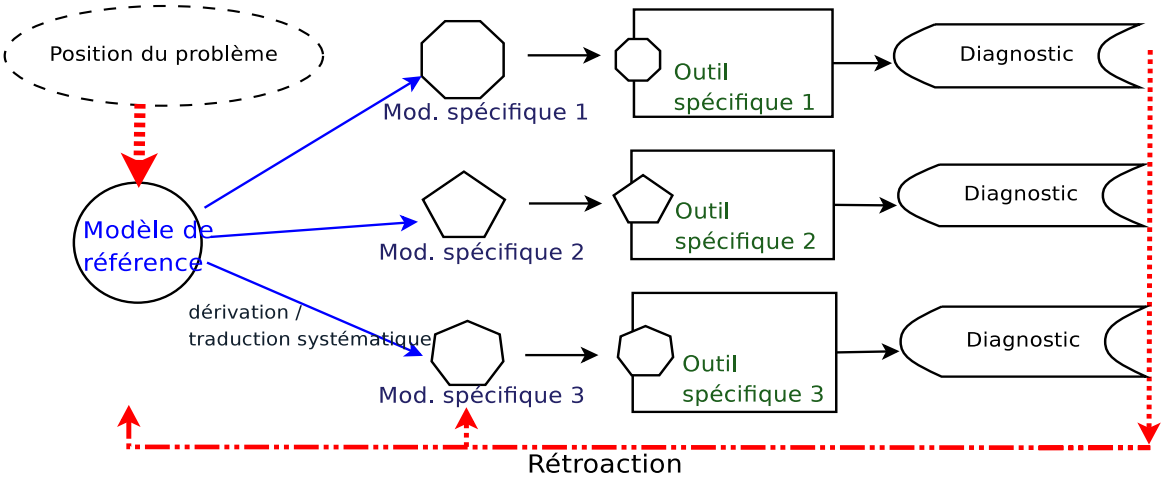

FIG. 1. Schéma de principe de l'analyse multifacette

Alors que dans le cadre de l'intégration de méthodes, nous partons de plusieurs modèles différents vers un modèle global pour l'analyse, la méthode que nous proposons pour l'analyse multifacette consiste à partir d'un modèle formel puis à en dériver plusieurs autres qui sont appropriés à différentes analyses.

PROPOSITION 7 (Dérivation de modèles). *La dérivation de modèles spécifiques à partir d'un modèle de base est nécessaire pour assurer la cohérence de l'analyse multifacette.*

Les modèles spécifiques peuvent être homomorphes ou non au modèle de base. Mais il doit être possible de modifier le modèle de base en rétroaction par rapport aux modèles spécifiques comme indiqué sur la figure 1.

Principe de construction du modèle de référence. La principale caractéristique du modèle de référence est qu'il donne une abstraction globale du système à étudier et ce, indépendemment de toute technique et de tout outil d'analyse.

Le modèle de référence doit donc être construit à un niveau d'abstraction qui permette d'exprimer les propriétés globales du système. Selon la nature du système, on peut utiliser la logique du premier ordre et exprimer des propriétés invariantes, exprimer des propriétés temporelles, etc ou bien utiliser un graphe d'état avec des propriétés d'accessibilité, etc

En ce qui concerne la dérivation des modèles spécifiques, elle se fait d'une part en fonction des outils visés (parce qu'ils offrent des fonctionnalités interessantes pour les études) et de leurs langages d'entrée et d'autre part en fonction des compatibilités de traduction et de compromis de préservation de propriétés.

Dans le cas général, on va soit perdre certaines informations soit en ajouter d'autres pour passer du modèle de référence au modèle spécifique. Il faut alors savoir et analyser quelles propriétés sont préservées et lesquelles sont ignorées avant de poursuivre l'étude, afin que le modèle spécifique soit pertinent. Par exemple, il est courant de réduire la plage de valeurs des données d'un système afin de pouvoir utiliser des outils d'eploration de graphe d'états (outils de *model checking*) ; en général on réduit ainsi l'étude d'un système infini à l'étude d'un système fini afin d'exploiter les outils disponibles pour des systèmes finis. Cependant le compromis qui est fait ici par rapport aux propriétés est que lorsque le système réduit est exempt d'erreurs, cele ne veut pas dire que le système initial l'est ; en revanche lorsqu'on détecte une erreur sur le système réduit, on est sûr que le système initial est aussi erroné et on peut ainsi le corriger.

Les modèles spécifiques et l'analyse des propriétés. Les modèles spécifiques sont construits par traduction ou par dérivation à partir du modèle de référence. Concernant la traduction, on s'attachera dans

le cas idéal à préserver la sémantique et aussi maintenir les propriétés globales énoncées sur le modèle de référence. On peut utiliser les techniques de bissimulation pour établir la préservation de la sémantique et des propriétés globales.

Lorsqu'il est nécessaire de réduire le modèle pendant la dérivation/traduction, par exemple pour la gestion des données, les propriétés ne sont pas toujours transférables du modèle spécifique au modèle de référence. Par exemple lorsqu'on fait l'analyse de propriétés sur la restriction d'un comportement paramétré avec des données à un comportement non paramétré par des données, on ne vérifie en fin de compte qu'une partie des propriétés.

Lorsqu'il est nécesessaire d'étendre le modèle de réference pendant la dérivation/traduction, du fait d'une plus grande expressivité du modèle spécifique, on se retrouve dans un cas de dérivation de modèle où l'interprétation des propriétés globales est plus minutieuse mais ici, les propriétés prouvées sur le modèle étendu le sont sur le modèle de référence, en revanche l'inverse n'est pas vrai. Lorsqu'on étudie des propriétés définies localement sur le modèle spécifique, cette analyse vient en complément des propriétés globales (les contraintes de temps par exemple rentrent dans ce cadre).

Il en va de même pour les propriétés (P') exprimées par rapport aux modèles spécifiques ; soit elles sont de simples traductions des propriétés globales (P) du modèle de référence, soit elles sont spécifques aux modèles dérivés, soit elles sont en relation (extension/réduction) avec les propriétés du modèle de référence.

En conséquence, selon la relation entre modèle de référence (M) et les modèles spécifiques (M'), on sait tirer parti des analyses effectuées. Ce que nous pouvons résumer par des lois de la forme suivante.

Soit $\mathcal{R}_m$:

– *Réduction* du modèle de référence,

– *Extension* du modèle de référence

– *Adaptation/traduction* du modèle

Soit $\mathcal{R}_p$:

– *Explicitation* (réduction, extension, traduction) du lien entre P et P'

$$\frac{M \vDash P \quad \mathcal{R}_m(M, M') \quad M' \vDash P' \quad \mathcal{R}_p(P, P')}{M \vDash \mathcal{R}_p(P, P')}$$

En cas de réduction, les propriétés non prouvées sur le modèle réduit sont également non prouvées sur le modèle de référence.

En cas d'extension, les propriétés prouvées sur le modèle dérivé sont également prouvées sur le modèle de référence.

L'impact des lois sur l'analyse des propriétés est ainsi résumé par la table suivante

Réduction	transfert des propriétés non vérifiées de M' à M
Extension	transfert des propriétés vérifiées de M' à M
Traduction	transfert des propriétés non vérifiées transfert des propriétés vérifiées

Dans la suite nous prenons un exemple, relativement à la nature du système (système réactif), pour illustrer la mise en œuvre de notre méthode.

2.2. Illustration : application aux systèmes réactifs. Nous considérons ici le principe d'étude des systèmes réactifs tel qu'il est résumé dans la figure 2. Nous illustrons à partir d'un tel système impliquant un système contrôlé et un système de contrôle, la mise en œuvre d'une analyse basée sur un modèle de référence incluant une partie statique et une partie dynamique.

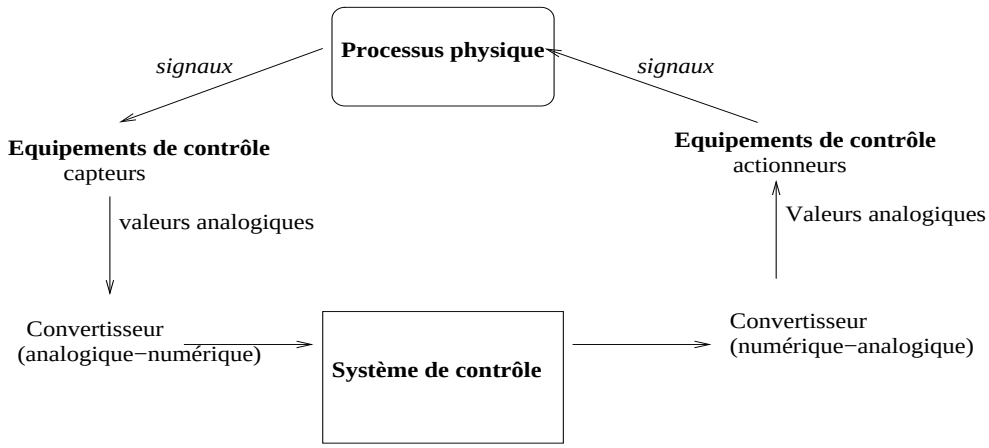

FIG. 2. Schéma de principe des systèmes de contrôle

2.2.1. *Construction d'un modèle de référence formelle.* Un modèle de référence sert de support pour des analyses de propriétés utilisant par exemple les techniques de preuve ou l'évaluation de modèle ; chacune de ces techniques emploie un formalisme d'entrée spécifique. Un modèle de référence est constitué ici d'une partie statique et d'une partie dynamique. Nous illustrons ici nos propos avec un exemple basé sur un système à forte dominante contrôle mais à faible dominante donnée ; un tel système est décrit par des règles logiques. Par conséquent des variables booléennes sont suffisantes pour les descriptions et l'extraction de certaines conclusions.

Par exemple, par rapport à une étude de contrôle de niveau d'eau dans une cuve alimentée par une vanne, si les variables NE, EV sont booléennes et désignent respectivement un niveau d'eau élevé, une vanne ouverte, alors une règle logique de la forme

$$(NE \wedge EV) \Rightarrow CMD = FermerVanne$$

permet d'exprimer que si le niveau d'eau est élevé et la vanne (contrôlée) est ouverte alors on donne un ordre de fermeture de la vanne. *CMD* est ainsi une variable d'état pour noter des actions à entreprendre.

Description de la partie statique. La partie statique d'un système de contrôle, dont le comportement est décrit par des règles logiques, est capturée par une structure abstraite $\mathcal{A} = \langle \mathcal{I}, \mathcal{S}, \mathcal{O}, \mathcal{R} \rangle$ où :

- $\mathcal{I}$ est un ensemble fini de variables booléennes : les variables d'entrée (il s'agit des entrées par rapport au système de contrôle),
- $\mathcal{S}$ est un ensemble fini de variables booléennes : les variables d'état (il s'agit de l'état du système contrôlé, il est indépendant du contrôle, il est observé),
- $\mathcal{O}$ est un ensemble fini de variables booléennes : les variables de sortie (sortie du système de contrôle) et
- $\mathcal{R}$ est un ensemble fini de règles de la forme : *condition* $\rightarrow$ *variable*. Une forme abstraite d'une règle est un couple de la forme $(condition, variable)$. Nous avons $\mathcal{R} \subseteq \mathcal{L}_{prop}(\mathcal{I}, \mathcal{S}) \times \mathcal{O}$.
  $\mathcal{L}_{prop}(\mathcal{I}, \mathcal{S})$ est l'ensemble des conditions (expressions logiques) construites à l'aide des connecteurs logiques et des variables dans $\mathcal{I}$ et $\mathcal{S}$. Pour manipuler les règles et les variables, les fonctions rhs and lhs dénotent respectivement la partie droite et la partie gauche d'une règle. Par exemple nous avons $\forall R_i \in \mathcal{R}.\mathsf{rhs}(R_i) \in \mathcal{O}$. De plus, nous utilisons la fonction vars pour dénoter l'ensemble des variables apparaissant dans la condition d'une règle : $\mathsf{vars}(\mathsf{lhs}(R_i))$.

$\mathcal{R}$ capture (via des règles données) la relation entre $\mathcal{I}, \mathcal{S}$ et $\mathcal{O}$. Ainsi, les règles logiques expriment par leurs parties gauches les conditions de réalisation de certaines actions sur le système contrôlé ou bien la

présence de dysfonctionnements du système contrôlé. Les parties droites des règles sont les variables de sortie. Les actions à entreprendre par le système de contrôle ou les dysfonctionnements constatés sont ainsi traités via les variables de sortie. La répercussion des effets sur le système peut entrainer le changement de l'état du système et par conséquent la modification (observée) des variables d'état.

Variables d'entrée ($\mathcal{I}$) : Elles décrivent le processus contrôlé. Elles correspondent essentiellement aux informations provenant de l'envrironnement du système de contrôle (des capteurs, ...) ; elles sont modélisées comme des booléens. D'autres variables d'entrée peuvent correspondre à des ordres (ou des commandes) donnés par un opérateur humain ou un dispositif dédié.

Si l'opérateur humain entreprend une action par une commande, la variable associée à cette action est mise à *true* autrement elle reste à *false*.

Variables d'état ($\mathcal{S}$) : Elles décrivent (l'espace d'état du) le système de contrôle et sont utilisées comme données d'entrée ou de sortie. Par exemple pour un système contrôlé, une variable d'état peut être associée à chaque anomalie. En cas d'anomalie la consultation de la variable permet de rendre compte de la détection de l'anomalie (positionnement après scrutation) et de la même manière lorsqu'une anomalie est résolue la variable d'état prend une valeur appropriée. De cette façon, l'état d'un système contrôlé est déterminé par l'ensemble des variables utilisées pour modéliser les différents composants du système en plus des variables utilisées pour raisonner sur le système.

Variables de sortie ($\mathcal{O}$) : Les valeurs des variables de sortie sont calculées à partir des variables d'entrée et des variables d'état. Par exemple si un système de contrôle détecte un mauvais fonctionnement la variable (état) correspondant est mise à jour. $\mathcal{O}$ est partitionné par les sous-ensembles $\mathcal{O}_m$ et $\mathcal{O}_o$ afin de discerner entre les actions à faire et les informations relatives à l'état du modèle du système contrôlé.
$\mathcal{O}_m$ contient les variables de sortie qui donnent des informations sur l'état du système (informations allant de l'environnement vers le système) ; $\mathcal{O}_o$ contient les variables de sortie qui donnent les commandes ou actions entreprises par le système de contrôle (du système de contrôle vers son environnement).

En travaillant à partir de cette architecture du système, notre modèle de référence est un modèle à états à partir duquel on dérive des modèles homomorphes.

Nous avons découvert que cette partie de notre travail basée sur un modèle de référence incluant l'environnement, présente de fortes ressemblances avec les travaux de PARNAS sur le modèle à quatre variables [**PM95**]. Dans cette optique, il est intéressant d'expérimenter l'usage du modèle à quatre variables (outillé avec SCR*) comme modèle de référence pour des analyses multifacettes.

Partie dynamique du modèle. Les systèmes de transition constituent le cadre général mais la partie dynamique peut être effectivement construite en rapport avec les outils d'analyse visés. Par exemple des processus asynchrones conviennent avec l'utilisation de Spin alors que des opérations modélisées avec des substitutions généralisées conviendront à l'utilisation de la méthode B.

Dans le cadre de cette proposition d'analyse multifacette à partir de modèles de référence, une plateforma à la UTP (voir la section 4.5, chapitre 3) peut servir de modèle de référence à partir duquel dériver des modèles spécifiques sujets à l'analyse selon des outils spécifiques.

2.2.2. *Dérivation des modèles spécifiques et analyse.* A partir du modèle de référence présenté précédemment, nous avons effectué des expérimentations en utilisant Atelier B [**Att01**] et Spin [**Att04**]. Nous

avons défini des procédures de traduction systématique des règles logiques en opérations B pour ce qui concerne les expérimentations avec Atelier B et des traductions en processus Promela pour ce qui concerne les expérimentations avec Spin.

