UNIVERSITE DE NICE-SOPHIA ANTIPOLIS - UFR Sciences

École Doctorale de Sciences et Technologies de l'Information et de la Communication

THESE

pour obtenir le titre de

Docteur en Sciences

de l'UNIVERSITE de Nice-Sophia Antipolis

Discipline : Informatique

présentée et soutenue par

Tomás BARROS

FORMAL SPECIFICATION AND VERIFICATION OF DISTRIBUTED COMPONENT SYSTEMS

Thèse dirigée par **Isabelle ATTALI**

soutenue le 25 novembre 2005

Jury:

Président du Jury	Laurence PIERRE	Université de Nice - Sophia Antipolis
Rapporteurs	Ana Rosa CAVALLI	Institut National des Telecommunications, France
	František PLASIL	Charles University, Prague
Examinateurs	Fréderic LANG	INRIA - Grenoble
	Luis MATEU	Universidad de Chile, Chili
Co-directeur de thèse	Eric MADELAINE	INRIA - Sophia Antipolis

à Isabelle

Contents

List of Figures

Acknowledgements

I would like to thank both the French Embassy at Chile and the Chilean Commission in Research and Technology (Conicyt) who granted me with a scholarship for doing my studies in France.

I'm specially thankful to my advisor Eric Madelaine, who demonstrated an amazing dedication guiding me through my studies. His advises were incommensurable helpful.

To the reviewers and the juries, I would like to thank their efforts. Hopefully it did not turn out to be extremely boring.

I'm fully grateful to INRIA, the Oasis team, and its former members. Not only did they welcome me, but for doing it in a very kind manner.

It is also my privilege to thank all those who shared their friendship along these last three years. My dear friends, your support has been far beyond professional, and absolutely indispensable.

Finally, I would like to remember Isabelle Attali who unfortunately passed away last year. She welcomed me from the very beginning and supported me enormously on this adventure. Her exemplary kindness will always be with me.

Part I

Résumé étendu en français (Extended french abstract)

1 Introduction

Je suis certain que de nombreuses personnes, les lecteurs de cette thèse compris, jureraient derrière l'écran comme l'a fait Richard Sharpe dans [150] quand Windows s'écrase.

> *S'ils faisaient des routes, des ponts, des voitures, des avions et des bateaux comme ce logiciel, la race humaine serait condamnée*

Pourtant, des logiciels sont maintenant au coeur de ces produits. Pourquoi les avions Airbus ne s'écrasent pas aussi souvent que le logiciel Windows? Pourquoi les voitures du métro de Paris ne s'écrasent pas entre elles tous les jours?

La réponse est en partie parce que ces logiciels critiques sont développés en utilisant des méthodes formelles. Les concepteurs des avions utilisent des mathématiques pour modéliser les systèmes complexes nécessaires pour maintenir un avion Airbus en vol. Les concepteurs de ponts utilisent des mathématiques pour estimer la charge sur les matériaux utilisés dans leurs constructions.

Les Méthodes Formelles forment la base mathématique du logiciel. Une méthode est formelle si elle a une base mathématique solide, normalement donnée à l'aide d'un langage de spécification formel. Cette base fournit des moyens pour définir précisément des notions comme la cohérence et la complétude et, plus important, la spécification, l'implantation et la correction. Elle fournit aussi des moyens pour démontrer qu'une spécification est réalisable, qu'un système a été correctement implanté et pour prouver des propriétés d'un système sans nécessairement le faire tourner.

C'est l'utilisation de mathématiques pour la spécification, la modélisation, le développement et le raisonnement sur les logiciels qui est à la fois la force et la faiblesse des approches formelles. Une faiblesse parcequ'il n'y a pas aujourd'hui assez de développeurs avec une base solide en mathématiques pour se sentir confortables avec les notations des méthodes formelles. Pire encore parceque la vérification formelle d'un système est une tâche difficile. Prouver qu'un système à une certaine propriété est souvent un problème indécidable. Çela est inévitable puisque les systèmes sont con cus pour exécuter des cycles infinis et qu'ils manipulent des ensembles de données non bornées (e.g. des nombres réels, des entièrs non bornés ou des mesures de temps). Il est alors nécessaire, en utilisent des techniques complexes d'abstraction [57], d'approcher les systèmes à l'aide de modèles discrêts et finis, sur lesquels on peut utiliser les algorithmes existants, efficaces pour la vérification [68].

Voici la motivation de cette thèse : nous voulons fournir des méthodes et des outils pour que les développeurs, qui ne sont pas nécessairement des experts dans le domaine des méthodes formelles, puissent vérifier la correction des systèmes d'une façon simple et directe sans entrer dans les détails complexes des techniques de modélisation et de preuve.

Les méthodes formelles bénéficient d'une littérature très riche, et le choix d'une méthode spécifique varie beaucoup selon le système cible ou le type de leur application. Notre travail est focalisé en particulier sur les systèmes construits avec des composants distribués.

Les méthodes formelles ont été appliquées avec succès dans plusieurs domaines, dont la conception des circuits, des systèmes embarqués, des logiciels synchrones ou dans certaines applications de temps réel entre autres. Dans le domaine des systèmes distribués les problèmes sont plus difficiles, la complexité introduite par le calcul parallèle et les communications asynchrones peuvent produire des comportements non désirés et rendre l'analyse et la vérification beaucoup plus difficiles.

L'auteur de cette thèse appartient à l'équipe OASIS [3] à l'INRIA Sophia-Antipolis. L'équipe OASIS concentre beaucoup de ressources pour le développement d'un intergiciel nommé ProActive [28, 46] qui permet la construction des applications en utilisant des objets Java distribués. Dernièrement, ProActive fourni aussi des moyens pour développer des applications en utilisent des composants [29].

La programmation par composants aide considérablement la conception, l'implantation et la maintenance des logiciels complexes. L'implantation des composants de ProActive, nommé Fractive [29], est basé sur le modèle proposé par Fractal [43] en y ajoutant les caractéristiques propres à ProActive (i.e. les composants distribuées et les communications asynchrones). Nos méthodes et outils pour la vérification formelles sont focalisés particulièrement sur des applications construites en utilisant Fractive, mais elles peuvent être utilisées également dans d'autres systèmes distribués comme vous pouvez le constater à la lecture du manuscrit complète de cette thèse.

1.1 Les méthodes formelles

Comment nous avons dit précédemment, une méthode formelle utilise les mathématiques pour spécifier, développer et raisonner sur les systèmes logiciels. Une méthode formelle adresse aussi des questions pragmatiques : qui l'utilise, sur quoi elle est utilisée, quand et comment elle est utilisée.

Les méthodes formelles peuvent être utilisées dans toutes les étapes du développement. Elles peuvent êtres utilisées de la définition des besoins du client, à travers la conception du système, l'implantation, le débogage, l'entretien, la vérification et jusqu'à l'évaluation.

Les méthodes formelles permettent de révéler des ambiguïtés, des problèmes de non-complétude et d'inconsistance. Quand elles sont utilisées tôt dans le processus de développement, elles peuvent découvrir des failles de conception qui autrement seront, peut-être, découvertes seulement avec de couteuses étapes de preuve et de débogage. Quand elles sont utilisées plus tard dans le cycle, elles peuvent aider à déterminer la correction de l'implantation d'un système et l'équivalence entre implantations différentes.

Les aspects particuliers qui peuvent êtres décrits par les méthodes formelles peuvent varier considérablement pour chaque méthode [20]. Grosso modo, nous pouvons distinguer deux familles de méthodes formelles : les méthodes *déductives* et les méthodes *basées sur des modèles*.

Dans les méthodes *déductives*, la correction des systèmes est définie par des propriétés dans une théorie mathématique. Le problème de la vérification est exprimé comme un théorème de la forme : *spécification du système* $\Rightarrow$ *propriété cherchée*. Établir ce résultat est connu comme le *theorem proving*.

Les méthodes *basées sur des modèles*, comme ce nom le suggère, sont basées sur

des modèles décrivant le comportement du système d'une façon mathématique précise et non-ambigüe. Ces modèles sont accompagnés par des algorithmes qui explorent systématiquement tous les états (tous les scénarios possibles du système) du modèle. De cette façon on montre qu'une certaine propriété est vraie dans le système, technique connue sous le nom de *model-checking*.

Vérification de modèles (Model Checking) La vérification de modèles est une technique qui consiste à construire un modèle fini d'un système et vérifier qu'une propriété cherchée est vraie dans ce modèle. Il y a deux façons générales de vérification dans le *model-checking* : vérifier qu'une propriété exprimée dans une logique temporelle est vraie dans le système, ou comparer (en utilisant une relation d'équivalence ou de préordre) le système avec une spécification pour vérifier si le système correspond à la spécification ou non.

Au contraire du theorem proving, le model-checking est complètement automatique et rapide. Il produit aussi des contre-exemples qui représentent des erreurs subtiles dans la conception et ainsi il peut être utilisé pour aider le débogage.

Preuve de théorm̀es (Theorem proving) La preuve de théorèmes est une technique où le système et les propriétés recherchées sont exprimés comme des formules dans une logique mathématique. Cette logique est décrite par un système formel qui défini un ensemble d'axiomes et de règles de déduction. La preuve de théoreme est le processus de recherche de la preuve d'une propriété à partir des axiomes du système. Les étapes pendant la preuve font appel aux axiomes et aux règles, ainsi qu'aux définitions et lemmes qui ont été possiblement dérivés.

Au contraire du model-checking, le theorem proving peut s'utiliser avec des espaces d'états infinis, à l'aide de techniques comme l'induction structurelle. Son principal inconvénient est que le processus de vérification est normalement lent, sujet à l'erreur, demande beaucoup de travail et des utilisateurs très spécialisés avec beaucoup d'expertise.

Au cause de cet inconvénient et parce que notre travail cible des outils automatiques et faciles à utiliser, nous préférons naturellement nous baser sur des techniques de model-checking.

1.2 Les mythes autour des méthodes formelles

L'utilisation des méthodes formelles a une longue histoire. Bien qu'il y ait eu une utilisation significative des méthodes formelles dans les industries critiques [94], elles n'ont pas été très bien accueillies en général par la communauté de développement de logiciels [35].

Cette situation ne doit pas surprendre puisque les méthodes formelles sont largement perçues comme une collection de notations souvent à l'état de prototype, des outils qui sont difficiles à utiliser, et qui ne passent pas à l'échelle facilement. Il y a eu plein d'idées fausses à propos des méthodes formelles ; Anthony Hall a écrit un article en citant sept mythes à leur sujet [90], nous reprenons deux d'entre eux :

1. *Les méthodes formelles peuvent garantir qu'un logiciel est parfait.* Rien peut garantir la perfection. Les méthodes formelles ne sont pas la panacée pour la fiabilité des

systèmes, mais elles peuvent considérablement l'améliorer. Elles doivent êtres vues comme des méthodes puissantes et complémentaires aux autres techniques déjà connues et utilisées comme le test et le débogage. En particulier, pour l'approche basée sur des modèles, nous ne devons jamais oublier le fait que :

> *Toute vérification basée sur le modèle d'un système est au mieux aussi précise que le modèle lui-même*

En conséquence cette thèse est principalement centrée sur la modélisation des systèmes. Sa première partie cherche à trouver le meilleur format pour décrire le comportement de systèmes, et sa deuxième partie profite de la sémantique et de la structure des composants pour construire, de la façon la plus automatique possible, les modèles des ces systèmes.

2. *Les méthodes formelles demandent des spécialistes mathématiques très entraînés*

   Il y a eu beaucoup d'efforts pour rendre l'utilisation des méthodes formelles plus faciles (au moins pour certains systèmes spécifiques), et nous ne sommes pas d'accord avec A. Hall. La méthode formelle choisie peut requérir de fortes connaissances mathématiques, dépendant du formalisme, le système cible ou la propriété à prouver. Simplement choisir la méthode formelle la plus adéquate peut déjà demander une connaissance de base des méthodes formelles. Mais le but est de fournir des outils automatiques qui cachent la complexité des algorithmes de vérification de modèles, et qui évitent totalement la difficulté d'intèraction requise par un prouveur de théorèmes.

   Pour nos systèmes cibles, en addition à l'automatisation, nous voulons cacher les notations, les logiques et les algorithmes complexes derrière des interfaces conviviales pour l'utilisateur. En général nous essayons d'adapter les approches déjà connues et utilisées pour la description de systèmes (par exemple le langage de description d'architecture (ADL) comme nous le montrons dans les chapitres 4 et 5 du manuscrit complet) pour la modélisation et vérification.

1.3 Les systèmes distribués

Le model-checking est une technique puissante, largement utilisée pour vérifier le matériel, les systèmes embarqués et les logiciels séquentiels ou à mémoire partagée. En général les spécifications sont exprimées dans une logique temporelle propositionnelle (comme CTL [68]) et le système est représenté comme un graphe de *transitions d'états* (connu sur le nom de structure de Kripke)

Cependant, quand on travaille avec des systèmes distribués, i.e. des processus concurrents et communicants, les modèles basés sur les états ne sont pas bien adéquats. En l'absence d'une mémoire partagé, où les états peuvent êtres facilement identifiés par les états des variables du système, il est difficile (sinon impossible) d'identifier l'état actuelle d'un système distribué et partant de son modèle. Par ailleurs, la concurrence ajoute au modèle de l'entrelacement et de l'indéterminisme, ce qui augmente exponentiellement la taille des modèles basés sur les états.

D'un autre coté, dans les systèmes concurrents et communicants, il est plus facile de distinguer les actions que chaque processus peut exécuter à un moment donné, en

particulier des actions qui peuvent représenter des communications entre processus. Les actions communicantes devront être éxécutées au même temps (synchronisées) dans tout le processus qui participe à la communication. La forme pour modéliser un processus où nous observons les actions possibles à exécuter mais pas les états, est connu comme les *systèmes de transitions étiquetés (Labelled Transition Systems = LTSs).*

Notez que l'approche LTS n'observe pas la nature des processus (comme les états de ses variables) mais plutôt ce qu'ils peuvent exécuter. Cette vision a mené, à partir des travaux séminaux de Milner sur CCS [130] et d'Hoare sur CSP [96], vers une famille riche de calculs qui permettent de raisonner sur le comportement des systèmes concurrents et communicants, connue comme *les algèbres de processus.*

Nous voulons utiliser les algèbres de processus pour modéliser le comportement de nos systèmes. Être basés sur ce cadre nous permet de profiter de notions comme les relations d'équivalences et de congruences, permettant une conception modulaire, ansi que des abstractions menant à des modèles plus petits en préservant leur sémantique.

1.4 La programmation par composants

La programmation par composants a émergé comme une méthodologie de programmation pour des systèmes complexes qui garanti la ré-utilisabilité et la compositionalité. De façon général, un composant est une entité autonome qui interagit avec son environnement à travers des interfaces bien définies. En dehors de ces interactions, un composant ne révèle pas sa structure interne.

Plusieurs modèles de composant ont été proposés [145, 43, 64, 8] dont quelques-uns sont actuellement utilisés dans l'industrie. Ils partagent tous des caractéristiques en commun, comme l'encapsulation, et quelques uns ont des caractéristiques complémentaires comme la composition hiérarchique et la distribution.

Dans les modèles hiérarchiques comme Fractal [43], des composants différents peuvent être assemblés pour devenir eux-mêmes un nouveau composant utilisable dans le prochain niveau de la hiérarchie. Les composants fournissent aussi une séparation entre des aspects fonctionnels et non-fonctionnels. Parmis les aspects non-fonctionnels les plus intéressants citons le cycle de vie et les capacités de reconfiguration, qui permettent le contrôle de l'exécution d'un composant aussi comme son évolution dynamique. Les composant hiérarchiques cachent, à chaque niveau, la complexité de leur structure interne.

Le but principal de ce travail de thèse est de construire un cadre formel pour garantir que les compositions sont bien faites quand le système est déployé. Nous nous concentrons sur Fractive [29], une implantation distribuée du modèle de composants Fractal [43] utilisant l'intergiciel ProActive [28].

1.5 Les hypothèses initiales de notre travail et ses buts

Nous avons précisé précédemment la nécessité de fiabilité dans les logiciels et nous avons introduit les méthodes formelles comme une technique puissante pour répondre à cette nécessité. Nous avons mentionné quelques causes pour lesquelles les méthodes formelles n'ont pas été tout à fait adoptées dans la production de logiciel et, en même temps, nous avons décrit les caractéristiques qu'une solution devrait avoir pour attaquer ces causes

(automatisation et convivialité). Nous avons aussi définit nos systèmes cibles, étant les systèmes construits avec des composants distribués, et nous avons pris quelques décisions initiales sur la meilleure méthode formelle à utiliser (algèbres de processus et systèmes de transitions étiquetés).

Notre but est de garantir qu'une application, construite avec des composants distribués, soit sûr au sens que ses parties s'assemblent correctement, et fonctionnent en harmonie. Chacun des composants doit correspondre au rôle que le reste du système attend de lui, et la mise à jour ou le remplacement d'un composant ne doit pas causer le blocage ou l'échec pour les autres.