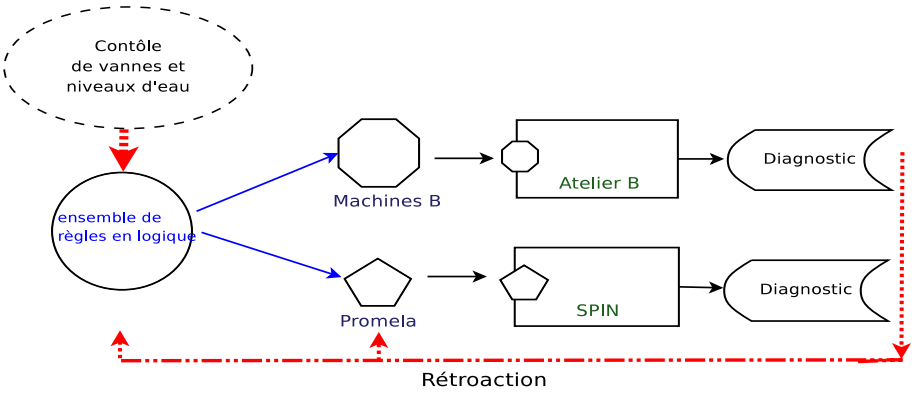

FIG. 3. Expérimentations sur un système réactif

Ces expérimentations complémentaires ont permis d'étudier de façon relativement complète le système réactif de contrôle de processus (ouverture/fermeture de vanne pour l'alimentaion en eau).

3. Combinaison de preuve de théorèmes et d'évaluation de modèles

Le manque de guides efficaces pour la pratique des méthodes formelles et la systématisation de l'analyse assistée par des outils, constituent des défis importants, dans le cadre des systèmes logiciels. Cela constitue une motivation principale pour une partie de nos travaux. Nous rendons compte ici d'expérimentations faites en analysant une même spécification de différentes façons. La méthode B et l'Atelier B sont utilisés pour les spécifications formelles, pour l'analyse de sûreté et pour les raffinements. L'outil ProB est utilisé pour compléter cette étude avec l'évaluation de modèle (*model checking*). Cela a permis de découvrir, lors des expérimentations, des erreurs et ainsi d'améliorer les spécifications initiales. Les résultats de ces travaux sont publiés dans [**Att04, Att05b, Att06b**].

3.1. Motivations. La mécanisation de l'analyse des systèmes et du processus de développement contribuent à garantir la qualité des logiciels développés. Nous nous intéressons ici à la combinaison effective des outils d'analyse formelle et de développement à travers leur emploi sur des projets. Ces derniers présentent différentes facettes. Les facettes analysées relèvent souvent de la vivacité et de la sûreté. La preuve de théorème est utilisée pour la cohérence, le raffinement et les propriétés de sûreté. L'évaluation de modèle est utilisée pour les propriétés de vivacité. Plus généralement l'évaluation de modèle est souvent utilisée pour les systèmes où les aspects contrôle prédominent alors que la preuve de théorèmes est utilisée pour les systèmes très généraux ou des systèmes où les données prédominent. La combinaison des deux approches est nécessaire lorsque la séparation des aspects n'est pas simple, c'est souvent le cas pour des systèmes de taille importante.

Très souvent ces techniques requièrent un formalisme ou un modèle d'entrée spécifique. Concernant les plates-formes opérationnelles, il n'y a pas d'interaction directe entre les outils dédiés aux différentes techniques. Les utilisateurs (ou développeurs) doivent manipuler différentes entrées selon les outils cibles.

C'est là une limitation pour leur complémentarité dans les développements réels. L'analyse multifacette consiste à appliquer plusieurs méthodes et outils appropriés au même modèle formel pour en étudier les différentes facettes. Cela peut être fait au cours d'un processus de développement allant de la spécification formelle abstraite au code exécutable en passant par l'analyse formelle de propriétés. Nous montrons sur un

cas comment effectuer une telle analyse. Nous utilisons pour ce faire Atelier B et ProB. Ici les spécifications B servent de modèle de référence pour les analyses effectuées.

Généralement, les techniques basées sur la preuve de théorème demandent des interactions avec l'utilisateur et beaucoup de temps surtout pour les projets de grande taille. L'analyse multifacette peut intervenir bénéfiquement sur ce type de projets ; elle peut contribuer à accélérer l'analyse formelle et couvrir d'autres aspects non couverts par la preuve de théorèmes. Le choix des outils pour l'analyse multifacette est important afin de réduire les coûts de développement.

Ici l'outil ProB [**LB03, LT05**] est bien adapté pour compléter l'Atelier B ; en effet ProB utilise le même formalisme d'entrée que l'Atelier B.

Nous mettons en exergue trois principaux aspects dans ce travail : *i)* l'étude de cas révèle des caractéristiques intéressante des spécifications en B : par exemple la prise en compte de l'apparition dynamique de processus qui peuvent alors interagir avec des processus existants *ii)* la complémentarité des outils employés pour renforcer l'analyse ; *iii)* un guide méthodologique pour l'analyse multifacette.

Pour l'analyse multifacette, un modèle de référence constitué d'une partie statique et d'une partie dynamique est utilisée. Selon le système à étudier, la partie statique décrit les entrées, les sorties, les variables d'état et les informations de typage. Il s'agit essentiellement d'un modèle à état abstrait à partir duquel des modèles spécifiques isomorphes sont dérivés pour correspondre à des formalismes spécifiques.

La partie dynamique est constituée de plusieurs opérations vues comme une relation de transition sur l'espace d'états décrit par les variables de la partie statique.

3.2. Présentation de l'étude de cas. Il s'agit d'un cas posant un problème générique : le problème des lecteurs-rédacteurs. Dans le cadre des applications distribuées (par exemple les services Internet), il est utilisé pour traiter les accès aux ressources.

Le problème des lecteurs-rédacteurs est un problème classique d'exclusion mutuelle et de synchronisation dont la solution est utilisée dans plusieurs domaines d'application tels que les systèmes d'exploitations et l'accès à des bases de données par des processus. Deux types de *processus* sont considérés : les processus lecteurs et les processus rédacteurs. L'écriture par les processus rédacteurs est exclusive mais plusieurs lecteurs peuvent lire simultanément.

Par exemple le système de gestion de fichiers d'un système d'exploitation implante une logique de verrouillage des fichiers afin de le prévenir des incohérences sur les fichiers partagés par les processus concurrents.

Une politique d'exclusion et un mécanisme de synchronisation son nécessaires parce qu'une incohérence peut arriver si certains processus (les lecteurs) lisent une ressource lorsque d'autres processus (les rédacteurs) écrivent la ressource.

Analyse. Il y a plusieurs stratégies possibles comme logique de gestion de cette ressource. Une stratégie simple autoriserait l'accès à la ressource à tout moment soit à un nombre quelconque de lecteurs soit à un seul rédacteur. Cette stratégie garantit l'intégrité de la ressource mais elle n'est pas très satisfaisante ; elle donne la préférence aux lecteurs par rapport aux rédacteurs. Considérer par exemple qu'il y ait plusieurs demandes de lecture, s'il y a un lecteur en cours, alors les lecteurs prennent la priorité par rapport aux rédacteurs. Ces derniers peuvent subir la *famine* qui est une propriété non désirée ici.

Une des solutions souvent utilisées pour palier cette situation est de limiter le nombre de lecteurs consécutifs qui peuvent accéder à a ressource. Ainsi les rédacteurs peuvent retrouver la priorité lorsque le nombre de lecteurs consécutifs prévu est atteint. Cette solution est plutôt juste et n'affame pas les rédacteurs.

Afin de développer des logiciels qui intègrent de telles stratégies d'accès, il est important de les étudier soigneusement avant leur implantation et leur développement. Une étude formelle du système en vue peut

aider à découvrir les défauts et les corriger ; elle peut aussi aider à développer correctement le système envisagé. C'est ce que nous illustrons dans la suite.

Derrière l'apparence simpliste de ce cas se cachent des aspects intéressants en matière de spécification formelle. Tous les formalismes de spécification ne peuvent pas être utilisés pour traiter ce cas simple mais réaliste. Dans un système d'exploitation par exemple, les processus sont créés à tout moment, ils s'exécutent et se terminent. Le nombre de processus n'est pas connu à l'avance, leurs comportements ne sont pas complètement connus, mais ils peuvent interagir avec des processus existants.

La plupart des formalismes de spécification (par exemple les algèbres de processus) ne traitent pas la composition d'un nombre indéfini de processus ; ils exigent la donnée non seulement des comportements des processus mais de leur nombre avant la composition. De plus il n'y a pas de façon simple de gérer la création dynamique de processus ; un système serait par exemple étudié avec un nombre fixé de lecteurs et de rédacteurs. En conséquence certains aspects liés à la création dynamique des processus ne sont pas bien couverts. L'utilisation de Event B est ainsi justifiée par le fait que nous pouvons y prendre en compte les aspects évoqués ci-dessus.

Nous présentons dans la suite l'analyse multifacette sur l'étude.

3.3. Analyse formelle des lecteurs/rédacteurs. Nous utilisons une spécification B comme modèle de référence. Elle possède une partie statique et une partie dynamique. Nous n'avons pas besoin de dériver des modèles d'entrée spécifique puisque les outils choisis utilisent les spécifications B comme entrée. Nous effectuons les analyses directement sur les mêmes spécifications B. Les rétroactions venant des outils permettent d'améliorer le modèle de référence.

Les principales étapes de l'expérience sont : la spécification abstraite en Event-B, le raffinement et l'analyse des propriétés.

3.3.1. *La spécification B : le modèle de référence.* L'approche Event B permet de spécifier le comportement asynchrone de processus. De plus on peut y spécifier l'interaction entre un nombre indéterminé de tels processus asynchrones en utilisant les substitutions nondéterministes gardées.

Nous considérons deux types de processus en nous conformant au cahier de charges : les processus lecteurs et les processus rédacteurs.

Première étape : spécification abstraite Nous avons spécifié deux systèmes abstraits : le premier est nommé *Readers*, le second est nommé *Writers*. Ils utilisent les mêmes ensembles READER et WRITER et leurs spécifications sont quasiment symétriques. Les ensembles READER et WRITER sont les abstractions des processus lecteurs et rédacteurs.

Le schéma du système abstrait des *Readers* est donné dans la figure 4, celui du système abstrait *Writers* est similaire.

La variable *readers* décrit les lecteurs courants (abstraction de processus lecteurs). L'invariant du système abstrait *Readers* est donné dans la figure 5.

Propriété d'exclusion. Le prédicat $\mathit{not}((\mathit{card}(\mathit{activeWriter}) = 1) \wedge (\mathit{card}(\mathit{activeReaders}) \geq 1))$ qui apparaît dans l'invariant (figure 5) décrit la propriété d'exclusion requise entre les lecteurs et rédacteurs. Il stipule qu'on ne peut avoir simultanément des lecteurs et un écrivain actifs. C'est une propriété de correction incluse dans l'invariant.

Le système abstrait *Readers* possède les événements `want2read`, `reading`, `endReading` et `newReader`. L'événement `want2read` survient lorsqu'un lecteur donné *rr* demande à lire ; il est alors noté dans la variable *waitingReaders* et doit commencer la lecture dès que possible en fonction de l'état global du système.

SYSTEM *Readers*
SETS *READER WRITER*
VARIABLES
readers, *waitingReaders*, *activeReaders*, *activeWriter*
INVARIANT ···
INITIALISATION ···
EVENTS
`want2read` $\widehat{=}$ ···
; `reading` $\widehat{=}$ ···
; `endReading` $\widehat{=}$ ···
; `newReader` $\widehat{=}$ ···
END

FIG. 4. Spécification B des lecteurs

INVARIANT
$readers \subseteq READER \wedge waitingReaders \subseteq READER$
$\wedge\ activeReaders \subseteq READER$
$\wedge\ card(activeReaders) \geq 0$
$\wedge\ readers \cap waitingReaders = \{\}$
$\wedge\ activeReaders \cap waitingReaders = \{\}$
$\wedge\ activeReaders \cap readers = \{\}$
$\wedge\ activeWriter \subseteq WRITER \wedge card(activeWriter) \leq 1$
$\wedge\ \neg\ ((card(activeWriter) = 1) \wedge$
$(card(activeReaders) \geq 1))$

FIG. 5. Spécification B des lecteurs : l'invariant

`want2read` $\widehat{=}$
ANY *rr* WHERE
$rr \in readers \wedge rr \notin waitingReaders$
$\wedge\ rr \notin activeReaders$
THEN
$waitingReaders := waitingReaders \cup \{rr\}$
$\|\ readers := readers - \{rr\}$
END

La spécification de l'événement `reading` est la suivante :

`reading` $\widehat{=}$
ANY *rr* WHERE
$rr \in waitingReaders \wedge activeWriter = \{\}$
THEN
$activeReaders := activeReaders \cup \{rr\}$
$\|\ waitingReaders := waitingReaders - \{rr\}$
END

La garde de l'événement `reading` stipule que : n'importe quel lecteur *rr* en attente de lecture, et tel qu'il n'y a aucun rédacteur en attente, peut commencer à lire. Notons qu'on n'a pas spécifié uniquement le

comportement d'un processus lecteur, mais celui de n'importe quel nombre de lecteurs. Cela est fait grâce à l'usage du nondéterminisme pour spécifier les événements du système abstrait.

L'invariant du second système abstrait (*Writers*) est similaire à celui de *Readers* mais il a en plus la contrainte sur l'exclusion entre les rédacteurs : $card(activeWriter) \leq 1$.

Le système abstrait des rédacteurs (*Writers*) a la même structure que *Readers* ; il a les événements `want2write`, `writing`, `endWriting` et `newWriter`. L'événement `writing` est comme suit :

`writing` $\widehat{=}$
ANY *ww* WHERE
$ww \in waitingWriters \wedge activeReaders = \{\}$
THEN
$activeWriter := \{ww\}$
$\|$ $waitingWriters := waitingWriters - \{ww\}$
END

Ensuite nous combinons parallèlement (voir chapitre 5) les systèmes abstraits pour obtenir un système interactif global *readWrite*.

$$readWrite \widehat{=} Readers]|[\, Writers$$

Le système abstrait résultant a comme invariant la conjonction des invariants des sous-systèmes ; ses événements consistent en une union des événements venant de *Readers* et *Writers*.

Dans cette première étape de l'étude nous avons considéré une stratégie simple (un nombre quelconque de lecteurs ou un seul rédacteur) pour définir une première solution opérationnelle. Nous avons ensuite raffiné cette solution pour prendre en compte la propriété d'équité.

Deuxième étape : raffinement Nous raffinons les spécifications qui résultent de la première étape ; nous incluons la variable *nbConsecutiveReaders*, une constante *maxConsecutiveR* et une nouvelle propriété dans l'invariant. Nous avons introduit également deux nouveaux événements : `leaveReader` et `leaveWriter`. Le premier simule la sortie/destruction d'un lecteur, le dernier simule la sortie/destruction de processus.

La propriété incluse concerne l'équité : il n'y a pas que des lecteurs ou des rédacteurs qui sont autorisés à accéder à la ressource. Cela est géré en limitant (avec *maxConsecutiveR*) le nombre de lectures consécutives.

Les événements et leurs gardes son réécrits et mis à jour en fonction des nouvelles variables. Nous avons utilisé une approche montante : les systèmes abstraits *Readers* et *Writers* sont raffinés respectivement en *ReadersR* en *WritersR*. Ces derniers sont composés parallèlement pour obtenir le système interactif global nommé *readWriteR*. Dans le raffinement, la variable *nbConsecutiveR* est mise à 0 lorsqu'un rédacteur obtient l'accès en écriture.

$$readWriteR \widehat{=} ReadersR]|[\, WritersR$$

L'invariant du système raffiné est le suivant :

INVARIANT
$writers \subseteq WRITER$
$\wedge\ activeWriter \subseteq WRITER$
$\wedge\ card(activeWriter) \leq 1$
$\wedge\ waitingWriters \subseteq WRITER$
$\wedge\ writers \cap waitingWriters = \{\}$
$\wedge\ activeWriter \cap waitingWriters = \{\}$
$\wedge\ activeWriter \cap writers = \{\}$
$\wedge\ nbActiveReaders \in NAT$
$\wedge\ nbActiveReaders = card(activeReaders)$
$\wedge\ \mathbf{nbConsecutiveR} \in \mathbf{NAT}$
$\wedge\ \mathbf{nbConsecutiveR} \leq \mathbf{maxConsecutiveR}$
$\wedge\ readers \subseteq READER$
$\wedge\ waitingReaders \subseteq READER$
$\wedge\ activeReaders \subseteq READER$
$\wedge\ card(activeReaders) \geq 0$
$\wedge\ readers \cap waitingReaders = \{\}$
$\wedge\ activeReaders \cap waitingReaders = \{\}$
$\wedge\ activeReaders \cap readers = \{\}$
$\wedge\ \neg\ ((card(activeWriter) = 1) \wedge$
$\quad (card(activeReaders) \geq 1))$

3.3.2. *Expérimentations.* Toutes les spécifications ont été analysées (cohérence et raffinement) et prouvées correctes en utilisant l'Atelier B.

Dans le contexte d'une analyse formelle sans génération de code l'étude aurait pris fin là. Mais dans le cadre de l'analyse multifacette nous poursuivons l'étude en animant le système modélisé et en l'explorant avec l'outil ProB.

Animation. Le système *readWriteR* est animé en utilisant la stratégie aléatoire pour les déclenchements et un paramétrage du nombre d'opérations à 40. Le résultat semble satisfaisant ; la couverture d'état montre que toutes les opérations (correspondant aux événements) sont couvertes.

Test du raffinement. L'espace d'état entier du système *readWrite* est exploré. Le graphe de transition résultant est stocké et les traces sont comparées avec celles du système raffiné *readWriteR*. Aucune erreur n'est détectée.

Évaluation de modèle L'analyse est poursuivie par l'exploration des états en utilisant la fonctionnalité Temporal Model Checking.

La fonctionnalité Compute Coverage permet d'avoir le nombre d'états couverts pendant l'exploration et indique les états bloquants (*deadlocks*). Un état d'interblocage a été détecté ici sur 3223 états. Nous avons découvert la cause de cette situation en analysant cet état : un processus lecteur (READER2) est actif et $nbConsecutiveR = 10$; normalement, l'événement `endReading` doit se produire mais ce n'est pas le cas. Nous revenons alors à la spécification qui suit :

```
endReading ≙
  ANY rr WHERE /* an active reader finishes reading */
    rr : activeReaders ∧ nbActiveReaders > 1
  THEN
    activeReaders := activeReaders − {rr}
  || readers := readers ∪ {rr}
  || nbActiveReaders := nbActiveReaders − 1
  END
```

Il apparaît que la garde de l'événement est fausse. Nous devions avoir $nbActiveReaders \geq 1$. En effet le problème est que, lorsqu'on a qu'un seul processus en cours de lecture, la garde `endReading` n'est pas vraie et le processus ne peut se terminer d'où le blocage. La spécification est alors corrigée en conséquence. Nous n'avions pas détecté le problème avec le prouveur Atelier B. En effet la preuve de cohérence, n'est pas en cause car l'invariant n'est pas violé.

Cela renforce l'obligation de preuve d'absence de blocage posée par Event B : *"si aucune garde n'est vraie le système est bloquant"*.

En conséquence ProB a aidé à découvrir une obligation de preuve (liée à Event B) non déchargée.

4. Synthèse et perspectives sur cette étude

Nous avons proposé et utilisé une approche d'analyse multifacette pour étudier des système. L'étude est illustrée par le problème des lecteurs-rédacteurs ; il aborde la composition, le raffinement, la preuve de théorème et l'évaluation de modèle afin de couvrir les diverses étapes dans un développement. Nous avons spécifié les lecteurs, les rédacteurs puis nous les avons combiné pour avoir un système interactif global. Une des principales caractéristiques de ce système est l'architecture dynamique des processus (composition d'un nombre indéfini de processus). Les systèmes abstraits *Readers* et *Writers* sont raffinés pour améliorer le modèle B initial et pour assurer les propriétés désirées. Le raffinement concerne la prise en compte de propriétés et l'introduction de nouveaux événements.

L'adoption de l'approche multifacette pousse le spécifieur à utiliser plusieurs techniques d'analyse. Ce faisant, la complémentarité des techniques et outils permet d'affiner l'étude entreprise.

Nous avons ainsi combiné les techniques de preuve de théorème et d'exploration de modèles via les outils Atelier B et ProB. Nous avons mis en évidence leur complémentarité, pour décharger les obligations de preuve et montrer l'absence de blocage.

Pour des systèmes de grande taille, nous pensons que l'approche multifacette est une voie intéressante ; le travail est un peu plus aisé lorsque les formalismes d'entrée des divers outils employés sont proches voir les mêmes, sinon il faut avoir recours à des traductions systématiques d'un formalisme à un autre afin de maintenir la cohérence globale. Dans cette étude nous avons considéré la méthode B mais il faut noter que parmi les environnements permettant la mise en œuvre de différentes techniques sur le même formalisme d'entrée on peut citer PVS [**Sha00**] ; le système PVS permet de combiner preuve et évaluation de modèles pour l'analyse de propriétés.

Dans le prolongement de ce travail nous travaillons sur l'analyse multifacette de modèles construit à partir des notations tabulaires de PARNAS [**PM95**].

CHAPITRE 8

Algèbre de spécifications multiparadigmes

La motivation de cette voie de recherche est le manque d'approches multifacettes pour spécifier, analyser et développer des systèmes. Plutôt que d'intégrer différentes méthodes traitant chacune un paradigme donné, nous explorons une voie vers une méthode pouvant intégrer à la base différents paradigmes mais extensible à d'autres paradigmes. On a ainsi une certaine *plasticité des modèles* à construire, c'est à dire que les modèles s'adaptent aux facettes ou évoluent selon les facettes exprimées. Cette idée est comparable à celle présente dans les automates à états finis, ou on peut procéder à des extensions du modèle pour prendre en compte différentes autres facettes. Dans le cadre des spécifications multiparadigmes, selon les besoins à exprimer dans un modèle, cette plasticité permettrait d'agréger des paradigmes de façon cohérente à un noyau.

1. Choix des matériaux et justifications

Nous motivons ici les choix faits dans la suite de cette proposition. Une brève présentation des matériaux est donnée en complément de ce que nous avions dans la première partie du mémoire ; nous insistons davantage ici sur les outils relatifs aux matériaux choisis ainsi que les aspects de la proposition qui sont précisément couverts par ces matériaux (en comparaison avec d'autres possibilités écartées). Le lecteur peut trouver des détails dans les références suivantes [**Gor, Pau93c, Mon96**].

1.1. Logique, ensembles et types. *Logique.* La logique est un formalisme largement répandu et utilisé pour la modélisation et le raisonnement formel. Il existe plusieurs logiques : la logique classique[1] et la logique intuitioniste [2] sont très utilisées dans les systèmes de raisonnement (preuve). Par conséquent, elles sont considérées à un très bas niveau dans le processus de conception et sont aussi utilisées comme support formel pour des formalismes plus évolués. La logique des fonctions partielles (LPF) introduites par C. JONES [**CJ91, JM94**] est aussi intéressante de certains points de vue ; par exemple pour prendre en compte les valeurs nondéterminées dans le raisonnement (l'exemple classique est celui de la valeur de vérité à donner à $x/0 = y$ où x, y sont des entiers). En effet le raisonnement sur les fonctions partielles est difficile en logique classique. Dans la pratique, à vart VDM, peu de systèmes de raisonnement et d'outils utilisent effectivement LPF ; en revanche on trouve des formalisations (en Coq par exemple) de LPF et d'autres logiques multivaluées, afin d'y raisonner en se servant d'outils généraux.

Théorie des types. Les types sont utilisés pour structurer les données. Les systèmes de type sont très populaires dans les langages et systèmes informatiques. En effet, la théorie des types renforcent les contrôles de cohérence dans les spécifications, les vérifications de propriétés et les programmes.

Logique d'ordre supérieur. La logique d'ordre supérieur est une logique qui admet des variables typées qui peuvent être des prédicats. Elle intègre un système de types pour structurer et controler les données ; elle est ainsi la plus populaire des théories de type. Elle est utilisée avec succès pour le raisonnement formel. De nombreux systèmes de preuve sont basés sur la logique d'ordre supérieur : Automath [**NGV94**], Coq [**DFH$^+$93**], HOL [**Gor93**], Nuprl [**Ca86**], PVS [**COR$^+$95**].

[1] avec le principe du tiers exclu : $A \vee \neg A$ est valide si on exhibe soit une preuve de A soit une preuve de $\neg A$

[2] pas de principe du *tiers exclu*, on ne peut avoir A ou $\neg A$, si les preuves ne sont pas données

Les manipulations de fonctions sont des traitements élémentaires en informatique, elles sont des primitives dans les logiques d'ordre supérieur. Cependant il y a une grande variété de logiques d'ordre supérieur mais elles ne disposent pas d'une formulation servant de référence ou de standard. Elles diffèrent aussi bien dans leurs notations que dans leur conception de la vérité (mathématique) [**Gor**]. Par exemple le système Coq est basé sur le calcul des constructions, PVS est basé sur une logique classique d'ordre supérieur avec des sous-types et des types dépendants, HOL supporte une théorie des types simple et polymorphe. Par conséquent il n'est pas aisé de transposer un raisonnement formel d'un contexte vers un autre (c'est le problème de transfert de propriétés entre logiques). Pratiquement cela veut dire que l'interaction entre les assistants de preuve bâtis sur ces logiques n'est pas simple.

Théorie des ensembles. C'est la formalisation la plus largement admise des mathématiques. Elle fournit de puissants mécanismes de modélisation pour les plateformes de spécification (Z, TLA, B). La théorie des ensembles axiomatisée est plus pragmatique ; ici les ensembles ne sont pas arbitraires mais ils doivent être construits[3] à partir d'ensembles donnés. La théorie des ensembles axiomatisée a été développée comme solution aux paradoxes relevés par RUSSEL sur la théorie naïve des ensembles. La théorie des ensembles a deux principales formulations (ou axiomatisations) bien acceptées : celle de ZERMELO-FRAENKEL (ZF) et celle de BERNAYS-GÖDEL (BG). Elles sont différentes mais ont le même pouvoir expressif [**Sho92**]. Cependant ZF est généralement considérée comme la théorie standard. Parmi les systèmes de raisonnement formel basés sur la théorie des ensembles, il y a EVES [**SKP$^+$93**], Mizar [**Rud92**], Isabelle/ZF [**Pau94**] et WATSON [**Cor91**].

La théorie des ensembles offre un très grand pouvoir d'expression comparable à celui des logiques d'ordre supérieur (par exemple la quantification sur les ensembles) où les types sont utilisés pour contraindre les raisonnements. Cependant le raisonnement en logique d'ordre supérieur est plus simple qu'en théorie des ensembles ; en effet les développeurs sont plus familiers avec l'emploi de fonctions et de types. A l'inverse, les modèles à états avec la théorie du raffinement encadrent un processus de développement correct commençant par les spécifications abstraites. Cela favorise le choix de la théorie des ensembles comme base de travail. Néanmoins la question n'est pas de choisir entre la théorie des types et la théorie des ensembles.

Cette question a fait l'objet il y a une quinzaine année d'un débat intéressant (LAMPORT, PAULSON [**Lam92, LP99**]). La préoccupation est plutôt de trouver un compromis pour fonder un environnement d'ingénierie logicielle. Pour l'analyse multifacette, un tel compromis peut être une logique d'ordre supérieur au dessus de la théorie des ensembles afin de bénéficier des deux approches. Une argumentation similaire est défendue dans [**Gor**]. Les possibilités offertes par la méthode B [**ACL02**] (spécifications ensemblistes, simples, à l'ordre supérieur) renforcent aussi cette préférence.

Dans cette voie, le prouveur de théorème générique Isabelle est un bon candidat car il inclut déjà des instanciations pour la logigue du premier ordre, la théorie des ensembles (ZF) et la logique d'ordre supérieur. Son caractère générique permet à l'utilisateur de particulariser la plateforme de vérification de proprités et de raisonnement. La méthode B elle, est intéressante car elle permet, en plus du raffinement, des spécifications d'ordre supérieur [**ACL02**].

2. Unités de spécification multiparadigmes (MSU) : la proposition

Une unité de spécification multiparadigme (MSU) est une structure autonome composée d'une partie donnée (espace d'états), de comportement (réactions aux événements externes/internes qui font parcourir l'espace d'états), de propriétés générales fonctionnelles et non fonctionnelles, de propriétés temporelles qui

[3] à l'aide des opérations : ensemble des parties, union, remplacement

contraignent le comportement, de contraintes graphiques qui permettent, le cas échéant, de visualiser l'unité, etc.

La partie donnée est exprimée dans un langage de type théorie des ensembles ; les données et l'espace d'états sont ainsi exprimées formellement dans ce langage. La description d'état est appelée *configuration*.

Le langage élémentaire pour décrire l'espace d'état est constitué :

- des opérateurs standard de la logique du premier ordre (*true*, *false*, $=$, $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\cdots$),
- des opérateurs standard de la théorie des ensembles ($\{\}$, $\in$, $\notin$, $\subseteq$, $\cap$, $\cup$, *domain*, *range*, $\cdots$),
- des constantes définies par l'utilisateur,
- des variables, des relations et des fonctions (f, g, r, u, $\cdots$).

Si on considère Σ comme l'ensemble des opérateurs du langage décrit ci-dessus, nous avons ainsi une algèbre de termes (T_Σ) pour décrire les états, les expressions et les propriétés relatives. Du point de vue pratique, nous avons fait le choix du *Common Logic Standard*[4] comme langage logique support. En effet il a l'avantage d'être un 'standard' (une proposition de normalisation est en cours, mais relativement bien avancée) et devrait de fait favoriser l'échange et l'interaction avec divers outils.

La partie comportement (ou dynamique) est fondée sur les systèmes de transitions ; en effet ce modèle est largement étudié et outillé. De plus il est extensible pour assurer l'intégration d'autres paradigmes. Pour prendre en compte d'autres paradigmes, les systèmes de transitions sont étendus de différentes façons ; par exemples : la hiérarchisation [**MLS97, GLL99**], les extensions au temps [**LV96, HMP91**], le traitement des aspects symboliques [**HM00**], la prise en compte de constraintes d'évolution [**PR99**].

Dans des travaux précédents (sur les Machines à Configurations, [**Att98, Att00b, Att02c**]) nous avions effectué des expériences en terme d'unités de base appelée *machines à configurations* pour spécifier des systèmes complexes. Il s'en suit que ce travail peut être généralisé aux unités multiparadigmes afin de les utiliser comme base pour l'intégration de paradigmes, langages ou méthodes.

De plus les unités de spécification multiparadigmes peuvent être utilisées pour générer des environnements de spécification et de développement selon les besoins des utilisateurs. Lorsqu'on construit des environnements de spécification et de développement, les formes abstraites des unités multiparadigmes servent de base aux opérateurs générés.

Une MSU a un espace de configurations et un comportement gardé. La garde exprime les contraintes sur l'évolution du système spécifié. L'évolution d'une MSU démarre d'une configuration initiale, se poursuit tant qu'une des gardes est vraie, et est bloquée lorsqu'aucune garde n'est vraie.

MSU : forme abstraite et sémantique. Formellement une MSU est un n-uplet $U = \langle T, V, P, C_0, \mathcal{B}, R, \mathcal{P} \rangle$ où :

- T est un ensemble fini de types introduits,
- V est un ensemble fini de variables,
- P dénote les propriétés globales,
- C_0 est la propriété de la(les) configuration(s) initiale(s),
- $\mathcal{B} = \{l_i \xrightarrow{a} r_i\}$ définit le comportement de l'unité, sous la forme de transitions gardées ; $l_i \in T_\Sigma$, $r_i \in T_\Sigma$
- R dénote les constraintes considérées pour l'unité,
- $\mathcal{P}$ représente les propriétés locales de l'unité.

[4] `cl.tamu.edu`

Les transitions gardées permettent au spécifieur de décrire les comportements voulus en exprimant les conditions nécessaires à travers les gardes. C'est le principe des commandes gardées de DIJKSTRA, qui est aussi utilisé dans les *Action Systems* de BACK et *Event B* de ABRIAL.