La notion habituelle de compatibilité de types des interfaces n'est pas suffisante pour cela; nous devons exprimer l'interaction dynamique entre les composants pour éviter les blocages ou les comportements incorrects du système.

Le défi est de construire un cadre formel avec des méthodes et outils pour garantir non seulement que l'assemblage est sûr quand il est déployé, mais aussi en présence de changements dynamiques et reconfigurations ultérieures. Ce cadre devrait être en même temps assez formel pour qu'il soit utilisable par des outils formels, mais assez simple pour qu'il soit utilisable par des non-spécialistes; les outils devront être le plus automatique possible et elles devront cacher les logiques et algorithmes complexes.

Structure de la thèse

Le manuscrit complet de cette thèse est écrit globalement dans l'espoir d'être autocontenu et nous recommandons de la lire dans l'ordre. Ceci parce que plusieurs idées et concepts dans ses parties dépendent fortement des chapitres précèdants pour avoir une meilleure compréhension.

Dans le chapitre 2 nous révisons les travaux principaux sur les algèbres de processus qui sont important pour notre sujet. Ensuite nous révisons les langages et outils de description de comportement les plus connus nés de ces algèbres. A la fin du chapitre 2 nous faisons une révision des travaux actuelles sur la vérification de composants.

Dans le chapitre 3 nous introduisons un nouveau format intermédiaire qui est une adaptation du travail de Lin, nommé *symbolic transition graphs with assignment* [111], et du travail d'Arnold, nommé *synchronisation automata networks* [15] : Nous étendons la notion générale de systèmes de transitions étiquetées (LTS) et des réseaux hiérarchiques de processus communicants (synchronisation networks) pour ajouter des paramètres dans les évènements de communication. Ces évènements peuvent avoir des conditions sur ses paramètres. Les processus peuvent être aussi paramétrés pour représenter des ensembles de processus équivalents qui s'exécutent en parallèle. Les résultats de ce travail ont été présentés dans [21, 26, 25, 17].

Dans le chapitre 4 nous utilisons notre format intermédiaire pour spécifier le comportement de composants hiérarchiques. Le comportement fonctionnel de composants primitives peut être obtenu en utilisant des outils d'analyse de code source, ou exprimé par le développeur dans un langage de spécification. Ensuite, nous incorporons automatiquement les comportements non-fonctionnels dans un contrôleur construit à partir de la description du composant. La sémantique d'un composant est donc obtenu comme le produit des LTSs de ses sous-composants avec le contrôleur. Le système résultant est

vérifié contre des propriétés exprimées dans des logiques temporelles ou par des LTSs. Les résultats de ce travail ont été présentés dans [22, 23].

Dans le chapitre 5 nous avons fait un grand pas en avant en passant aux composants distribués de Fractive : ces membranes ont un unique fil d'exécution non-préemtif qui sert, basé sur différentes politiques, les appels de méthodes de sa queue de requètes. Les appels vers des autres composants sont fait en utilisant une phase de rendez-vous qui garanti la délivrance et l'ordre des appels. Les réponses sont toujours asynchrones avec des références futures; la synchronisation est fait à l'aide d'un mécanisme *d'attente-par-nécessite*.

De façon similaire aux composants synchrones, nous ajoutons automatiquement le comportement non-fonctionnel des composants dans ses contrôleurs obtenus à partir de la description du composant. En addition, nous ajoutons les caractéristiques de Fractive en incorporant des automates qui représentent les queues, les réponses futures et les politiques de service des appels (en particulier les politiques par défaut en charge de servir les appels fonctionnels versus non-fonctionnels en fonction du cycle de vie du composant). Finalement nous montrons comment notre approche nous permet de vérifier des propriétés pendant toutes les étapes de la vie d'un composant en incluant les interactions entre évènements fonctionnels et non-fonctionnels : par exemple la réponse inévitable des appels des méthodes asynchrones même si le composant est dans une phase de reconfiguration. Les résultats de ce travail ont été publiés dans [24].

Dans le chapitre 6 nous introduisons les outils que nous avons développés pour travailler avec notre approche et finalement le chapitre 7 introduit les conclusions de nos travaux ainsi que des pistes de travaux futurs.

2 Spécification de comportements

Dans les systèmes de vérification comportementaux, la première question naturelle qui se pose est de savoir quel est le langage/format le plus adapté pour modeler son comportement.

Nos systèmes cibles sont des applications réelles, concurrentes, distribuées asynchrones construites en utilisant des composants. Nous recherchons le langage le plus adéquat pour décrire le comportement de tels systèmes.

Nous voulons aussi que ce langage soit basé sur les théories des algèbres de processus [130, 96, 33] qui nous permettront de tirer profit de leur principales caractéristiques : *sémantiques opérationnelles* pour décrire sans ambiguïté le comportement du système et vérifier ses propriétés, *équivalences et pré-ordres* qui donnent des relations comportementales entre des systèmes différents et *conception compositionnelle* pour modeler des systèmes plus larges à partir de petites pièces. Le dernier aspect est important, lorsque qu'on s'attache à la capacité de passage à l'échelle.

De plus, nous souhaitons que ce langage soit à la fois assez intuitif et expressif pour spécifier le comportement de notre système cible, ainsi que le langage cible d'outils d'analyse statique (ensuite, l'implantation pourra être validée par les spécifications formelles). Il faut aussi qu'il soit assez formel, en tant que langage d'entrée pour des outils de model-checking.

Comme nous discutons dans le manuscrit complet, les langages existants ne sont

pas très adéquats pour nos besoins. Parmi les plus connus, Promela [85] (le langage d'entrée de l'outil SPIN [98, 97]) ne supporte pas les descriptions compositionnelles, ou LOTOS [104] qui est riche en expressivité, mais, en effet, trop riche pour se soumettre à des procédures des décision automatiques essentielles à notre approche.

Notre approche est une adaptation de *symbolic transition graphs with assignment* de [111] avec les *synchronisation automata networks* de [15] : nous étendons la notion générale de systèmes de transitions étiquetées (LTS) et de réseaux hiérarchiques de processus communicants (synchronisation networks) pour ajouter des paramètres dans les évènements de communication, à la manière de [111]. Nous avons nommé le langage résultant *Réseaux paramétrés d'automates communicants*.

2.1 Réseaux paramétrés d'automates communicants

Nous modélisons le comportement d'un processus comme un système de transitions étiquetées paramétré (Parameterized Labelled Transition System, pLTS). Nous utilisons des paramètres à la foiswaussi bien pour l'encodage de données de messages que pour manipuler des familles indexées de processus.

Nous utilisons alors un réseau paramétré pour synchroniser un nombre fini de processus. Un réseau synchronisé paramétré (Parameterized Synchronisation Network, pNet) est une forme d'opérateur parallèle généralisé, où chacun de ses arguments est typé par l'ensemble de ses actions possibles observables.

Les actions à synchroniser entre les arguments du réseau sont encodées dans un automate nommé *transducer*. Un état dans le transducer défini un ensemble particulier de synchronisations, et un changement d'état dans le transducer introduit un nouvel ensemble de synchronisations, i.e. il modélise une reconfiguration dynamique. Un réseau avec un état unique est appelé un réseau *statique*.

Syntaxe Concrète

Nous avons développé une syntaxe concrète pour nos réseaux paramétrés d'automates communicants basé sur le format FC2 [36, 118], qui nous appelons FC2Parameterized.

Le format FC2 permet la description de systèmes de transitions étiquetés (LTSs). Les LTSs sont des tableaux des états et les états eux-mêmes deviennent des tableaux des transitions avec états cibles indexés. Les réseaux sont des vecteurs avec des références aux autres sous-processus (autres tableaux) plus des vecteurs de synchronisation qui dénotent une combinaison des actions des différents processus qui devront se synchronisés. Les sous-processus sont eux-mêmes aussi des réseaux permettant ainsi la description hiérarchique.

Le format FC2 permet aussi d'être étendu avec des nouveaux opérateurs et d'ajouter des informations complémentaires dans les états et transitions avec le label *hook*. Nous avons défini la syntaxe de FC2Parameterized, étant toujours dans la syntaxe FC2, en utilisant ces extensions et labels *hook* pour introduire des paramètres dans les états et transitions, ainsi que de nouveaux opérateurs pour gérer ces paramétres.

La syntaxe de FC2 et de sa version étendu FC2Parameterized sont introduites dans le chapitre 6 du manuscrit complet.

Syntaxe graphique

Nous introduisons aussi une syntaxe graphique pour représenter les réseaux *statiques* paramétrés, qui est un compromis entre l'expressivité et la convivialité. Le but principale de cette syntaxe graphique est de mieux introduire notre approche aux lecteurs.

Nous utilisons une syntaxe graphique similaire à celle de l'éditeur Autographe [10], augmenté par des éléments pour les paramètres et les variables : un *pLTS* est dessiné comme un ensemble de cercles et de flèches, représentant respectivement les états et les transitions. Les états sont étiquetés par l'ensemble des variables associées ($\overrightarrow{v_s}$), et les transitions par $[b]\ \alpha(\overrightarrow{x}) \rightarrow \overrightarrow{e}$.

Un *pNet statique* est représenté par un ensemble de boîtes nommées, chacune encodant un argument du pNet, plus une boîte englobante. Ces boîtes (arguments) peuvent être remplies par un pLTS satisfaisant la condition d'inclusion. Chaque boîte a un nombre fini de *ports* sur ses arêtes, représentés par des bulles nommées, chacune encodant une action paramétrée particulière de l'argument.

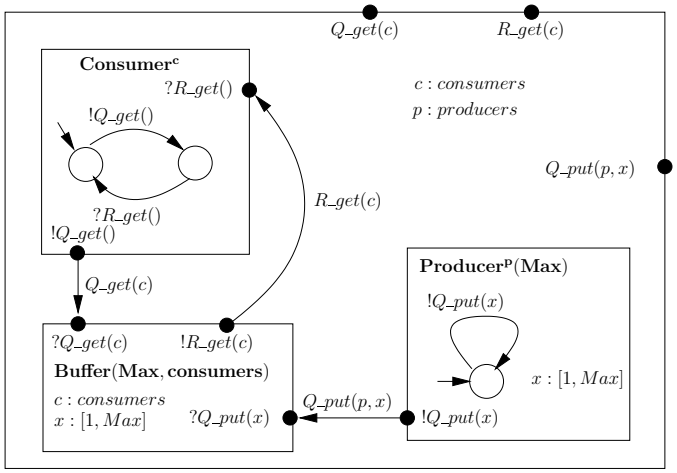


Figure i: Système paramétré consommateur-producteur

La figure i montre un exemple d'un tel système paramétré. Il est composé d'un unique buffer borné (avec une capacité Max) et une quantité bornée de consommateurs ($\#consumerss$) et de producteurs ($\#producers$). Chaque producteur rempli le buffer avec une quantité (x) d'éléments à la fois. Chaque consommateur ne réclame qu'un seul élément du buffer à la fois ($!Q\_get()$) et attend la réponse ($?R\_get()$).

La figure i introduit aussi la notation pour encoder l'ensemble des processus; par exemple, **Consumer$^{\mathbf{c}}$** encode un ensemble composé d'un processus **Consumer** pour chaque valeur du domaine c. Ainsi, chaque élément du domaine de c est associé (identifié) à un unique **Consumer** de l'ensemble.

Les lignes entre les ports de la figure i sont appelées des liens. Les liens d'un réseau expriment la synchronisation entre des boîtes internes ou à des processus externes. Ils peuvent aussi être entre des ports de différentes instances de la même boîte. Une synchronisation peut être requise entre plus de deux actions (i.e entre plus de deux ports). Ceci est représenté par une ellipse avec de multiples liens entrant/sortant vers les ports de boîtes (processus) dont les actions doivent être réalisées simultanément (un

exemple est montré dans la figure 3.5 du manuscrit complet). Chaque lien encode une transition entre le transducer LTS du *pNet*.

Quand l'état initial est paramétré par une expression, il peut lui être indiqué quelle évaluation de l'expression (valeurs des variables) doit être considérée comme l'état initial.

2.2 Instanciation

Étant donnée une abstraction finie des paramètres du modèle, nous avons introduit dans [21] une procédure automatique qui produit un instanciation finie (hiérarchique) des pLTS et pNet. Une fois l'instanciation réalisée, nous pouvons générer le produit de synchronisation, qui est un LTS encodant le comportement complet du réseau lorsque l'on synchronise les actions de ses processus, comme défini dans le transducer. Comme le résultat du produit de synchronisation est un LTS lui-même, il peut être utilisé comme un argument de la définition d'un Net supérieur. En d'autres mots, nous offrons la possibilité de composer hiérarchiquement des processus.

Nous avons développé un outil, FC2Instantiate, pour instancier automatiquement nos systèmes paramétrés.

3 Composants hiérarchiques distribués

3.1 Introduction

La programmation par composants hérite d'une longe expertise sur les modèles, les objets et les interfaces. Le modèle de composants Fractal [43] fournit des compositions hiérarchiques pour une meilleure structure, et des spécifications d'interfaces de contrôle pour l'entretien dynamique. Les différentes interfaces de contrôle permettent le contrôle de l'exécution d'un composant et de son évolution dynamique : ajouter et enlever des composants dynamiquement permet l'adaptabilité et l'entretien. En particulier, les composants distribués doivent s'adapter à leur environnement.

Fractive [29] est une implantation du modèle Fractal utilisant l'intergiciel ProActive [47]. Nous ciblons la spécification et vérification de composants construits avec Fractive qui sont *distribués*, *hiérarchiques*, *asynchrones* et capables de se reconfigurer dynamiquement. Le défit est de construire un cadre formel qui garantisse la composition initial d'un système (conception et implantation) et son évolution (entretien et adaptation). Notre utilisateur cible est donc le développeur en charge de ces tâches. Ce cadre doit cacher le plus possible la complexité du processus de vérification et il devrait fonctionner en sa majoritéle plus automatiquement possible.

Quelques travaux sur la spécification de comportement de composants, comme Wright [13], Darwin [120] ou Sofa [146] ont été proposés. Ils sont tous basés sur des relations d'équivalences ou de raffinement des aspects fonctionnel du système pour garantir la composition correcte. Au contraire de notre approche (que nous détaillons ensuite), ils ne prennent pas en compte les aspects non-fonctionnels des composants qui peuvent affecter considérablement le comportement du système, par exemple en l'arrêtant ou en redéfinissant les liens entre ses composants.

Notre approche est d'exprimer le comportements des composants sous forme de réseaux hiérarchiques d'automates communicants. Le comportement fonctionnel des composants de base (primitives) peuvent être dérivés, comme nous avons décrit dans [21], avec des outils d'analyse de code source, ou bien exprimés avec un langage de spécification. Ensuite, nous incorporons automatiquement le comportement non-fonctionnels dans un contrôleur construit à partir de la description du composant. La sémantique d'un composant est obtenu comme le produit synchronisé des LTSs de ses sous-composants avec le contrôleur. Le système résultant peut être vérifié contre des propriétés exprimées avec des logiques temporelles ou avec des LTSs.

3.2 Contexte

Nous traitons ici de systèmes de composants construits avec *Fractive*. Fractive est une implantation de Fractal basée sur l'intergiciel ProActive [47]. Elle présente donc un modèle de composants ayant les mêmes fonctionnalités que ProActive, les principaux étant les appels de méthodes asynchrones, l'absence de mémoire partagée, une politique de service configurable par l'utilisateur, et la transparence de la distribution et de la migration.

Fractal

Un composant Fractal est formé de deux parties : un *contrôleur* (ou membrane) en un *contenu*. Fig. ii présente un exemple d'un système de composants Fractal.

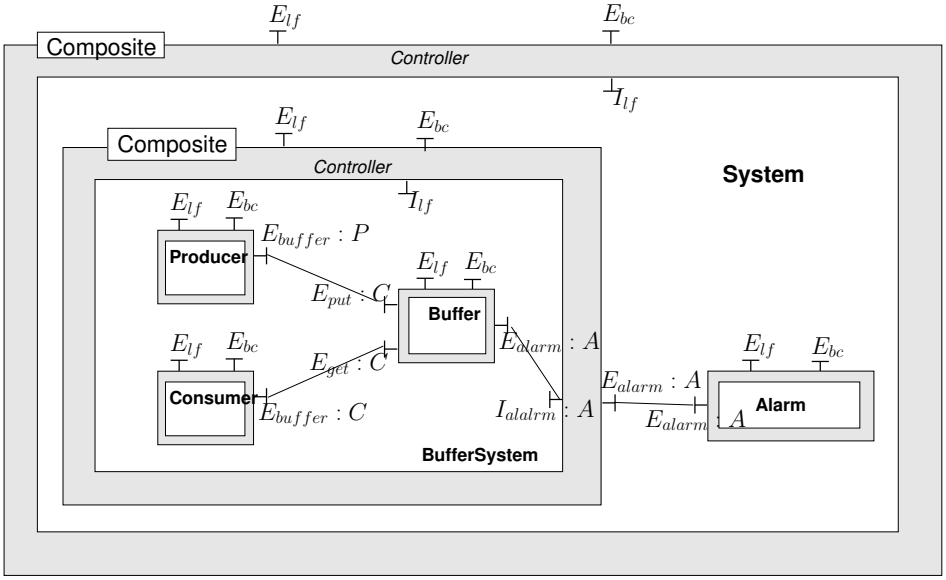


Figure ii: Un exemple d'un système à composant avec Fractal

La membrane d'un composant peut avoir des interfaces *externes* (e.g., E in Fig. ii) et *internes* (e.g., I in Fig. ii). Un composant peut interagir avec son environnement par l'intermédiaire d''*opérations* avec ses interfaces externes, alors que les interfaces internes ne sont accessibles que par les sous-composants.