Pour simplifier la notation afin de faciliter la lisibilité, nous considérons trois composants dans la suite : une partie donnée $\mathcal{D} = \langle T, V, P\rangle$, une partie comportement $\mathcal{B} = \langle C_0, \mathcal{B}, R\rangle$ et une partie propriété $\mathcal{P}$ qui est $\langle\mathcal{P}\rangle$.
Ces différentes parties sont liées par les variables utilisées dans la formation des expressions et des prédicats. Ainsi nous notons explicitement $\mathcal{D} \rightleftharpoons \mathcal{B}$ pour indiquer que le comportement $\mathcal{B}$ affecte les données de $\mathcal{D}$. Lorqu'on a une MSU $M = \langle\mathcal{D}, \mathcal{B}, \mathcal{P}\rangle$, on déduit que $\mathcal{D} \rightleftharpoons \mathcal{B}$. Mais l'explicitation est nécessaire entre différentes MSU. On exprime aussi la négation avec l'opérateur idoine.

Formalisation dans d'autres cadres. Formellement, une MSU est un système de transitions gardées composable (*composable guarded transition system :CTS*) ; c'est un système dont l'évolution et la composition sont contraintes. L'aspect composition requiert des hypothèses sur l'environnement.

A partir de ce modèle de système de transitions, nous avons envisagé un plongement des *CTS* dans le cadre de la méthode B ou dans Isabelle.

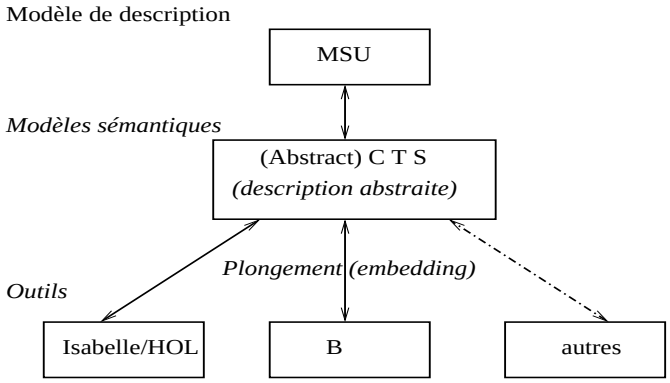

FIG. 1. Structuration d'une MSU

Dans la suite nous définissons une algèbre de MSU. Nous munissons les MSU d'un ensemble d'opérateurs pour les composer en vue de construire de grands systèmes.

Les opérateurs géneraux de SCCS/Meije [**Sim85**] nous ont en partie inspiré. En effet, ils généralisent les opérateurs communs aux algèbres de processus et ils sont très expressif pour capturer la plupart des concepts nécessaires à la prise en compte de l'interaction entre sous-systèmes. L'autre source d'inspiration est le π-calculus pour les aspects évolutivité et mobilité.

3. Conception et dévelopement de systèmes avec les MSU

On peut composer verticalement ou horizontalement les MSU pour spécifier des systèmes complexes. La composition parallèle de sous-systèmes est un exemple de composition horizontale ; l'extension de sous-système est un exemple de composition verticale.

3.1. Définitions préliminaires. Soient deux MSU $M_1 = \langle\mathcal{D}^1, \mathcal{B}^1, \mathcal{P}^1\rangle$ et $M_2 = \langle\mathcal{D}^2, \mathcal{B}^2, \mathcal{P}^2\rangle$
Nous définissons

(1) $\mathcal{D}^1 \oplus \mathcal{D}^2$ la composition des parties données ;

(2) $\mathcal{B}^1 \uplus \mathcal{B}^2$ la composition des parties comportement qui sont décrites par des systèmes de transitions gardées ; la composition résulte en la fusion des systèmes de transition ;

(3) $\mathcal{P}^1 \oslash \mathcal{P}^2$ la combinaison des propriétés ; cela résulte en la conjonction des prédicats.

Ces opérateurs de base sont ensuite définis dans le cadre de chaque opérateur de structuration, afin de les adapter aux contraintes spécifiques.

3.2. Structuration verticale. Deux opérateurs sont définis ici : $\odot$ pour l'extension d'une MSU par une autre et $\sharp$ pour le partage de données entre des MSU.

Extension. Une MSU peut être étendue pour construire une autre MSU. Dans ce cas, le composant résultant est obtenu en fusionnant la partie donnée, en augmentant la partie comportement et en augmentant la partie propriété.

$$\frac{\begin{array}{c} M_1 = \langle \mathcal{D}^1, \mathcal{B}^1, \mathcal{P}^1 \rangle \\ M_2 = \langle \mathcal{D}^2, \mathcal{B}^2, \mathcal{P}^2 \rangle \\ \mathcal{P}^1 \wedge \mathcal{P}^2 \end{array}}{M_2 \odot M_1 = \langle \mathcal{D}^1 \oplus \mathcal{D}^2, \mathcal{B}^1 \uplus \mathcal{B}^2, \mathcal{P}^1 \oslash \mathcal{P}^2 \rangle} \textit{ExtensionRule}$$

Conditions de correction : $\mathcal{P}^1 \wedge \mathcal{P}^2$; pas de contradiction entre les propriétés des MSU.

Partage. Cet opérateur permet aux MSU de partager leurs espaces d'états.

Condition de correction : $\neg\ (\mathcal{D}^1 \rightleftharpoons \mathcal{B}^2)$; le comportement d'une MSU, $\mathcal{B}^2$, n'affecte pas l'autre MSU et vice-versa.

$$\frac{\begin{array}{c} M_1 = \langle \mathcal{D}^1, \mathcal{B}^1, \mathcal{P}^1 \rangle \\ M_2 = \langle \mathcal{D}^2, \mathcal{B}^2, \mathcal{P}^2 \rangle \\ \neg\ (\mathcal{D}^1 \rightleftharpoons \mathcal{B}^2) \\ \neg\ (\mathcal{D}^2 \rightleftharpoons \mathcal{B}^1) \end{array}}{M_2 \sharp M_1 = \langle \mathcal{D}^1 \oplus \mathcal{D}^2, \mathcal{B}^1 \uplus \mathcal{B}^2, \mathcal{P}^1 \oslash \mathcal{P}^2 \rangle} \textit{SharingRule}$$

Notons que la structuration des MSU résultats est la même pour les différents opérateurs ; ce qui varie d'un opérateur à un autre est la portée des transitions et par conséquent le fonctionnement de ces transitions.

Dans le cas présent du partage, les transitions exprimées dans $\mathcal{B}^i$ ont une visibilité sur $\mathcal{D}^1 \oplus \mathcal{D}^2$.

3.3. Structuration horizontale. Les opérateurs considérés ici sont : $]\,|\,[$ pour la composition parallèle asynchrone de MSU, $\|$ pour la composition parallèle synchrone de MSU et $+$ pour le choix non-déterministe entre MSU. La structuration est la même pour les MSU résultants des compositions parallèles ; mais les comportements sont différents selon le cas de composition.

Composition asynchrone. Cet opérateur permet de composer deux ou plusieurs MSU (interagissants). Le comportement de la MSU résultante est caractérisé par l'**entrelacement** pur des comportements des composants ou leur interaction qui est basée sur des stratégies de communication décrites dans les transitions initiales composées dans le cas où les parties données sont mutuellement affectées par les parties comportements.

$$\frac{\begin{array}{c} M_1 = \langle \mathcal{D}^1, \mathcal{B}^1, \mathcal{P}^1 \rangle \\ M_2 = \langle \mathcal{D}^2, \mathcal{B}^2, \mathcal{P}^2 \rangle \\ \mathcal{P}^1 \wedge \mathcal{P}^2 \\ \mathcal{D}^1 \rightleftharpoons \mathcal{B}^2 \\ \mathcal{D}^2 \rightleftharpoons \mathcal{B}^1 \end{array}}{M_1]\,|\,[M_2 = \langle \mathcal{D}^1 \oplus \mathcal{D}^2, \mathcal{B}^1 \uplus \mathcal{B}^2, \mathcal{P}^1 \oslash \mathcal{P}^2 \rangle} \textit{ASyncCompoRule}$$

Composition synchrone. Cet opérateur aussi permet de composer deux ou plusieurs MSU. Le comportement de la MSU résultante est caractérisé par la synchronisation des comportements des composants. Leur interaction est basée sur des stratégies de communication synchrone exprimées dans les gardes des transitions.

$$\frac{\begin{array}{c} M_1 = \langle \mathcal{D}^1, \mathcal{B}^1, \mathcal{P}^1 \rangle \\ M_2 = \langle \mathcal{D}^2, \mathcal{B}^2, \mathcal{P}^2 \rangle \\ \mathcal{P}^1 \wedge \mathcal{P}^2 \\ \mathcal{D}^1 \rightleftharpoons \mathcal{B}^2 \\ \mathcal{D}^2 \rightleftharpoons \mathcal{B}^1 \end{array}}{M_1 \| M_2 = \langle \mathcal{D}^1 \oplus \mathcal{D}^2, \mathcal{B}^1 \uplus \mathcal{B}^2, \mathcal{P}^1 \circledwedge \mathcal{P}^2 \rangle} \textit{SyncCompoRule}$$

Choix non-déterministe. Cet opérateur permet un choix entre le comportement de deux ou plusieurs MSU interagissants. Le comportement du résultat est exactement le comportement d'un des composants choisis de façon non-déterministe.

$$\frac{\begin{array}{c} M_1 = \langle \mathcal{D}^1, \mathcal{B}^1, \mathcal{P}^1 \rangle \\ M_2 = \langle \mathcal{D}^2, \mathcal{B}^2, \mathcal{P}^2 \rangle \\ M = \langle \mathcal{D}^1, \mathcal{B}^1, \mathcal{P}^1 \rangle \end{array}}{M_1 + M_2 = M} \textit{ChoiceRuleL}$$

$$\frac{\begin{array}{c} M_1 = \langle \mathcal{D}^1, \mathcal{B}^1, \mathcal{P}^1 \rangle \\ M_2 = \langle \mathcal{D}^2, \mathcal{B}^2, \mathcal{P}^2 \rangle \\ M = \langle \mathcal{D}^2, \mathcal{B}^2, \mathcal{P}^2 \rangle \end{array}}{M_1 + M_2 = M} \textit{ChoiceRuleR}$$

3.4. Analyse formelle. Les unités de spécification doivent garantir/préserver des propriétés de correction. Les propriétés sont exprimées au cours de la phase de spécification. Elles doivent être vérifiées par des obligations de preuve. De plus, la composition des unités requiert des conditions de correction.

Nous avons opté pour la technique de raffinement comme technique de développement principal.
Pour outiller notre proposition nous avons choisi d'utiliser des plateformes existants. En effet, la méthode B dispose d'outils pratiques d'aide au développement formel (Atelier B, jBtools, ProB, GeneSys, B4free, Composys, Brama).

3.5. Mise en œuvre. Nous avons expérimenté l'idée des MSU, dans un premier temps comme formalisme de base pour intégrer des méthodes ou pour la traduction entre formalismes. Pour cela nous nous sommes restreint à des systèmes de transition abstraits (Abstract Transition Systems). Cette expérimentation est présentée dans le chapitre 10.

4. Perspectives

Cette voie de recherche est encore à ses débuts ; il nécessite un développement aussi bien sous les angles théoriques que pour les aspects pratiques. Dans cette direction, nous avons démarré des expérimentations avec les *Abstract Transition Systems* qui servent d'unité de spécification à un certain niveau d'abstraction, dans le cadre de l'analyse multifacette (Chapitre 10).

La limite actuelle de ces expérimentations est que les modèles induits sont figés, or nous visons à travers les MSU, des modèles multiparadigmes qui puissent être adaptés aux besoins, en intégrant différents paradigmes (plasticité des modèles). Il nous reste donc à expérimenter ces pistes dans des environnements appropriés.

Nous pensons que le cadre général de UTP (voir la section 4.5 du chapitre 3) peut servir de cadre d’études approfondies de formalisation et d’expérimentations de nos pistes.

Cinquième partie

Applications, outils et développement

Table des Matières

Au cours de nos travaux, nous avons développé et participé au développement de plusieurs prototypes logiciels ou bibliothèques pour mettre en application ou expérimenter certains aspects étudiés :

- NaBLa[5] (∇) : un ensemble de librairies de spécifications formelles en B.
- NatIF[6]/ORYX : une plateforme d'intégration de méthodes et de génération d'environnement de développement formel. Nous avons développé un prototype (ORYX) qui aide à l'analyse de cahiers de charges pour identifier le type de système (ou les paradigmes sous-jacents) puis qui génère un environnement de spécification approprié au cahier de charges.
- ATACORA : un prototype pour l'analyse formelle multifacette.
- COSTO : une plateforme de spécification et d'analyse de composants logiciels. Il s'agit de la plateforme expérimentale associée au modèle de composants (Kmelia) en cours de développement dans notre équipe.

En guise d'illustration, nous donnons dans ce mémoire un aperçu des plateformes ORYX et ATACORA.

[5] Nantes B Libraries

[6] Nantes Integration Framework

CHAPITRE 9

La plateforme NatIF/ORYX

Nous présentons ici le prototype ORYX qui sert de support à notre méthode de génération d'environnements de développement spécifiques (voir Chapitre 3). A partir de ORYX un spécifieur/développeur génère un environnement de spécification propre à son cahier de charges.

La génération d'environnement est effectuée en fonction du cahier de charges ou du type de système à développer ; cela dépend du spécifieur qui en a le choix.

Considérons deux approches qui aident au développement de systèmes. La première est une approche orientée paradigme ; ici le spécifieur met l'accent sur les paradigmes qui caractérisent son cahier de charges. L'environnement de développement adéquat peut être construit à partir de ces paradigmes. La seconde est une approche orientée langage où le spécifieur se focalise sur les langages qu'il trouve appropiés à son cahier de charges. L'environnement de développement envisagé peut être construit à partir de l'intégration de ces langages. L'intégration est effectuée sous conditions de compatibilité des langages (voir Chapitre 3).

Dans les deux cas, l'environnement finalement généré est constitué des mêmes composants ; mais la façon de construire ces composants diffèrent d'une approche à l'autre.

La figure 1 illustre la démarche de travail que préconise la plateforme ORYX.

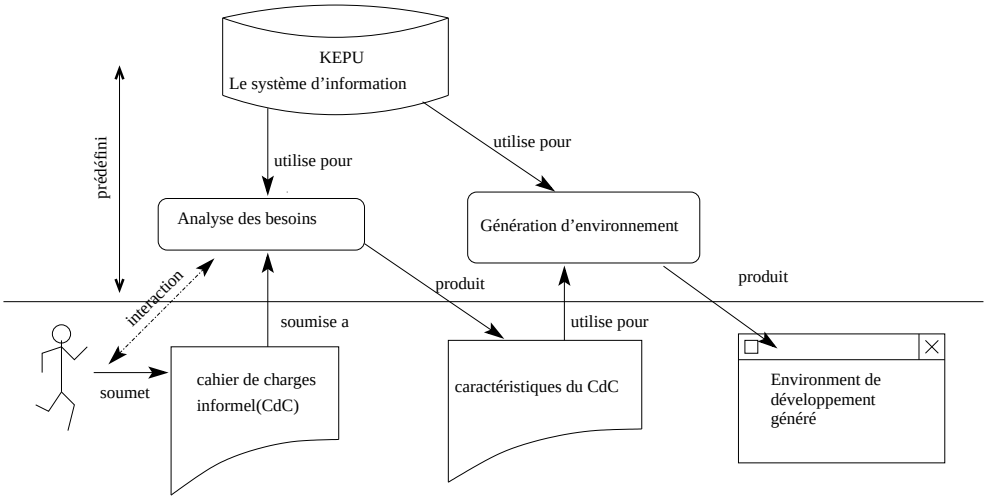

FIG. 1. Synoptique du travail sous ORYX

Le prototype ORYX composé de divers outils est conçu comme cadre expérimental de notre approche. Il comprend les constituants suivants :

– Un système d'information (KEPU) pour les méthodes d'ingénierie et la base de données associée. C'est un système d'information très générale consacré aux spécifications formelles, à l'analyse formelle et au développement. C'est le cœur du prototype.
– Un noyau d'opérateurs pour la modélisation de systèmes (SMOK). Il contient les principaux opérateurs pour divers paradigmes de spécification. Du point de vue de l'implantation, le noyau est contenu dans

les tables du système d'information KEPU ; il est utilisé pour générer les langages de spécifications abstraites.
- Un générateur d'environnement d'ingénierie de système : GENESIS. Il utilise le système d'information KEPU et les spécificités du cahier de charges en cours pour fournir à l'utilisateur un environnement d'ingénierie spécifique : un ensemble d'outils pour spécifier, analyser et développer des systèmes. En plus de l'éditeur de spécifications, diverses interfaces graphiques sont aussi générées ici.
- Un ensemble de décompilateurs (*pretty-printers*) de syntaxes abstraites de spécifications (ASP). Ils transforment les spécifications sous forme de syntaxe abstraite en diverses syntaxes concrètes.
- diverses interfaces graphiques d'utilitaires.