Les interfaces peuvent avoir deux rôles : *client* ou *serveur*. Une interface serveur reçoit des invocations de méthodes alors qu'une interface cliente émet des appels de

méthodes. Une interface *fonctionnelle* propose ou requiert les fonctionnalités d'un composant, alors qu'une interface de contrôle correspond à une fonctionnalité de gestion pour l'architecture du composant. Fractal définit 4 types d'interfaces de contrôle : *contrôle de liaison*, pour lier/délier les interfaces clientes (e.g. E_{bc} dans Fig. ii) ; *contrôle de cycle de vie*, pour arrêter et démarrer le composant (e.g. E_{lf} dans Fig. ii) ; *contrôle de contenu* pour ajouter/ enlever/mettre à jour des sous-composants, et *contrôle d'attribut* pour définir/récupérer la valeur d'attributs internes.

ProActive

ProActive est une implantation en Java d'objets actifs distribués avec appels de méthodes asynchrones et retours de résultats par références futures. Une application distribuée construite avec ProActive est composée de plusieurs *activités*, chacune ayant un point d'entrée différent, l'*objet actif*, accessible depuis n'importe où. Les autres objets d'une activité (appelés *objets passifs*) ne peuvent être référencés directement depuis l'extérieur. Chaque activité possède son propre et unique fil d'exécution de service et le programmeur décide de l'ordre dans lequel les requêtes sont servies en redéfinissant la méthode `runActive` (point d'entrée de l'activité). Les appels de méthodes à des objets actifs se comportent de la manière suivante :

1. Lorsqu'un objet effectue un appel de méthode vers un objet actif (e.g., $y = \mathbf{O}_B.m(\vec{x})$), l'appel est déposé dans la queue de requêtes de l'objet appelé et une référence future est créée et retournée (y référence f). Une référence future contient la promesse du retour d'un appel de méthode asynchrone.

2. A un moment donné, l'activité appelée décide de servir la requête. La requête est extraite de la queue et la méthode est exécutée.

3. Une fois la méthode terminée, son résultat est mis à jour, i.e. la référence future f) est remplacée par le résultat concret de l'appel de méthode (`la valeur de` y).

Lorsqu'un fil d'exécution tente d'accéder à un futur avant qu'il ait été mis à jour, il est bloqué jusqu'à ce que la mise à jour ait lieu (*attente par nécessité*). Le calcul ASP [46] a été défini afin de fournir une sémantique formelle pour ProActive.

Fractive

Fractive est l'implantation de Fractal basée sur ProActive. Certains aspects sont ouverts dans la spécification de Fractal, et peuvent donc être redéfinis dans une implantation donnée de Fractal, ou bien peuvent être laissés à redéfinir par l'utilisateur. Fractive fait le choix d'actions démarrer/arrêter récursives, i.e. elles agissent sur le composant et chacun de ses sous-composants, depuis le haut vers le bas.

Composants Primitifs Un composant primitif dans Fractive est constitué d'une activité dont l'objet actif implante les interfaces proposées. Les requêtes fonctionnelles tout comme les requêtes de contrôle sont déposées dans la queue de requêtes de l'objet actif. Un composant primitif Fractive se comporte de la manière suivante :

1. Lorsqu'il est arrêté, seules les requêtes de contrôle sont servies.

2. Démarrer un composant primitif signifie démarrer la méthode `runActive` de son objet actif

3. Arrêter un objet actif signifie sortir de la méthode `runActive` Puisque les objets actifs sont non-préemptifs, la sortie de la méthode `runActive` ne peut être forcée : les requêtes d'arrêt sont signalées en attribuant la valeur faux à la variable locale `isActive` ; ensuite, la méthode `runActive` devrait se terminer.

Composites Fractive implante la membrane d'un composite comme étant un objet actif, par conséquent il contient une queue de requêtes unique et un seul fil d'exécution de service. Les requêtes vers ses interfaces serveuses externes (y compris les requêtes de contrôle) et depuis ses interfaces clientes internes sont déposées dans sa queue de requêtes. Une vue graphique de n'importe quel composite Fractive est présentée dans la Figure iii.

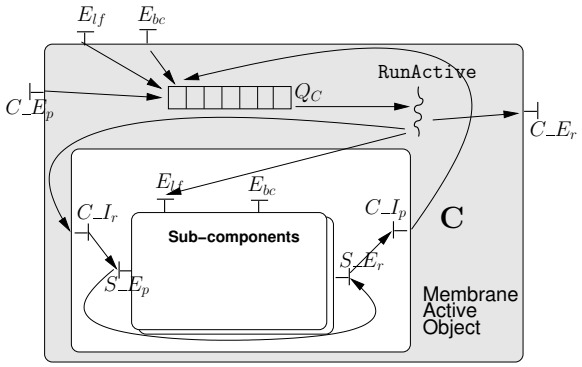


Figure iii: Un composite de Fractive

Le fil d'exécution de service sert les requêtes dans un ordre FIFO mais ne sert que les requêtes de contrôle lorsque le composant est arrêté. Par conséquent, un composite *arrêté* n'émettra pas d'appels fonctionnels sur ses interfaces clientes, même si ses sous-composants sont actifs et envoient des requêtes sur ses interfaces internes. Servir une requête fonctionnelle sur une interface serveuse interne signifie transférer l'appel vers l'interface cliente externe correspondante du composite. Servir une requête fonctionnelle sur une interface serveuse externe consiste à transférer l'appel vers l'interface interne cliente correspondante du composite.

3.3 Modèles de Comportement

Le coeur de notre travail consiste en la synthèse d'un modèle comportemental de chaque composant, sous la forme d'un ensemble de LTSs qui utilisent notre formalisme paramétré. Dans [21] nous avons montré comment construire le comportement des activités ProActive ; cela correspond exactement à la partie fonctionnelle du comportement de composants primitifs de Fractive.

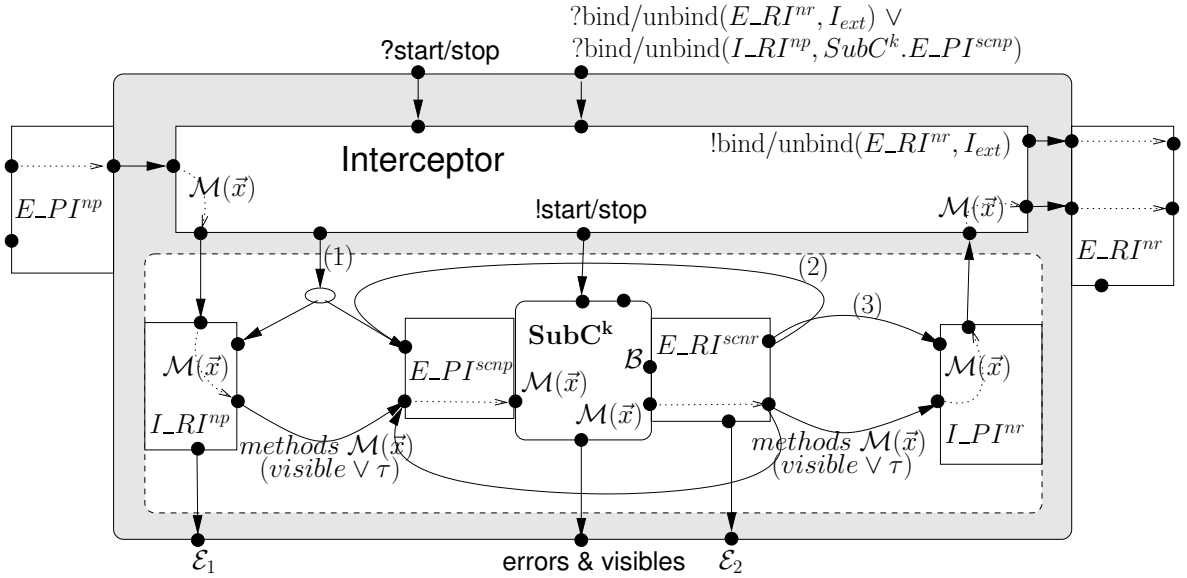
Étant donné le comportement fonctionnel d'un composant primitif, ou d'un sous-composant d'un composite, nous extrayons de sa description architecturale les informations requises pour générer les LTSs qui encodent ses fonctionnalités de contrôle (cycle de vie et attachement). La sémantique d'un composant est alors calculée comme le

produit synchrone de toutes ses parties, et est nommé l'automate *contrôleur* du composant.