Le noyau SMOK est construit sur la base d'un ensemble d'opérateurs formalisés au préalable. Nous voulons le noyau interchangeable pour les expérimentations. On peut ainsi disposer de plusieurs noyaux et passer de l'un à l'autre sans changer l'environnement global ni le processus de génération.

Un système d'information pour les méthodes d'ingénierie : KEPU.
Le système d'information KEPU contient et relie : une classification de languages de spécification, une adéquation entre langages de spécification et types de systèmes des paradigmes de spécification des opérateurs associés aux paradigmes, un répertoire de langages de spécifications, un répertoire des modèles sémantiques des langages de spécification, les domaines de compatibilité de divers langages, etc ([**Att02a**]).

Une base de données est dédiée à ce système d'information. Ainsi les informations peuvent être constamment mises à jour pour refleter l'état de l'art.

Ce système d'information est utilisé par exemple pour aider le spécifieur à choisir les paradigmes appropriés à son système. Les tables correspondant aux paradigmes constituent une partie importante de la base de données ; la base contient pour cela les opérateurs (avec leurs signatures) pour construire les objets relevant des différents paradigmes (données, dynamique, etc). Chaque opérateur a sa signature.

Un générateur d'environnement d'ingénierie de système : GENESIS.
La génération d'environnement consiste à extaire de la base les types de données et les opérateurs sélectionnés avec leurs caractéristiques. Par exemple, il y a des types de données prédéfinis et des opérateurs de spécification de comportement de processus. Cet ensemble d'opérateurs forme un langage de spécification abstraite (ASSL) qui est employé par l'utilisateur. Un éditeur graphique (structurel) est construit à partir de l'ASSL.

L'environnement généré à partir de ORYX est constitué des éléments suivants :
- un language de spécification de syntaxe abstraite (ASSL) : c'est l'ensemble des opérateurs disponibles pour la spécification du système ;
- un éditeur graphique et générique : il s'agit d'un éditeur de spécification formelle orienté structure et basé sur le précédent langage ASSL ;
- des connexions vers des outils pour l'analyse formelle et le développement (preuve de propriétés, raffinement, génération de code, etc) ;
- des décompilateurs (*pretty-printers*) qui transforment les spécifications sous forme de syntaxe abstraite en diverses syntaxes concrètes (des syntaxes d'entrée des outils d'analyse externe par exemple).

Plusieurs étudiants de Master 1 (ex. Maîtrise Informatique) ont participé au développement de ce prototype dans le cadre de leurs travaux de TER (Travaux d'Etude et de Recherche) :
- CLERGEAU et GUERET, ROCARD et SALOTTI, 2000/2001,
- Julien LANDURÉ et Albin JOSSIC, Christopher HARRIS et Gael TRICAUD, 2004/2005.

CHAPITRE 10

La plateforme Atacora

Atacora est un prototype de système de spécification et d'analyse formelle multiformalisme. Avec Atacora, on peut par exemple, soumettre une spécification écrite dans un langage donné, à différents outils d'analyse formelle (pour l'analyse multifacette) en traduisant lorsque c'est possible (compatibilité sémantique) la spécification dans les formalismes d'entrée de ces outils.

Ce prototype sert de cadre expérimental pour mettre à l'épreuve les idées proposées pour une algèbre de spécifications multiparadigmes (Chapitre 8). Les MSU sont dans ce contexte représentées par des systèmes de transition abstraits (*Abstract Transition Systems* : ATS).

Le cœur du prototype Atacora est constitué :
- d'un langage de description de systèmes de transitions abstraits : ATS ;
- d'un ensemble de compilateurs/décompilateurs de spécifications. Ces derniers prennent en entrée des spécifications dans un formalisme donné, les traduisent en ATS (phase compilation) puis encodent les ATS dans un autre formalisme (phase décompilation) choisi comme sortie.
- d'un système d'informations contenant des informations sur les modèles de description et sémantiques, les langages de spécification, les techniques et outils ainsi que des informations sur la compatibilité sémantique entre ces entités ; ce système d'information est partagé avec la plate-forme Natif/ORYX.

La figure 1 donne un aperçu de la construction de passerelles entre formalismes en passant par les descriptions sous forme de ATS.

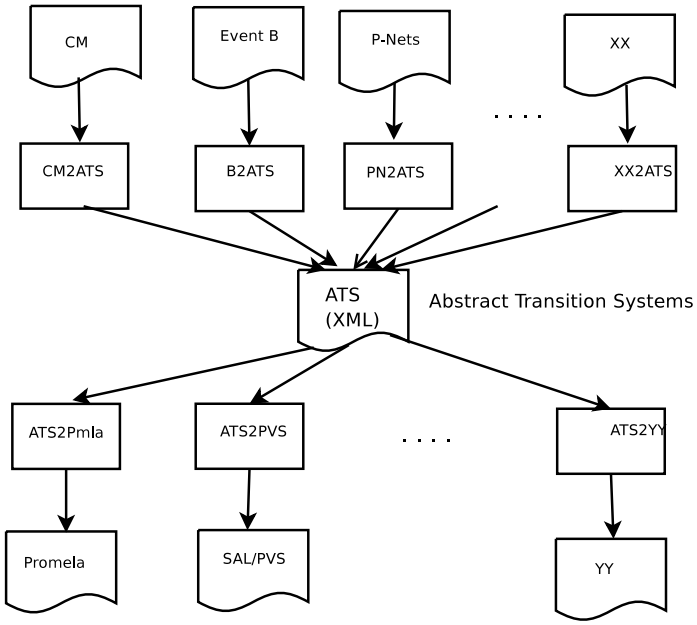

FIG. 1. Aperçu des passerelles entre spécification via ATS

Un analyseur donné transforme une spécification écrite dans un formalisme donné (l'entrée), en une description ATS (la sortie). Un décompilateur lui, transforme une description ATS (l'entrée) dans un formalisme donné (la sortie). Par exemple, pour utiliser un outil d'analyse formelle spécifique, on transformera une description ATS dans le format d'entrée de cet outil d'analyse formelle.

Dans l'état actuel du prototype, nous avons des traductions entre B (un sous-ensemble) et les réseaux de PETRI, entre une partie de B et le langage Promela. Ainsi nous pouvons utiliser, des outils dédiés à ces différents langages : Atelier B, PEP (Réseaux de PETRI), Spin (Promela) pour des analyses formelles.

La grammaire des ATS, ainsi que des exemples de passerelles construites sont accessibles sur le site `www.sciences.univ-nantes.fr` avec le lien `/info/perso/permanents/attiogbe/atacora`.

Nous avons commencé le développement d'un couple analyseur/décompilateur générique et extensible. (avec la technologie ANTLR). Il s'agit d'un analyseur paramétrable par une grammaire d'entrée qui remplacerait les N analyseurs qui sont chacun spécifique à une grammaire. La sortie par défaut est une description ATS (avec des structures stockée en XML).

Nous envisageons de traiter de la même façon la décompilation de ATS vers une grammaire de sortie qui est un paramètre.

Notons que cette expérimentation partage des points communs avec des projets comme PEPTOOLS [1] ou projet VeriTech [**Kat01, KGK02, KG02**]. Dans le cadre de VeriTech les auteurs proposent également des traductions entre différents langages de spécification et discutent la notion de *faithful relation* entre modèles : notre formalisme ATS est comparable au formalisme intermédiaire CDL (*Core description language*) proposé pour VeriTech. Les liens entre les formalismes traduits de l'un vers l'autres ne sont pas précisés, mais au vu des exemples traités, on déduit qu'il s'agit de formalisme basés sur les systèmes de transitions.

Cependant, notre proposition appuyée à la fois sur le principe de l'analyse formelle et celui d'une algèbre multiparadigme, présente l'avantage d'être d'une part assistée par un système d'information permettant d'assurer les compatibilités entre spécifications à traduire, d'autre part d'être plus général au sens de l'évolution par rapport aux aspects multiparadigmes.

Plusieurs étudiants de Master2 Professionnel ont participé avec mon encadrement, au développement du prototype Atacora, dans le cadre de leur projet développé en cours d'année.

– Sylvie CASSAN, Adrien GEFFROY, Nicolas LE NARDOU, Sébastien PRUNIER, 2004/2005
– Isabelle SÉGALEN, Manuel DUJARDIN, Arnaud VAILLANT, 2005/2006.

[1] sourceforge.net/projects/peptool

Sixième partie

Conclusions et perspectives

Table des Matières

CHAPITRE 11

Conclusions et perspectives de recherche et d'enseignement

Nous avons présenté dans ce document, en vue de l'habilitation à diriger des recherches, un ensemble de travaux parmi les thématiques de recherches qui nous ont préoccupées pendant plusieurs années. Le fil conducteur de ces travaux est l'emploi des méthodes formelles pour l'analyse et le développement de systèmes corrects. Nous avons notamment insisté sur l'intégration de méthodes formelles, des travaux autour de la méthode B et nous avons introduit l'analyse multifacette comme une méthode permettant d'effectuer l'analyse formelle de systèmes complexes.

Nous avons présenté des contributions et des résultats et évoqué aussi certaines pistes qui sont encore en exploration. Nous soulignons que nos travaux sont basés sur les travaux visant la construction rigoureuse (par des méthodes formelles appropriées) du logiciel et s'inscrivent dans la lignée de ces travaux.

Nos principales contributions concernent la systématisation de l'intégration de méthodes formelles, des extensions à la méthode formelle B et l'analyse multifacette. Les domaines d'application sont variés, du fait même des méthodes formelles ; cependant nous pouvons spécifiquement mettre en avant les systèmes concurrents et réactifs qui nous ont souvent servi de cas d'étude pour expérimenter nos pistes.

Ce document fait une synthèse qui marque une étape dans nos recherches, par conséquent les conclusions que nous tirons sont plus en terme de perspectives de travail que sous forme de clôture d'un chantier.

Une habilitation à diriger des recherches engage de façon directe à une forte implication dans l'animation scientifique : animation de projets ou de groupes de recherches, responsabilité de thèses, etc. Par conséquent les perspectives mentionnées ici vont au delà de travaux circonstanciés ; nous essayons plutôt de situer ces perspectives dans le cadre de l'animation scientifique de façon générale. De façon indirecte, les responsabilités dans l'(administration de l') enseignement sont engagées. En effet, nous sommes convaincus de l'alliance fructueuse entre l'enseignement et la recherche. Un point préoccupant est celui du profil des étudiants entreprenant des travaux de thèse de doctorat. Une bonne formation alliant les fondamentaux de la discipline et les techniques adhoc (souvent empiriques) du développement, est un paramètre tout aussi important que la motivation de l'étudiant et sa capacité à investir des terrains nouveaux. Or la pression socio-économique actuelle et les fortes demandes des entreprises et des industriels du logiciel poussent de plus en plus les programmes de nos cursus universitaires à une formation déséquilibrée entre les contenus fondamentaux et les contenus liés aux techniques pratiques.

Nous pensons qu'il faut veiller particulièrement à cet équilibre non seulement dans les formations mais également dans les offres de formation à l'Université. Nous abordons alors en guise de perspectives, non seulement des perspectives de recherches mais aussi quelques réflexions sur l'enseignement.

1. Perspectives de recherche

La ligne directrice de nos travaux reste l'approche formelle (basée sur des outils mathématiques) consistant à construire correctement les systèmes de façon modulaire, en partant de leur modèles abstraits vers des modèles concrets associés et en effectuant les preuves de correction des systèmes ainsi développés par rapport aux propriétés voulues. Dans ce cadre, nous mettons spécifiquement l'accent sur le caractère multifacette des systèmes, et la mise au point de méthodes et techniques outillées pour leur analyse et développement.

Dans ce contexte nos perspectives de recherche à moyen terme sont axées sur

- l'analyse formelle multifacette de systèmes hétérogènes (intégrant des sous-systèmes ou composants logiciels issus de formalismes/modèles divers) et
- l'étude de modèles de développement de composants logiciels ouverts (au sens interopérabilité entre divers modèles de composants).

Ces perspectives, déclinées sous différents angles, constituent en partie les grandes lignes de notre projet de recherche pour les années à venir.

L'état de l'art révèle que les logiciels autonomes ou monolitiques sont de moins en moins courants. Le logiciel tend à être, aussi bien dans sa construction que dans son utilisation, un service (une fonctionnalité spécifique) ou une ressource dans un système global incluant d'autres services logiciels ou matériels de différentes natures.

Lorsqu'on rapproche les préoccupations de recherches fondamentales de celles des applications actuelles, on note l'utilisation du logiciel comme un service, disponible partout, tout le temps et idéalement fiable. Il y a de nombreux exemples : systèmes embarqués (PDA, véhicules, automates de services,...), services multimédia, téléphonie/télévision numérique, accès aux données distribuées, ...

En dehors des aspects techniques liés à la programmation des applications, les aspects nécessitant des travaux de recherche en amont sont par exemple :

- la disponibilité de langages, d'analyseurs et compilateurs de spécifications/logiciels intégrant des propriétés de correction ;
- la fiabilité des entités logicielles utilisées pour la construction d'autres plus grandes ;
- la garantie de bonne qualité des entités logicielles ;
- l'étude et le développement d'entités logicielles intégrables dans des systèmes ouverts ;
- l'interaction entre des entités logicielles ou matérielles diverses (correction du système global malgré l'hétérogénéité de ses sous-systèmes) ;
- la garantie de correction des interactions entre les entités ;
- le raisonnement sur un système global composé de différents sous-systèmes (ou ses composants) ;
- la préservation de correction en cas d'intégration de nouvelles entités dans un contexte plus large ;
- etc

La recherche en amont doit offrir les moyens pour garantir le bon fonctionnement d'un système (intégrant du logiciel) dans des environnements de plus en plus ouverts et coopératifs du fait de l'utilisation de différents moyens de (télé)communication.

Parmi les pistes de recherche citées ci-dessus, nous nous focalisons sur :

- la modélisation formelle en vue de comprendre, d'analyser et d'expliquer le fonctionnement des systèmes (à base d'entités logicielles) ;

– l'étude de concepts, mécanismes et moyens langagiers adaptés à la spécification de systèmes complexes (*i.e.* ceux qui ne se réduisent pas à un problème connu, mais qui nécessitent une abstraction non triviale),
– l'étude de *modèles sémantiques* (structures mathématiques permettant d'interpréter les objets d'étude, par opposition aux *modèles de description* qui ne véhiculent pas d'interprétations), de leur interactions et de leur utilisation conjointe dans des développements

Ces investigations font déjà en partie les objets d'étude et d'expérimentation dans le cadre de nos travaux actuels (ORYX/Atacora) où nous nous basons sur des méthodes et systèmes existants tels que B, PVS, réseaux de Petri, SPIN, Lotos, etc.

Dans cette même otique, nous avons entamé des travaux sur le développement de composants logiciels corrects. Ici les composants logiciels sont pris au sens large. Il s'agit de l'étude et le développement d'entités logicielles, intégrables dans des systèmes ouverts ; par exemple partir de composants logiciels abstraits génériques, prouvés corrects et les adapter par raffinements successifs à différents environnements d'exécution. Pour ce faire, nous faisons des expérimentations sur un modèle abstrait de composants nommé Kmelia et nous avons quelques résultats préliminaires [**AAA05, AAA06b, AAA06a, AAA07b, AAA07a**].