La construction est faite depuis le bas et vers le haut de la hiérarchie. A chaque niveau, c'est à dire pour chaque composite, une phase de déploiement est appliquée. Le déploiement est une séquence d'opérations de contrôle, exprimé par un automate, qui se termine par une action de réussie $\surd$. Un déploiement réussi est vérifié par l'analyse d'accessibilité de l'action $\surd$ sur l'automate obtenu par le produit de synchronisation du contrôleur de composant et de ses déploiements.

Nous définissons l'automate *statique* d'un composant comme étant le produit synchronisé de l'automate contrôleur avec l'automate de déploiement, en cachant les actions de contrôle, en oubliant toute autre reconfiguration, et réduit par équivalence faible (weak bisimulation). Lorsque l'on n'est pas intéressé par les reconfigurations, l'automate statique devient le LTS encodant le comportement de ce sous-composant au niveau suivant de la hiérarchie.

La figure iv montre la structure générique du contrôleur pour un composant Fractive à n'importe quel niveau de la hiérarchie.



(1) !bind/unbind($I\_RI^{np}, SubC^k.E\_PI^{scnp}$)
(2) !bind/unbind($SubC^k.E\_RI^{scnr}, SubC^j.I\_PI^{scnr}$), $k \neq j$
(3) !bind/unbind($SubC^k.E\_RI^{scnr}, I\_PI^{nr}$)

Figure iv: Modèle de comportement de composants

Dans la figure, le comportement des sous-composants (c'est à dire leur LTSs statiques) est représenté par la boîte appelée **SubC$^{\mathbf{k}}$**. Pour chaque interface fonctionnelle définie dans la description ADL du composant, une boîte encodant le comportement de ses vues internes (I_PI et I_RI) et externes (E_PI et E_RI) est incorporée. Le traitement d'un appel de méthode dans Fractive est encodé dans la boîte appelée Interceptor que nous détaillerons plus tard. Les pointillés dans les boîtes indiquent une relation de causalité induite par le flot des données au travers de la boîte.

Le comportement des interfaces inclut les aspects fonctionnels (appel de méthodes $\mathcal{M}(\overrightarrow{x})$) et non-fonctionnels (contrôle), et la détection des erreurs ($\mathcal{E}_1$ et $\mathcal{E}_2$) comme

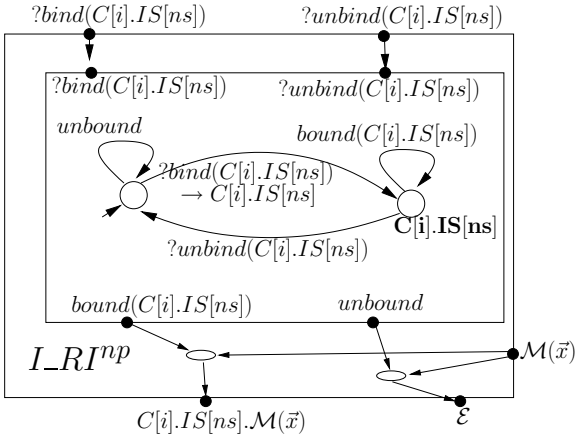


Figure v: Détail de l'interface interne d'une boîte

l'usage d'une interface non liée. Ces erreurs devient visibles dans les prochaines niveaux de la hiérarchie. Par exemple dans la figure v nous montrons le détail de I_RI^{np} qui inclut la création d'une erreur lorsque une méthode est appelée sur une interface non attachée.

Notez que nous avons choisi de mettre les automates des interfaces extérieures dans le prochain niveau de la hiérarchie. Çela nous permet de calculer l'automate *contrôleur* d'un composant avant même de connaître son environnement. Ainsi, toutes les propriétés qui ne concernent pas les interfaces extérieures peuvent être vérifiées d'une manière complètement oppositionnelles.

Modéliser les Primitives

La figure vi montre le principe de communication asynchrone être deux objets actifs.

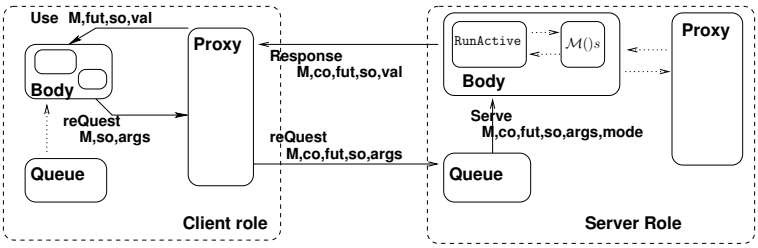


Figure vi: Communication entre deux activités

Dans le modèle (figure vi), un appel distant vers une activité traverse un subrogé (proxy), qui crée localement un objet futur, pendant que l'appel va vers la queue distante des appels. Les arguments de l'appel incluent une référence vers le futur avec une copie profonde des arguments de la méthode parce que il n'y a pas de partage de donnés entre activités distantes. Plus tard, l'appel peut éventuellement être servi et la valeur de son résultat sera envoyée pour remplacer la référence future.

Pour construire le modèle d'un composant primitif (figure vii) nous ajoutons au modèle d'un objet actif deux boîtes additionnelles : **LF** et **NewServe** (qui correspondent à l'Interceptor dans la figure iv.

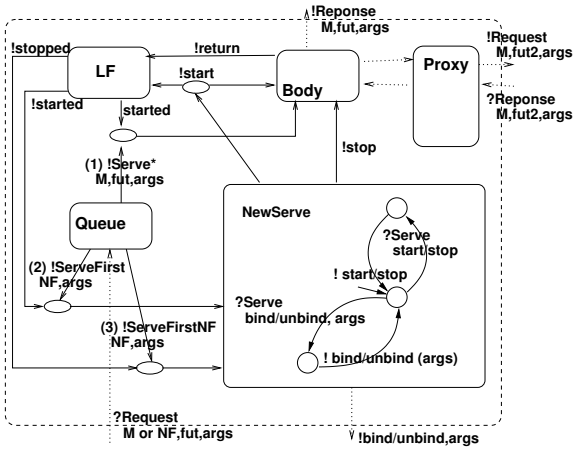


Figure vii: Modèle de comportement d'un composant primitif de Fractive

La boîte **NewServe** implante le traitement des appels de contrôle. L'action "start" déclenche la méthode `RunActive` de **body**. L'action "stop" déclenche la synchronisation "!stop" avec **body** (Fig.vii). Cette synchronisation doit finalement produire la terminaison de la méthode `RunActive` (`!return` synchronisation). Dans l'implantation de Fractive, cette terminaison est faite par l'affectation de la valeur faux à la variable `isActive`, qui doit causer la terminaison de la méthode `RunActive`. Seulement quand la méthode a terminé le composant est considéré comme arrêté.

La boîte Queue peut servir trois actions : (1) servir le premiér appel fonctionnel qui correspond à la primitive `Serve` de l'API ProActive utilisée dans le code de body, (2) servir une méthode de contrôle seulement s'il est en tête de la queue, et (3) servir seulement les méthodes de contrôle dans la queue en ordre FIFO, en ignorant les appels fonctionnels.

Modeler les Composites

La membrane d'un composant Fractive est elle-même un objet actif. Quand elle est démarré, elle sert les méthodes fonctionnelles et non-fonctionnelles en ordre FIFO, en réexpédiant les méthodes entre des interfaces intérieures et extérieures. Quand elle est stoppée, elle ne sert que des méthodes de contrôle.

L'objet actif d'une membrane est construit à partir de la description du composite (donné par l'ADL). Cette membrane correspond à la boîte Interceptor de la figure iv. Notez que les références futures (la boîte **proxy** dans la figure viii) sont mises à jour en suivant une chaîne du composant primitif qui sert la méthode jusqu'au composant primitif qui a fait l'appel. Comme les appels de méthodes incluent une référence au futur, la mise à jour de futurs peuvent être adressés directement à l'appelant juste avant dans la chaîne. Par conséquent, comme dans l'implantation, la mise à jour du futur n'est pas influencé par des actions de re-attachement ou par l'état du cycle de vie du composant.

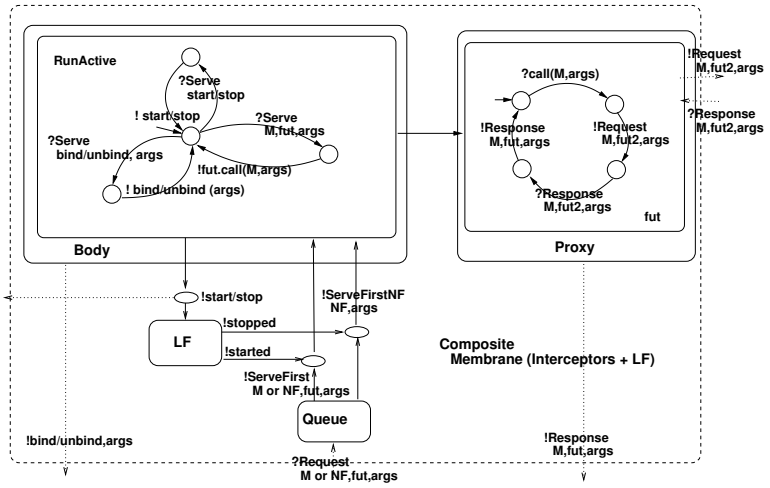


Figure viii: Modèle de comportement de la membrane de un composite de Fractive

Générer le comportement global

La prochaine étape est de générer le comportement global du composant. Cette construction comportementale est compositionnelle dans le sens où chaque niveau de la hiérarchie (chaque composite) peut être étudié indépendamment.

Similaire à nos travaux précèdents [22, 21], avant de générer le produit synchronisé, nous construisons des abstractions finies de nos modèles avec des abstractions finies des valeurs des paramètres. Quand l'outil (model-checker) le permet, l'instanciation est faite à la volé pendant la vérification. Cette abstraction des données est interprété comme une partition des domaines de donnés et induit une interprétation abstraite du LTS paramétré. L'instanciation sera aussi choisie à partir des valeurs qui sont présentes dans la propriété à vérifier.

3.4 Le point de vue de l'utilisateur

Les modèles pour les aspects non-fonctionnels décrits ici sont construits automatiquement. L'utilisateur n'a qu'à fournir l'architecture sous forme d'ADL Fractal et le comportement fonctionnel des composants primitifs.

Étude d'un exemple

Nous revenons à l'exemple de la figure ii. Celui-ci montre, sous forme de système de composants hiérarchiques, le problème classique d'un buffer borné avec un producteur et un consommateur. Le consommateur consomme un élément à la fois alors que le producteur peut remplir le buffer avec un nombre arbitraire d'éléments en une seule action. De plus, le buffer émet une alarme sur son interface I_{alarm}, quand il est plein.

L'utilisateur peut décrire la topologie du système en utilisant l'ADL (langage de définition d'architecture) Fractal. Fractive utilise la syntaxe par défaut de cette ADL basée sur XML. Le fichier XML décrivant **System** est montré sur la figure ix.

La description XML de la figure ix spécifie que le système est composé du composite **BufferSystem** (ligne 6) lui-même décrit dans un fichier séparé

System.fractal

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE .... >
3
4  <definition name="components.System">
5
6     <component name="BufferSystem"
7           definition="components.BufferSystem(3)">
8        <interface name="alarm" role="client"
9              signature="components.AlarmInterface"/>
10    </component>
11
12    <component name="Alarm">
13       <interface name="alarm" role="server"
14             signature="components.AlarmInterface"/>
15       <content class="components.Alarm">
16          <behaviour file="AlarmBehav"
17                format="FC2Param"/>
18       </content>
19    </component>
20
21    <binding client="BufferSystem.alarm"
22         server="Alarm.alarm"/>
23 </definition>
```

Figure ix: System ADL

`components/BufferSystem.fractal` et du primitif **Alarm** dont l'implantation est la classe Java `components.Alarm` (ligne 15). **BufferSystem** reçoit comme paramètre de construction la taille maximale du buffer (3 dans notre exemple, ligne 7) et requiert une interface nommée `alarm` de type `components.AlarmInterface` (lignes 8,9). **Alarm** fournit une interface `alarm` de type `components.AlarmInterface` (lignes 13,14). Le tag `behaviour` (ligne 16) pointe vers un fichier qui contient la description du comportement de `alarm` sous forme d'un fichier FC2Parameterized.

Enfin, aux lignes 21 et 22 l'ADL définit que, lors du déploiement, l'interface `alarm` de **BufferSystem** doit être branchée à l'interface `alarm` de **Alarm**.

3.5 Propriétés

Les sections précédentes étaient focalisées sur la construction de modèles corrects, et non pas sur l'expression de propriétés. Cette section présente des propriétés temporelles exprimant des comportements désirés de l'exemple, afin d'illustrer la capacité de vérification de notre approche.

Déploiement

Dans la section 3.3 nous avons défini l'automate de *déploiement*, qui décrit les étapes de contrôle nécessaires pour mettre en place les éléments et branchements du système, et démarrer tous les composants.

En Fractive, les appels de méthodes sont asynchrones, et il peut y avoir un délai entre l'appel d'une méthode de contrôle et son traitement. Donc, la vérification de l'exécution d'une opération de contrôle doit être basée sur l'observation de son application au composant, plutôt que sur l'arrivée de la requête.

- Les actions `Sig(bind(intf1,intf2))` et `Sig(unbind(intf1,intf2))` représentent le fait qu'un branchement entre les interfaces `intf1` et `intf2` est effectif. Il correspond par exemple aux synchronisations `!bind/unbind(`E_RI^{nr}, I_{ext}`)` ou `!bind/unbind(`$I\_RI^{np}, SubC^{k}.E\_PI^{scnp}$`)` de la figure iv.
- Les actions `Sig(start(name))` et `Sig(stop(name))` représentent le fait que le composant `name` est effectivement démarré/arrêté. Il correspond à la synchronisation `!start/stop` de la figure iv.

Une des propriétés intéressantes est que le démarrage hiérarchique a effectivement lieu durant le déploiement; c'est à dire que le composant et tous ses sous-composants sont démarrés à un certain moment. Cette propriété peut être exprimée comme l'atteignabilité (inévitable) de `Sig(start(name))` dans l'automate statique de `System`, pour toutes les exécutions possibles, où `name` = {`System, BufferSystem, Alarm, Buffer, Consumer, Producer`}. Nous laissons les actions `Sig(start(name))` observables dans l'automate statique et nous exprimons cette propriété d'atteignabilité comme la formule de *μ-calcul régulier sans alternance* [129] ci-dessous, vérifiée par notre exemple :

$$\mu\ X.\ (<\texttt{true}>\texttt{true} \wedge [\ \neg\ \texttt{Sig(start(System))}\ ]\ X\ ) \wedge$$
$$\mu\ X.\ (<\texttt{true}>\texttt{true} \wedge [\ \neg\ \texttt{Sig(start(BufferSystem))}\ ]\ X\ ) \wedge$$
$$\mu\ X.\ (<\texttt{true}>\texttt{true} \wedge [\ \neg\ \texttt{Sig(start(Alarm))}\ ]\ X\ ) \wedge$$
$$\mu\ X.\ (<\texttt{true}>\texttt{true} \wedge [\ \neg\ \texttt{Sig(start(Buffer))}\ ]\ X\ ) \wedge$$
$$\mu\ X.\ (<\texttt{true}>\texttt{true} \wedge [\ \neg\ \texttt{Sig(start(Consumer))}\ ]\ X\ ) \wedge$$
$$\mu\ X.\ (<\texttt{true}>\texttt{true} \wedge [\ \neg\ \texttt{Sig(start(Producer))}\ ]\ X\ ) \quad \textbf{(1)}$$

Propriétés purement fonctionnelles

La plupart des propriétés intéressantes concernent le comportement du système après son déploiement, au moins lorsqu'il n'y a pas de reconfiguration. Sur l'exemple, nous voudrions prouver que toute demande d'un élément dans la queue est finalement servie, c'est à dire que l'élément est finalement obtenu. Si l'action consistant à demander un élément est étiquetée `get_req()` et la réponse à cette requête `get_rep()`, alors cette propriété d'inévitabilité est exprimée sous la forme de la formule μ-calcul régulier ci-dessous, également vérifiée par l'automate statique de l'exemple :

$$[\ \texttt{true*.get\_req()}\ ]\ \mu\ X.\ (<\texttt{true}>\texttt{true} \wedge [\ \neg\ \texttt{get\_rep()}\ ]\ X\ ) \quad \textbf{(3)}$$

Propriétés fonctionnelles en présence de reconfigurations

L'approche décrite ici permet de vérifier des propriétés, non seulement après un déploiement correct, mais aussi après et pendant les reconfigurations. Par exemple, la propriété (3) devient fausse si nous arrêtons le producteur car au bout d'un certain temps, le buffer sera vide, et le consommateur sera bloqué en attente d'un élément. Cependant, si le producteur est redémarré, le consommateur recevra, finalement, un élément et la propriété est à nouveau vérifiée. Autrement dit, nous pouvons vérifier que, si le consommateur demande un élément, et le producteur est stoppé, si en définitive le producteur est démarré à nouveau, le consommateur obtiendra l'élément demandé.

Pour prouver ce genre de propriétés, l'automate statique n'est pas suffisant, nous avons besoin d'un modèle comportemental contenant l'opération de reconfiguration demandée. Nous ajoutons au composant un *contrôleur de reconfiguration* (figure x) : son

état de départ correspond à la phase de déploiement, et l'état suivant correspond au restant de la vie du composant, où les opérations de reconfiguration requises sont activées mais ne sont plus synchronisées avec le déploiement. Ce changement d'état est déclenché par la terminaison réussie du déploiement ($\surd$).

Figure x: Synchronisation product supporting further reconfigurations

En ce qui concerne la propriété énoncée ci-dessus, les reconfigurations `?stop(Producer)` et `?start(Producer)` sont laissées visibles, et cette propriété est exprimée par la formule en μ-calcul, qui est aussi vérifiée dans notre exemple :