Il s'agit dans le cadre de notre projet de recherches de poursuivre les études visant à trouver des méthodes pratiques outillées pour la construction correcte de (classes de) systèmes. La réutilisation de résultats eprouvés tels que les environnements associés à la méthode B (AtelierB, Projet Rodin) devrait apporter une aide considérable.

Une autre voie importante dans notre projet de recherche est le raisonnement et la maintenance en contexte hétérogène.

En effet pour faire face à l'analyse ou la maintenance d'un système complexe, une solution est l'isolation de proche en proche des modules à analyser. Par exemple, dans une interconnexion de modules formant un système, on peut étudier progressivement le système en supposant certains modules corrects puis en étudiant le système global sous ces hypothèses ; on arrive ainsi de proche en proche à identifier les impacts des différents sous-systèmes sur le système global.

De la même manière, l'analyse de propriétés globales peut se faire avec des hypothèses de correction sur les différents sous-systèmes, jusqu'à l'établissement des propriétés globales puis la levée des hypothèses faites sur les sous-systèmes.

Cette façon de faire n'est pas nouvelle, en effet c'est une démarche de preuve courante. Mais il s'agit de la mettre en œuvre avec des méthodes d'ingénierie outillées sur des systèmes hétérogènes.

La piste de réflexion principale est l'extension des techniques de la famille *Rely-Guaranty (Assume-Commitment)*.

Ces travaux de recherches ont une portée très générale dans le cadre du génie logiciel et ne peuvent aboutir que si des normes, des interfaces et des protocoles d'interaction normalisés entre systèmes logiciels/matériels sont mis au point.

Nous avons fait des études de faisabilité en nous appuyant sur des formalismes adhoc, par exemple l'usage des ATS (voir Chapitre 10) pour l'interaction entre modèles sémantiques. La généralisation de cette solution avec l'usage de la *Common Logic* (*International Standard for Common Logic* [1]) par exemple est une piste en exploration. L'idée est de se servir en plus des systèmes de transition abstraits, de la CL (ou

[1] cl.tamu.edu

autre initiative semblable de standard) comme un langage pivot pour la description de données, l'expression de propriétés de relations entre entités en vue des raisonnements.

Des initiatives similaires existent par exemple dans le domaine des algèbres de processus et celui des systèmes dynamiques et hybrides. Par exemple, les formats de fichiers pour représenter des systèmes de transition ont été proposés :

- le format FC2[2] [**MdS93, BRRdS96**] développé à l'INRIA pour servir d'interface entre des outils de vérification ;
- le *compact file format for labelled transition systems* [**vL01**] ; c'est un format d'encodage très compact de systèmes de transitions ;
- le HSIF (*Interchange Format for Hybrid Systems*) [**PCPSV06**] ; c'est un format d'échange entre des outils de conception et d'analyse de systèmes hybrides. Il est basé sur des systèmes de transitions.

A partir de ces généralisations, il est tout à fait envisageable de mettre en œuvre des interfaces formelles *standard* entre des outils divers basés sur des concepts ou des modèles sémantiques variés. Chaque outil ou environnement doit prévoir l'interconnexion avec d'autres outils, tout en préservant la sémantique, en s'adaptant simplement aux interfaces standard. De tels outils/environnements s'adaptant aux interfaces standard sont des outils/environnements ouverts.

La construction d'environnements multi-outils est ainsi à portée de main. Dans de tels environnements on peut introduire et analyser diverses spécifications sous différentes facettes mais en se référant à des modèles sémantiques standardisés. On a là un cadre fondamental pour aider au raisonnement de systèmes intégrant plusieurs modèles. Sur le plan pratique, il y a aujourdhui un engouement pour l'ingénierie dirigée par des modèles ; ce thème n'est pas nouveau (les *model-based approaches* comme VDM, Z, CSP, RdP, etc ont fait leur preuve), et il ne peut être sérieusement traité que par des approches rigoureuses formelles. Un environnement ouvert tel que nous le proposons est une solution pour les expérimentations en ingénierie de modèles.

Un des intérêts de cette interconnexion facilitée, est la complémentarité de différents outils à différents niveaux d'abstraction pour garantir la correction de systèmes informatiques de différentes natures ; ce qui démeure l'un des défis majeurs en Informatique.

Pratiquement, la stratégie que nous explorons est celle du ***Plug and Check*** (Charger et analyser) : cela veut dire qu'on doit pouvoir charger une spécification quelconque et y déclencher différentes analyses qui sont identifiées systématiquement en fonction de la spécification chargée ; ensuite d'autres analyses peuvent être entreprises via des conversions/traductions.

Nos projets de recherche sont en phase avec des sujets qui mobilisent la communauté internationnale, en terme de défis majeurs (*Grand Challenges*[**fCR03**]). La problématique de la correction des systèmes apparaît de façon récurrente dans les programmes de l'Agence Nationale de la Recherche (ACI, RNTL, etc), dans les programmes européens (PCRD) et est aussi au cœur des *Grand Challenges* internationaux[3](T. HOARE, R. MILNER, J. WOODCOCK, J. CROWCROFT, M. KWIATKOWSKA) posés à la communauté pour les 10-15 ans à venir[4]. Le problème central à résoudre à long terme est celui de disposer des outils scientifiques pouvant justifier la sûreté des systèmes informatiques (fonctionnalité, disponibilité, sécurité, fiabilité).

[2] www-sop.inria.fr/oasis/Vercors/doc/Fc2Parameterized/html/

[3] www.nesc.ac.uk/esi/events/Grand_Challenges/proposals/

[4] Voici trois exemples de ces défis en guise d'illustration et de repère : *The Verifying Compiler* (T. HOARE, ACM, 2003), *The Dependable Systems Evolution* (J. WOODCOCK, 2003) et *The Grand Challenge of Trusted Components* (B. MEYER, IEEE, 2003)

Enfin, nous aimerons signaler ici, notre intention vivace depuis quelques années de développer des *Cahiers du logiciel* comme un vecteur ou un canal d'échange pour contribuer à notre manière au défi que représente l'édification de la construction rigoureuse du logiciel dans sa diversité comme pratique courante du développement logiciel.

Ces *cahiers* vont, nous l'espérons, servir de vecteur d'idées sur le *développement* formel alliant le savoir-faire et le faire-savoir rapprochant la communauté académique et les développeurs. En effet, à la différence des revues spécialisées et des actes de conférences spécialisées qui sont des outils académiques, nous voyons les cahiers comme un outil de travail servant de référence ou de lien de travail entre académiques et praticiens du terrain. La proposition d'études de cas d'analyse formelle ou de développement formel sur des aspects variés du logiciel, avec des techniques variées est une des toutes premières pistes qui feront les premières pages des *Cahiers*.

2. Impact et perspectives en enseignement

Nos réflexions embrassent la problématique des enseignements informatiques au niveau de la Licence et du Master mais également en amont. Nous prenons donc le point de vue de la discipline plutôt qu'une focalisation sur nos enseignements de type méthodes formelles.

La place de l'Informatique à côté des autres disciplines scientifiques n'est pas encore bien délimitée. Au niveau du baccalauréat, la méconnaissance de la discipline est criante. Dans certaines sections comme *mesures physiques et informatiques* (mise en place en 2001/2002 [5]), les élèves travaillent autour de l'acquisition de données (tableur/grapheur, utilisation de logiciels dédiés).

Au delà des options de ce type, et du B2I (brevet Informatique et Internet) où il s'agit d'une auto-évaluation des élèves sur la base de la découverte des principaux concepts de fonctionnement d'un ordinateur à partir d'un livret, la perception de l'Informatique se résume à la bureautique. Cela ne favorise pas le choix par les bacheliers, de l'Informatique comme filière d'études à l'Université. A ce niveau, l'introduction de cours d'Algorithmique/programmation, de Logique et de rudiments sur les langages formels nous semble d'une importance déterminante. La construction et la diffusion de supports pédagogiques adéquats par des experts (universitaires) est indispensable à côté du militantisme pour le renforcement de la contribution de l'Université à la formation des professeurs du secondaire au niveau CAPES/CAPET/aggrégation.

La formation universitaire (post-baccalauréat de façon générale) est l'autre paramètre important pour marquer la place de la discipline Informatique.

L'enseignement des méthodes formelles pour la *construction du logiciel*, par exemple, est malheureusement une exception aujourd'hui dans la plupart des cursus universitaires en Informatique. Cependant, il n'y a pas de freins réels à donner le caractère rigoureux qui leur revient à la grande partie de nos modules d'enseignement. Pour preuve, la technologie des compilateurs est aujourd'hui relativement bien maîtrisée parce que, de notre point de vue, ce domaine a bénéficié d'enseignements 'formels' appuyés sur la théorie des langages et des automates. Les bases étant solides, l'édifice a tenu et va certainement tenir encore longtemps. Le cloisonnement entre les enseignements que nous dispensons à nos étudiants, l'instabilité du langage et des concepts enseignés sont autant de facteurs qui désservent la qualité de nos enseignements.

La réalité socio-économique pousse effectivement aux développements et à l'emploi de technologies ad hoc, pour résoudre ponctuellement ou rapidement des problèmes particuliers. Pour cela nos étudiants doivent être réactifs aux besoins des industriels mais les enseignements fondamentaux garantissant une bonne compréhension des lois fondamentales de la discipline et des bases pour son développement sont absolument nécessaires. Ces enseignements des fondamentaux, sous des formes adéquates (sans cloisonnement), doivent constituer la grande partie des offres de formation au niveau Master.

Aujourd'hui, l'enseignement des méthodes formelles se fait en tant que tel à travers des cours de logique, langages, spécification, sémantique, vérification.

Nous pensons que sur ces bases, on peut élaborer des enseignements intégrant ou reprenant tous les aspects cités au dessus et qui sont adaptés à la construction (formelle) d'un bout à l'autre de classes de systèmes informatiques. Il s'agit d'associer étroitement dans le même enseignement, les concepts fondamentaux et leur utilisation pour construire ou analyser des systèmes.

Le pari sera gagné lorsqu'on pourra enlever sans ambiguïté l'adjectif formel devant le mot construction ; on devrait comprendre que la construction ne peut être faite autrement que par des méthodes systématiques éprouvées, et qui conduisent à des résultats dont la correction est mesurée. Notons que les *curricula* actuels de l'ACM/IEEE [6] sont moins ambitieux sur ce plan.

[5] www.education.gouv.fr/bo/2000/hs6/info.htm

[6] www.acm.org/education/curricula.html

L'analyse multifacette des systèmes et leur construction formelle constituent, à notre avis des enseignements qui devraient rentrer dans des bouquets d'enseignements fondamentaux en génie logiciel. En effet, en considérant non seulement les aspects fonctionnels et non-fonctionnels des logiciels mais aussi les paramètres tels que la répartition sur le réseau, la mobilité des services, des ressources et des machines, le caractère multimode de certains services (multimédia par exemple), la prise en compte du caractère multifacette dans la conception est nécessaire non seulement pour analyser les systèmes à concevoir mais aussi pour assurer les propriétés des différentes parties du logiciel.

Il est nécessaire de disposer de concepts, techniques, méthodes et outils pour traiter l'hétérogénéité sémantique intrinsèque à la construction de systèmes non triviaux.

Les bases de tels enseignements se trouvent par exemple dans le renforcement des enseignements de logique formelle (expression et preuve de correction de propriétés et de constructions), de modèles et sémantiques (voir les HANDBOOKS [**vL90a, vL90b**]), de théories unificatrices de la programmation (HOARE, JIFENG).

L'utilisation à haute dose de l'Informatique au quotidien dans les services et dispositifs courants est peut être une aubaine dans cette direction. L'exigence de la qualité du logiciel va devenir un paramètre important pour son utilisateur. La concurrence entre les éditeurs/constructeurs de systèmes embarquant le logiciel va se faire aussi sur la correction et la fiabilité des logiciels proposés ; et se faisant le recours à des développeurs/analystes/concepteurs bien formés va devenir le cheval de bataille.

Nous maintenons la motivation et la volonté, lors de nos travaux de recherche d'alimenter nos enseignements, de proposer des expériences nouvelles et des enseignements qui contribueront à faire de la construction de logiciel une ingénierie moins empirique qu'elle ne l'est aujourd'hui et contribuer à faire la place qui lui convient à l'Informatique.

Nous avions déjà proposé en 1999 un cours sur l'intégration de méthodes formelles au niveau DEA (aujourd'hui Master2 Recherche) où nous avions abordé les problèmes d'hétérogénéité sémantique et proposé des solutions en Logique pour les aborder ; en perspective nous envisageons un enseignement sur l'analyse et la construction multifacette de systèmes réactifs (ou à événements discrets) avec les mêmes bases de logique formelles mais avec plus d'ouverture sur la construction et le raisonnement sur un système global.

De façon générale, peut-être devons nous aborder beaucoup plus tôt dans les cursus de formation, l'emploi des techniques formelles pour l'analyse des cahiers de charges, le prototypage, des études de cas comparatives, etc.

L'expérience montre à travers les travaux pratiques, les projets et les stages que les étudiants apprennent très vite et s'approprient les nouvelles technologies de développement ; par conséquent on peut s'interroger sur la nécessité d'octroyer de nombreuses heures à des modules d'enseignement de ce type. En revanche, les enseignements fondamentaux en bonne dose avec leur mise en œuvre, leur permettant de comprendre, d'analyser, d'expliquer et de s'adapter aux technologies logicielles sont plus qu'indispensables surtout avec les contraintes de réductions des heures que nous connaissons aujourd'hui dans nos cycles de Licence et de Master.

Bibliographie

[AAA05] P. André, G. Ardourel, and C. Attiogbé. Behavioural Verification of Service Composition. In *ICSOC Workshop on Engineering Service Compositions*, 2005.

[AAA06a] Pascal André, Gilles Ardourel, and Christian Attiogbé. Spécification d'architectures logicielles en Kmelia : hiérarchie de connexion et composition. In *1ère Conférence Francophone sur les Architectures Logicielles*, pages 101–118. Hermes Sciences/Lavoisier, 2006.

[AAA06b] Christian Attiogbé, Pascal André, and Gilles Ardourel. Checking Component Composability. In *5th International Symposium on Software Composition*, volume 4089 of *LNCS*, pages 18–33, 2006.

[AAA07a] Pascal André, Gilles Ardourel, and Christian Attiogbé. Adaptation for Hierarchical Components and Services. *Electronic Notes in Theoretical Computer Science (Proceedings of the Third International Workshop on Coordination and Adaption Techniques for Software Entities (WCAT 2006))*, 189 :5–20, 2007.

[AAA07b] Pascal André, Gilles Ardourel, and Christian Attiogbé. Protocoles d'utilisation de composants : spécification et analyse en Kmelia. In *(LMO'07)*, 2007.

[ABK+02] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL : The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2) :153–196, 2002.

[Abr96a] J-R. Abrial. *The B Book*. Cambridge University Press, 1996.

[Abr96b] J-R. Abrial. Extending B without Changing it (for developping distributed systems). *Proc. of the 1st Conf. on the B method, H. Habrias (editor), France*, pages 169–190, 1996.

[Abr01] J-R. Abrial. Event Driven Distributed Program Construction. MATISSE project, August 2001.

[Abr02] J-R. Abrial. Discrete System Models. Internal Notes (www-lsr.imag.fr/B), February 2002.

[ACL02] Jean-Raymond Abrial, Dominique Cansell, and Guy Lafitte. "Higher-Order" Mathematics in B. In *ZB'2002 – Formal Specification and Development in Z and B*, pages 370–393, 2002.

[ADW04] Yamine Aït Ameur, Remi Delmas, and Virginie Wiels. "a framework for heterogeneous formal modeling and compositional verification of avionics systems". In *MEMOCODE*, pages 223–232. IEEE, 2004.

[AG91] A. Asperti and G.Longo. *Categories, Types and Structures*. Foundations of Computing Series. The MIT Press, 1991. Cambridge, MA.

[AM98] J-R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In *Proc. of the 2nd Conference on the B method, D. Bert (editor)*, volume 1393 of *Lecture Notes in Computer Science*, pages 83–128. Springer-Verlag, 1998.

[APGR00] A. Arnold, G. Point, A. Griffault, and A. Rauzy. The Altarica Formalism for Describing Concurrent Systems. *Fundamenta Informaticae*, 34 :109–124, 2000.

[APS03] C. Attiogbé, P. Poizat, and G. Salaün. Specification of a Gas Station using a Formalism Integrating Formal Datatypes within State Diagrams. In *Proc. of the 8th International Workshop on Formal Methods for Parallel Programming : Theory and Applications (FMPPTA'03)*, IEEE Computer Society Press, France, 2003. International Parallel and Distributed Processing Symposium.

[APS07] C. Attiogbé, P. Poizat, and G. Salaün. A Formal and Tool-Equipped Approach for the Integration of State Diagrams and Formal Datatypes. *Transactions on Software Engineering*, 33(3) :157–170, March 2007.

[AS02] C. Attiogbé and G. Salaün. Generation of Formal Development Environment : Foundation and Application. Technical Report 02.09, IRIN, University of Nantes, December 2002.

[Att98] C. Attiogbé. Configuration Machines : A Simple Formalism For Specifying Multifaceted Systems. Technical Report 181, IRIN, University of Nantes, 1998.

[Att00a] C. Attiogbé. Formal Methods Integration : Some Locks and Outlines. Technical Report 00.8, IRIN, University of Nantes, July 2000.

[Att00b] C. Attiogbé. Le système de contrôle d'accès : une spécification à base des machines à configurations. In *Proc. of AFADL'2000*, pages 203–220, France, 2000.

[Att01] Christian Attiogbé. Refining a Cooling System of a Nuclear Power Station with the B-Method. Technical Report 01.X, IRIN, University of Nantes, 2001.

[Att02a] C. Attiogbé. A Generic Framework for Methods Integration (NatIF Project). Technical Report 02.05, IRIN, University of Nantes, July 2002.

[Att02b] C. Attiogbé. Communicating B Abstract Systems (CBS). Technical Report 02.08, IRIN, University of Nantes, December 2002.

[Att02c] C. Attiogbé. Mechanization of an Integrated Approach : Shallow Embedding into SAL/PVS. In *Proc. of ICFEM'02*, volume 2495 of *LNCS*, pages 120–131. Springer-Verlag, October 2002.

[Att04] Christian Attiogbé. Systematic Derivation of a Validation Model from a Rule-oriented Model : A System Validation Case Study using PROMELA/SPIN. In *Proc. of the IEEE ICTTA'04*, Damscus, Syria, 2004.

[Att05a] C. Attiogbé. A Stepwise Development of the Peterson's Mutual Exclusion Algorithm Using B Abstract Systems. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *Proc. of ZB'05*, volume 3455 of *LNCS*, pages 124–141. Springer, 2005.

[Att05b] C. Attiogbé. Practical Combination of Theorem Proving and Model Checking for the Multi-facet Analysis. In *Proc. of SOFSEM'05*, pages 1–10, 2005.

[Att05c] C. Attiogbé. Semantic Embedding of Petri Nets into Event-B. Technical Report SE/0510073, LINA, University of Nantes, October 2005.

[Att06a] Christian Attiogbé. Multi-process Systems Analysis using Event B : Application to Group Communication Systems. In Z. Liu and J. He, editors, *ICFEM'2006*, volume 4260 of *LNCS*, pages 660–677, 2006.

[Att06b] Christian Attiogbé. Tool-Assisted Multi-Facet Analysis of Formal Specifications (Using Alelier-B and ProB). In Peter Kokol, editor, *IASTED Conf. on Software Engineering*, pages 85–90. IASTED/ACTA Press, 2006.

[Att06c] Christian Attiogbé. Combining B Tools for Multi-Process Systems Specification. In E. Badouel, Y. Slimani, and M. Assogba, editors, *8th African Conference on Research in Computer Science (CARI'06)*, pages 35–42, 2006.

[Bac80] Ralph-Johan Back. *Correctness Preserving Program Refinements : Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, The Netherlands, 1980.

[Bac89] C. Backhouse. *Construction et vérification de programmes*. coll. MIM. Masson, Prentice-Hall, 1989.

[BB98] R. J. R. Back and M. J. Butler. Fusion and Simultaneous Execution in the Refinement Calculus. *Acta Informatica*, 35(11) :921–949, 1998.

[BBHR02] D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors. *ZB'2002 : Formal Specification and Development in Z and B, 2nd International Conference of B and Z Users, France*, volume 2272 of *LNCS*. Springer-Verlag, 2002.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[BCLE04] J-Y. Béziau, A. Costa-Leite, and A. Facchini (Editors). *Aspects of universal logic*. Institute of Logic, University of Neuchâtel, 2004.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking : 1020 States and Beyond. *Information and Computation*, 98(98) :142–170, June 1992.

[BD02] Manfred Broy and Ernst Denert, editors. *"Software pioneers : contributions to software engineering"*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[BdR97] P. Blackburn and M. de Rijke. Why combine logics ? *Studia Logica (Edited by D. Gabbay and F. Pirri)*, 59 :5–27, 1997.

[Ber98] D. Bert, editor. *B'98*, volume 1393 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[BG95] J. P. Bowen and M. J. C. Gordon. A Shallow Embedding of Z in HOL. *Information and Software Technology*, 37(5-6) :269–276, 1995.

[BG00] Didier Buchs and Nicolas Guelfi. A Formal Specification Framework for Object-Oriented Distributed Systems. *IEEE Transactions on Software Engineering*, 26(7) :635–652, 2000.

[BGG+92] R. Boulton, A. Gorgon, M.J.C. Gordon, J. Hebert, and J. van Tassel. Experience with Embedding Hardware Description Language in HOL. In *Proc. of the International Conference on Theorem Provers in Circuit Design : Theory, Practice and Experience*, pages 129–156, North-Holland, 1992. IFIP TC10/WG 10.2.

[BJK02] F. Bellegarde, J. Julliand, and O. Kouchnarenko. Synchronised Parallel Composition of Event Systems in B. In Bert et al. [**BBHR02**], pages 436–457.

[BKS83] R.J. Back and R. Kurki-Suonio. Decentralisation of Process Nets with Centralised Control. In *Proc. of the 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

[BLMF00] Jean-Michel Bruel, Johan Lilius, Ana Moreira, and Robert France. Defining precise semantics for UML. In J. Malenfant, S. Moisan, and A. Moreira, editors, *Proceedings ECOOP 2000 Workshops, Panels, and Posters, Sophia Antipolis and Cannes, France, June 2000*, volume 1964 of *LNCS*, pages 113–122. Springer, 2000.

[BM95] M. J. Butler and C. C. Morgan. Action Systems, Unbounded Nondeterminism, and Infinite Traces. *Formal Aspects of Computing*, 7 :37–53, 1995.

[Bou04] Raymond T. Boute. Integrating Formal Methods by Unifying Abstractions. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *IFM*, volume 2999 of *Lecture Notes in Computer Science*, pages 441–460. Springer, 2004.

[BPR96] D. Bert, M-L. Potet, and Y. Rouzaud. A Study on Components and Assembly Primitives in B. *Proc. of the 1st Conference on the B method, H. Habrias (editor), France*, pages 47–62, November 1996.

[BRRdS96] Amar Bouali, Annie Ressouche, Valerie Roy, and Robert de Simone. The Fc2Tools set : a Toolset for the Verification of Concurrent Systems. In R. Alur and T.A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, USA, 1996. Springer-Verlag.

[But96] M. J. Butler. Stepwise Refinement of Communicating Systems. *Science of Computer Programming*, 27(2) :139–173, 1996.

[But99] M. J. Butler. Csp2B : A practical Approach to Combining CSP and B. In J. Wing and J. Woodcock and J. Davies, editor, *Proc. of the International Confernce on Formal Methods (FM'99)*, volume 1708 of *Lecture Notes in Computer Science*, pages 490–, Toulouse, France, September 1999. Springer-Verlag.

[BW90] M. Barr and C. Wells. *Category Theory for Computer Science*. Prentice Hall, 1990.

[BW96] M. Butler and M. Walden. Distributed System Development in B. *Proc. of the 1st Conference on the B method, H. Habrias (editor), France*, pages 155–168, 1996.

[BW98] R-J. Back and J V Wright. *Refinement Calculus : A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.

[Ca86] R. L. Constable and al. *Implementing Mathematics with the Nuprl Proof Developemnt System*. Prentice Hall, 1986.

[Chr03] Christian Attiogbé and Pascal Poizat and Gwen Salaün. Integration of Formal Datatypes within State Diagrams. In *Proc. of the 6th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 2621 of *LNCS*, pages 341–355. Springer, 2003.

[CJ91] J. H. Cheng and C. B. Jones. On the Usability of Logics which Handle Partial Functions. In C. Morgan and J. C. P. Woodcock, editor, *Proceedings of 3rd Refinement Workshop*, pages 51–69. Springer-Verlag, 1991.

[CKV01] Gregory V. Chockler, Idid Keidar, and Roman Vitenberg. Group Communication Specifications : a Comprehensive Study. *ACM Comput. Surv.*, 33(4) :427–469, 2001.

[CMCHG96] E. M. Clarke, K. L. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic Model Checking. In R. Alur and T. A. Henzinger, editors, *CAV'96 : Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 419–422. Springer-Verlag, 1996.

[Cor91] F. Corella. Mechanizing Set Theory. Technical Report 232, University of Cambridge Computer Laboratory, August 1991.

[COR+95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. In *Proc. of the Workshop on Industrial-Strength Formal Specification Techniques (WIFT'95)*. IEEE Press, 1995. Florida.

[Cox86] B. Cox. *Object-oriented Programming : an Evolutionary Approach*. Addison-Wesley, 1986.

[CSS98] Carlos Caleiro, Cristina Sernadas, and Amilcar Sernadas. Parameterisation of Logics. In *Workshop on Algebraic Development Techniques*, pages 48–62, 1998.

[CSS99] C. Caleiro, C. Sernadas, and A. Sernadas. Mechanisms for Combining Logics. Technical report, Universidas de Lisboa, 1049-001 Lisboa, Portugal, 1999.

[CSS05] C. Caleiro, A. Sernadas, and C. Sernadas. Fibring logics : Past, present and future. In *We Will Show Them : Essays in Honour of Dov Gabbay*, pages 363–388. King's College Publications, 2005.

[dCH96] L. F. del Cerro and A. Herzig. Combining Classical and Intuitionist logics. In *Frontiers of Combining Systems*, pages 93–102. Kluwer Academic Publishers, 1996.

[DdB00] Sophie Dupuy and Lydie du Bousquet. A Multi-formalism Approach for the Validation of UML Models. *Formal Aspects of Computing*, 12(4) :228–230, 2000.

[DFH+93] G. Dowek, A. Felty, G. Huet, C. Murphy, C. Parent, , C. Paulin-Mohring, and B. Werner. The Coq Proof Assistant Users's Guide (5.8),. Technical Report 154, INRIA Rocquencourt, 1993.

[Dij68] E. W. Dijkstra. A constructive Approach to the Problem of Program Correctness. *BIT*, 14(8) :174–186, 1968.

[Dij72] Edsger W. Dijkstra. "chapter i : Notes on structured programming". pages 1–82, 1972.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8) :453–457, 1975.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewoods, Cliffs, NJ, 1976.

[DN66] O.-J. Dahl and K. Nygaard. SIMULA – an ALGOL-based Simulation Language. *CACM*, 9(9) :671–678, 1966.

[DN67] O.-J. Dahl and K. Nygaard. Class and Subclass Declarations. In J. Buxton, editor, *Simulation Programming Languages - IFIP Working Conference in Oslo*, IFIP, 1967.

[DS06] S. E. Dunne and W. J. Stoddart, editors. *First International Symposium on Unifying Theories of Programming*, volume 4010 of *Lecture Notes in Computer Science*. Springer, 2006.

[Dun99] S. Dunne. The Safe Machine : A New Specification Construct for B. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *Proceedings of FM'99 : World Congress on Formal Methods*, number 1709 in Lecture Notes in Computer Science (Springer-Verlag), pages 472–489. Springer Verlag, September 1999.

[Dun02] S. Dunne. A Theory of Generalised Substitutions. In Bert et al. [**BBHR02**], pages 270–290.

[Edi96] D. Powel (Guest Editor). Special Issue on Group Communications Systems. *Communications of the ACM*, 39(4), 1996.

[EM90] H. Ehrig and B. Mahr. Fundamentals of algebraic specification 2. In *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1990.

[fCR03] The Verifying Compiler : A Grand Challenge for Computing Research. . *Journal of the ACM*, 50(1) :63–69, 2003.

[Fia06] J. L. Fiadeiro. *Categories for Software Engineering*. Springer, 2006.

[FM92] Jose Luiz Fiadeiro and T. S. E. Maibaum. Temporal Theories as Modularisation Units for Concurrent System Specification. *Formal Aspects of Computing*, 4(3) :239–272, 1992.

[FM93] José Luis Fiadeiro and Tom Maibaum. "generalising interpretations between theories in the context of p-institutions". In *Proceedings of the First Imperial College Department of Computing Workshop on Theory and formal methods 1993*, pages 126–147, London, UK, 1993. Springer-Verlag.

[Gab96a] D. Gabbay. An overview of Fibred Semantics and the Combination of Logics. In *Frontiers of Combining Systems*, pages 93–102. Kluwer Academic Publishers, 1996.

[Gab96b] D. Gabbay. Fibred Semantics and the Weaving of Logics. *Journal of Symbolic Logic*, 61(4) :1057–1120, 1996.

[GB92] J. A. Goguen and R. M. Bustall. Institutions : Abstract Model for Specification and Programming. *Journal of the ACM*, 1(39), 1992.

[Gen87] H. J. Genrich. Predicate/Transition Nets. In W. Brauer, W. Reisig, and G. Rozenber, editors, *Petri Nets : Central Models and their Properties, Advances in Petri Nets(1986)*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247. Springer-Verlag, 1987.

[GHS01] Orna Grumberg, Tamir Heyman, and Assaf Schuster. Distributed Symbolic Model Checking for μ-Calculus. In *Computer Aided Verification*, pages 350–362, 2001.

[GL81] H. J. Genrich and K. Lautenbach. System Modelling with High-level Petri Nets. *TCS*, 13 :109–136, 1981.

[GLL99] Alain Girault, Bilung Lee, and Edward A. Lee. Hierarchical Finite State Machines with Multiple Concurrency Models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, 18(6) :742–760, June 1999.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. (Cambridge Tracts in T.C.S. 7). Cambridge University Press, 1989. ISBN 0 521 37181 3 xi+176pp.

[Gor] Mike Gordon. Set Theory, Higher Order Logic or Both ? University of Cambridge Computer Laboratory.

[Gor93] M.J.C. Gordon. *Introduction to HOL : A Theorem Proving Environment*. Cambridge University Press, 1993.

[GP98] A. W. Gravell and C. H. Pratten. Embedding a Formal Notation : Experiences of Automating the Embedding of Z in the Higher Order Logic of PVS and HOL. In [**J. 98**], pages 73–84, 1998.

[Gri81] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[HH98] C. A. R. Hoare and J. He. *Unifying theories of programming*. Prentice-Hall, NJ, 1998.

[HJGP99] Wai-Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. UMLAUT : an extendible UML transformation framework. In *Proc. of the Automated Software Engineering (ASE'99)*, 1999.

[HM00] T. A. Henzinger and R. Majumdar. A Classification of Symbolic Transition Systems. In *Proc. of STACS 2000*, volume 1770 of *LNCS*, pages 13–34. Springer-Verlag, 2000.

[HMP91] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Timed Transition Systems. In *REX Workshop*, pages 226–251, 1991.

[Hoa69] C. A. R. Hoare. "an axiomatic basis for computer programming". *Commun. ACM*, 12(10) :576–580, 1969.

[Hoa76] C. A. R. Hoare. Proof of correctness of data representation. In *Language Hierarchies and Interfaces, International Summer School*, pages 183–193, London, UK, 1976. Springer-Verlag.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, NJ, 1985.

[J. 98] J. Grundy and M. Newey, editor. *Supplementary Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics : Emerging Trends, (TPHOL'98)*. Australian National University, 1998.

[Jen86] K. Jensen. Coloured Petri Nets and the Invariant Method. *TCS*, 14 :317–336, 1986.

[Jen96] K. Jensen. Coloured Petri Nets Vol. I-III. In *EATCS Monographs on Theoretical Computer Science*, EATCS. Springer-Verlag, 1992-1996.

[JM94] C. B. Jones and C. A. Middelburg. A Typed Logic of Partial Functions Reconstructed Classically. *Acta Informatica*, 31(5) :399–430, 1994.

[JM04] Christophe Joubert and Radu Mateescu. "distributed on-the-fly equivalence checking". In *Proceedings of the 3rd Workshop on Parallel and Distributed Methods in Verification PDMC'2004 (London, U.K.)*, 2004.

[JM06] C. Joubert and R. Mateescu. Distributed On-the-Fly Model Checking and Test Case Generation. In *Proceedings of the 13th International Workshop on Model Checking of Software SPIN'2006 (Vienna, Austria)*, volume 3925 of *LNCS*, pages 126–145. Springer, 2006.

[Jon83] C. B. Jones. "tentative steps toward a development method for interfering programs". *ACM Trans. Program. Lang. Syst.*, 5(4) :596–619, 1983.

[Kat01] S. Katz. "faithful translations among models and specifications". In *In Proceedings of the International Symposium of Formal Methods Europe*, pages 419–434. Springer-Verlag, 2001.

[KG02] S. Katz and O. Grumberg. A Framework for Translating Models and Specifications. In *Proceedings of Integrated Formal Methods (IFM'2002)*, volume 2335 of *LNCS*, pages 145–164, UK, May 2002. Springer-Verlag.

[KGK02] K. Korenblat, O. Grumberg, and S. Katz. Translations between textual transition systems and Petri nets. In *Proceedings of Integrated Formal Methods (IFM'2002)*, volume 2335 of *LNCS*, pages 339–359, UK, May 2002. Springer-Verlag.

[KJ04] L. M. Kristensen and K. Jensen. Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad-hoc Networks. In *Proceedings of INT'04*, volume 3147 of *LNCS*, pages 248–269. Springer-Verlag, 2004.

[KJJ04] Lars Michael Kristensen, Jens Bæk Jørgensen, and Kurt Jensen. Application of Coloured Petri Nets in System Development. volume 3098 of *LNCS*, pages 626–685. Springer-Verlag, Jan 2004.

[Knu97] D. Knuth. *The Art of Programming (Volume 1-3)*. TAOCP. Addison-Wesley, 1997.

[KR00] H. Kirchner and C. Ringeissen, editors. *Frontiers of Combining Systems*, volume 1794 of *LNCS*. Springer-Verlag, 2000.

[Kra99] M. Kracht. Tools and Techniques in Modals Logic. *Studies in Logic and the Foundations of Mathematics*, 142, 1999.

[KW97] M. Kracht and F. Wolter. Simulation and transfer results in modal logic. *Studia Logica*, 59(2) :149–177, 1997.

[Lam92] L. Lamport. Types Considered Harmful. Technical report, manuscript, December 1992.

[LB03] M. Leuschel and M. Butler. ProB : A Model Checker for B. In Keijiro A., Stefania G., and Dino M., editors, *FME 2003 : Formal Methods Europe*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[LDSM01] Jing Liu, Jin Song Dong, Kun Shi, and Brendan Mahony. Linking UML with Integrated Formal Techniques. In Keng Siau and Terry Halpin, editors, *Unified Modeling Language : Systems Analysis, Design and Development Issues*, chapter 13, pages 210–223. Idea Publishing Group, 2001.

[LMM99] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In *Proc. FMOODS'99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, February 15-18, 1999*. Kluwer, 1999.

[LOT88] ISO LOTOS. *A Formal Description Technique Based on The Temporal Ordering of Observational Behaviour*. International Organisation for Standardization - Information Processing Systems - Open Systems Interconnection, Geneva, 1988. International Standard 8807.

[LP99] L. Lamport and L. C. Paulson. Should Your Specification Language Be Typed ? *ACMTOPLAS : ACM Transactions on Programming Languages and Systems*, 21, 1999.

[LSC03] Hung Ledang, Jeanine Souquières, and Sébastien Charles. Argo/UML B : un outil de transformation systématique de spécification UML en B. In *AFADL'2003*, pages 3–18, IRISA Rennes – France, January 2003. AFADL2003, IRISA, IRISA.

[LT05] M. Leuschel and E. Turner. Visualizing Larger State Spaces in ProB. In *Proc. of ZB'05*, volume 3455 of *LNCS*, pages 6–23. Springer-Verlag, April 2005.

[LV96] Nancy A. Lynch and Frits W. Vaandrager. Action Transducers and Timed Automata. *Formal Aspects of Computing*, 8(5) :499–538, 1996.

[MdS93] E. Madelaine and R. de Simone. The FC2 Reference Manual. Technical report, Inria, 1993.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, 1980.

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

[MKa91] d F. Wolter M. Kracht a. Properties of Independently Axiomatizable Bimodal Logics. *Journal of Symbolic Logic*, 56(4) :1469–1485, 1991.

[ML04] Hans Wolfgang Loidl Martin Lange. Parallel and Symbolic Model Checking for Fixpoint Logic with Chop. In *Proceedings of the 3rd Workshop on Parallel and Distributed Methods in Verification PDMC'2004 (London, U.K.)*, 2004.

[MLS97] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In R. K. Shyamasundar and Kazunori Ueda, editors, *ASIAN*, volume 1345 of *Lecture Notes in Computer Science*, pages –. Springer, 1997.

[Mon96] Jean-François Monin. *Comprendre les méthodes formelles*. coll. CNET-ENST. Masson, Paris, 1996.

[Mor90] Caroll Morgan. *Programming from Specification*. Prentice-Hall, NJ, 1990.

[Mos03] T. Mossakowski. Foundations of Heterogeneous Specification. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002*, LNCS, pages 359–375, 2003.

[Mos05] Till Mossakowski. Heterogeneous Theories and the Heterogeneous Tool Set. In Y. Kalfoglou, M. Schorlemmer, A. Sheth, S. Staab, and M. Uschold, editors, *Semantic Interoperability and Integration*, number 04391 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germ, 2005.

[MPS+06] Igor Melatti, Robert Palmer, Geoffrey Sawaya, Yu Yang, Robert M. Kirby, and Ganesh Gopalakrishnan. "parallel and distributed model checking in eddy". In Antti Valmari, editor, *SPIN*, volume 3925 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2006.

[MPW92] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Journal of Information and Computation*, 100, 1992.

[MR99] C. Muñoz and J. Rushby. Structural Embeddings : Mechanization with Method. In J. Wing and J. Woodcock, editor, *Proc. of the World Congress in Formal Methods (FM99)*, volume 1708 of *Lecture Notes in Computer Science*, pages 452–471, France, 1999. Springer-Verlag.

[Mur89] T. Murata. Petri-Nets : Properties, Analysis and Applications. In *Proc. IEEE*, volume 77, pages 541–580. IEEE, 1989.

[MV90] S. Mauw and G.J. Veltink. A Process Specification Formalism. *Fundamenta Informaticae*, pages 85–139, 1990.

[MW97] Pierre Michel and Virginie Wiels. A Framework for Modular Formal Specification and Verification. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97 : Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313, pages 533–552. Springer-Verlag, 1997.

[NGV94] R. P. Nederpelt, J. H. Geuvers, and R. C. De Vrijer, editors. *Selected Papers on Authomath*, volume 133 of *Studies in Logic and The Foundations of Mathematics*, 1994.

[Obj01] Object Management Group. OMG UML Specification V1.5. Technical report, OMG, March 2001.

[OBY03] Vadim Okun, Paul E. Black, and Yaacov Yesha. Testing with Model Checker : Insuring Fault Visibility. *WSEAS Transactions on Systems*, 2(1) :77–82, January 2003.

[Par72] D. L. Parnas. A Technique for Software Module Specification with Examples. *CACM*, 15(5), May 1972.

[Par75] D. L. Parnas. "the influence of software structure on reliability". In *Proceedings of the international conference on Reliable software*, pages 358–362, New York, NY, USA, 1975. ACM Press.

[Pau88] Lawrence C. Paulson. A Formulation of the Simple Theory of Types (for Isabelle). In P. Martin-Löf and G. Mints, editors, *Proc. of the International Conference on Computer Logic COLOG'88*, volume 417 of *Lecture Notes in Computer Science*, pages 246–274, Tallinn, Estonia, December 1988. Springer-Verlag.

[Pau93a] Lawrence C. Paulson. Introduction to Isabelle. Technical Report 280, University of Cambridge, Computer Laboratory, 1993.

[Pau93b] Lawrence C. Paulson. Isabelle's object-logics. Technical Report 286, University of Cambridge, Computer Laboratory, 1993.

[Pau93c] Lawrence C. Paulson. Set Theory for Verification : I. From Foundations to Functions. *Journal of Automated Reasoning*, 11(3) :353–389, 1993.

[Pau94] Lawrence C. Paulson. *Isabelle : A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[PCPSV06] A. Pinto, L. Carloni, R. Passerone, and A. Sangiovanni-Vincentelli. Interchange Format for Hybrid Systems : Abstract Semantics. In J. Hespanha and A. Tiwari, editors, *Proc. of Hybrid Systems : Computation and Control (HSCC 2006)*, volume 3927 of *LNCS*, pages 491–506. Springer, 2006.

[PM95] D. L. Parnas and J. Madey. Functional Documents for Computer Systems. *Science of Computer Programming*, 25(1) :41–61, 1995.

[PPS06] F. Patin, G. Pouzancre, and T. Servat. Approche formelle pour la réalisation d un système sécuritaire de contrôle commande de façades de quais. *4ème Conférence Annuelle d Ingénierie Système (AFIS'06)*, 2006.

[PR99] G. Point and A. Rauzy. Altarica - Constraint Automata as a Description Language. *Journal Européen des Systèmes Automatisés*, 33(8-9) :1033–1052, 1999.

[PS02] A. Papatsaras and B. Stoddart. Global and Communicating State Machine Models in Event Driven B : A Simple Railway Case Study. In Bert et al. [**BBHR02**], pages 458–476.

[RCA01] Gianna Reggio, Maura Cerioli, and Egidio Astesiano. Towards a rigorous semantics of UML supporting its multi-view approach. In Heinrich Hussmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001, Proceedings*, volume 2029 of *LNCS*, pages 171–186. Springer, 2001.

[Rei91] Wolfgang Reisig. Petri Nets and Algebraic Specifications. *Theor. Comput. Sci.*, 80(1) :1–34, 1991.

[Rei98] W. Reisig. *Elements of Distributed Algorithms : Modeling and Analysis with Petri Nets*. Springer, 1998.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

[Rob99] Jason Elliot Robbins. *Cognitive Support Features for Software Development Tools*. PhD thesis, University of California, Irvine, 1999.

[Ros98] A.W. Roscoe. *The Theory and Practice of concurrency*. Prentice-Hall, 1998.

[RR98] W. Reisig and G. Rozenberg, editors. *Lectures on Petri nets I : Basic models and II : Applications*, volume 1491/1492 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[RR99] G. M. Reed and A. W. Roscoe. The Timed Failures-Stability Model for CSP. *Theor. Comput. Sci.*, 211(1-2) :85–127, 1999.

[Rud92] Piotr Rudnicki. An Overview of the Mizar Project. Technical report, University of Alberta, 1992.

[SA04] G. Salaun and C. Attiogbé. MIAOw : a Method to Integrate a Process Algebra with Formal Data. *Informatica*, 28(2) :207–219, 2004.

[SAA02a] G. Salaün, M. Allemand, and C. Attiogbé. A Method to Combine any Process Algebra with an Algebraic Specification Language : the π-Calculus Example. In *Proc. of the 26th Annual International Computer Software and Applications Conference (COMPSAC'02)*, IEEE Computer Society Press, England, 2002.

[SAA02b] Gwen Salaün, Michel Allemand, and Christian Attiogbé. Specification of an Access Control System with a Formalism Combining CCS and CASL. In *IPDPS*. IEEE Computer Society, 2002.

[SB06] C. Snook and M. Butler. UML-B : Formal modelling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 15(1) :92–122, 2006.

[Sch00] Klaus-Dieter Schewe. UML : A modern dinosaur ? A critical analysis of the Unified Modelling Language. In H. Kangassalo, H. Jaakkola, and E. Kawaguchi, editors, *Proc. 10th European-Japanese Conference on Information Modelling and Knowledge Bases, Saariselkä (Finland), 2000*. IOS Press, Amsterdam, 2000.

[SDGS98] W. J. Stoddart, S. E. Dunne, A. J. Galloway, and R. Shore. Abstract State Machines : Designing Distributed Systems with State Machines and B. In *Proc.* [**Ber98**], 1998.

[SG99] Anthony J. H. Simons and Ian Graham. 30 Things That Go Wrong in Object Modelling with UML 1.3. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 221–242. Kluwer, Dordrecht, 1999.

[Sha00] N. Shankar. Combining Theorem Proving and Model Checking through Symbolic Analysis. In *Proc. of CONCUR'00*, volume 1877 of *LNCS*, pages 1–16. Springer-Verlag, 2000.

[Sho92] J. R. Shoenfiel. Axioms of Set Theory. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 2, pages 321–344. 1992.

[Sim85] R. De Simone. Higher-level Synchronizing Sevices in Meije-SCCS. *TCS*, 37 :245–267, 1985.

[SKP+93] M. Saaltink, S. Kromodimoeljo, B. Pase, D. Craigen, and I. Meisels. An EVES Data Abstraction Example. In J. C. P. Woodcock and P. G. Larsen, editors, *Proc. of Formal Methods Europe : Industrial-Strength Formal Methods (FME'93)*, volume 670 of *Lecture Notes in Computer Science*, pages 578–596, 1993.

[SSC97] A. Sernadas, C. Sernadas, and C. Caleiro. Synchronization of logics. *Studia Logica*, 59(2) :217–247, 1997.

[ST02] S. Schneider and H. Treharne. Communicating B Machines. In Bert et al. [**BBHR02**], pages 416–435.

[SU98] Morten Heine. Sørensen and P. Urzyczyn. Lectures on the curry-howard isomorphism. Available as DIKU Rapport 98/14, 1998.

[SU06] Morten Heine Sørensen and Pawel Urzyczyn. Lectures on the curry-howard isomorphism. *Studies in Logic and the Foundations of Mathematics*, 149, 2006.

[SW96] C. Seguin and V. Wiels. Using a Logical and a Categorical Approach for the Validation of Fault-tolerant Systems. In *Proceedings of Formal Methods Europe FME'96*, volume 1051, pages –. Springer-Verlag, 1996.

[SZ02] E. Sekerenski and R. Zurob. Translating Statecharts to B. In *Proc. of the Integrated Formal Methods (IFM'2002)*, volume 2335 of *Lecture Notes in Computer Science*, UK, May 2002. Springer-Verlag.

[vL90a] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume A : Algorithms and Complexity*. Elsevier and MIT Press, 1990.

[vL90b] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics*. Elsevier and MIT Press, 1990.

[vL01] I. van Langevelde. A Compact File Format for Labeled Transition Systems. Technical Report R0102, 2001.

[WE98] Virginie Wiels and Steve Easterbrook. Management of evolving specifications using category theory. In *Automated Software Engineering*, pages 12–21, 1998.

[Wir71a] Niklaus Wirth. "program development by stepwise refinement". *Commun. ACM*, 14(4) :221–227, 1971.

[Wir71b] Niklaus Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4) :221–227, 1971.

[XdRH97] Q. Xu, W. P. de Roever, and J. He. The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. *Formal Aspects of Computing*, 9(2) :149–174, 1997.

[XS98] Q. Xu and M. Swarup. Compositional Reasoning Using the Assumption-Commitment Paradigm. *Lecture Notes in Computer Science*, 1536 :565–583, 1998.