```
(* If a request from the consumer is done before reconfiguration *)
[ (¬ (?stop(Producer) ∨ ?start(Producer))*.get_req() ] (
    (* a response is given before stopping the producer *)
    μ X . (
        < ¬ ?stop(Producer) > true ∧ [¬ (get_rep() ∨ ?stop(Producer))] X)
∨
    (* or given after restart the producer and without stopping it again *)
    [ true* .  ?start(Producer) ] μ X . (
        <¬ ?stop(Producer)> true ∧ [¬ (get_rep() ∨ ?stop(Producer))] X))
```
(4)

Propriétés de comportement asynchrone

Nous nous intéressons maintenant à une propriété spécifique à l'aspect asynchrone du modèle de composants. Le mécanisme de communication en Fractive permet à tout futur, une fois obtenu, d'être mis à jour par la valeur associée, à condition que la méthode correspondante soit servie et ait terminé correctement; les opérations de branchement, débranchement ou d'arrêt ne permettent pas d'empêcher cela. Par exemple, si le consommateur est débranché après une requête, il obtient, de toutes façons la réponse, même si le lien est alors débranché ou le composant arrêté. En utilisant l'approche pour la reconfiguration présentée ci-dessus : en activant `?unbind(buffer,Buffer.get)` et `?stop(Consumer)`, la propriété peut être exprimée comme suit. Cette propriété est vérifiée dans le cadre de l'exemple :

```
[ true*.get_req() ] μ X. (< true > true ∧ [ ¬ get_rep() ] X )
```
(5)

4 Conclusions et Travaux Futurs

Cette thèse a traité de la vérification des propriétés comportementales des systèmes répartis à base de composants. Une importance particulière a été accordée à l'applicabilité de la vérification, à l'aide d'outils automatiques, à des systèmes réels.

Nous avons tout d'abord mis en avant la nécessité de la fiabilité des systèmes informatiques, et introduit l'utilisation de méthodes formelles comme une puissante technique pour atteindre cet objectif. Nous avons ensuite établi les difficultés particulières associées à la vérification des systèmes répartis à l'aide de méthodes formelles, et nous avons dressé un état de l'art de ce domaine de recherche.

Nous avons discuté les raisons pour lesquelles les formalismes et langages de descriptions existants ne sont pas adaptés pour les systèmes que nous envisageons et pour nos objectifs. Suite à cette discussion, nous avons proposé une nouvelle approche pour la modélisation des systèmes répartis. Nous avons validé notre approche par une étude de cas, et l'avons appliquée à la modélisation automatique et à l'analyse des systèmes répartis à base de composants.

Les contributions principales de nos travaux incluent :

- Une nouvelle approche pour la modélisation du comportement des systèmes répartis à base de composants. Cette approche combine le meilleur de deux approches existantes : les réseaux d'automates communiquants [16, 15], et les graphes symboliques avec affectations [111, 93]. Nous avons nommé les modèles comportementaux de notre approche "réseaux paramétrés d'automates communiquants". Nos modèles paramétrés jouent trois rôles; ils décrivent : les systèmes infinis de manière naturelle et finie (en considérant des domaines de variables non- bornés), une famille de systèmes (en considérant divers domaines de variables), et les grands systèmes de manière compacte (en considérant des grands domaines de variables). Dans [21], nous avons montré que ces modèles sont particulièrement adaptés comme langage cible pour les outils d'analyse statique.

- La définition de FC2Parameterized, une syntaxe concrète pour l'écriture des spécifications de systèmes à l'aide de nos modèles paramétrés. Le format FC2Parameterized a été développé comme une extension du format FC2 [36, 118] incluant des paramètres. Nous avons aussi introduit une notation graphique pour un sous-ensemble du format FC2Parameterized.

- Une étude de cas de notre approche sur un système réparti réel : le système chilien de factures électroniques [65] (actuellement opérationnel).

- L'implantation de FC2Instantiate, un outil pour obtenir (étant donnés les domaines de variables) des systèmes finis non-paramétrés à partir de réseaux paramétré d'automates communiquants. Cet outil a également été appliqué profitablement pour : comparer différentes instanciations, instancier à partir de critères **"per-formula"**, ainsi que pour la recherche de meilleures minimisations. En particulier, les capacités de débogage de l'outil ont aidé aussi bien pour la détection d'erreurs au plus tôt que pour l'analyse de **"backtrack"**.

- L'utilisation de réseaux paramétrés d'automates communiquants pour les spécifications comportementales de composants hiérarchiques. Partant du comportement de composants (primitifs) basiques, nous avons développé un mécanisme permettant l'incorporation du comportement non-fonctionnel au sein d'un contrôleur construit à partir de la description d'un composant. La sémantique d'un composite est calculée comme le produit des LTSs de ses sous-composants et du contrôleur du composite.

- L'utilisation du mécanisme décrit précédemment pour la modélisation de systèmes basés sur Fractal [43]. Fractive est la réalisation d'un système de composants répartis avec le intergiciel ProActive [28, 18, 47] suivant le modèle de composants

Fractal. Notre mécanisme supporte l'incorporation automatique des caractéristiques de Fractive, telles que les queues de requêtes, les subrogés (proxies) de futurs, et les stratégies de service d'appels de méthodes.

- La base pour un outil actuellement en développement, nommé ADL2NET, pour la lecture des descriptions d'un système, données sous la forme d'un ensemble d'ADL Fractal [43], ainsi que pour la génération de modèles comportementaux à l'aide de notre mécanisme.

- L'implantation de FC2EXP, un outil et divers scripts, pour l'incorporation de nos formats au sein des boîtes à outils FC2Tools [36] et CADP [81].

- Une classification temporelle des propriétés à vérifier sur les systèmes à base de composants. La plupart de ces propriétés, telles que le déploiement correct et la détection d'erreurs, peuvent être appliquées de manière systématique à tout système à base de composants.

- L'illustration de la vérification de propriétés à l'aide de trois formalismes différents pour les exprimer : les automates abstraits [36, 10], les formules ACTL [60], et les formules régulières μ-calcul [129]. L'illustration inclue des propriétés de chaque classification temporelle et considère les aspects asynchrones des composants répartis.

Par ailleurs, lorsque cela s'est avéré nécessaire, nous avons proposé et analysé divers mécanismes pour éviter autant que possible le fameux problème d'*explosion d'états* dans la construction du comportement d'un système.

Finalement, diverses approches ont été développées pour couvrir la bonne composition de composants en considérant leurs aspects fonctionnels. Un des avantages principaux de l'utilisation de composants est la séparation des préoccupations du point de vue de l'utilisateur. Cependant, lors de l'application de la vérification comportementale, il est toujours nécessaire de prendre en compte les interdépendances entre les aspects fonctionnels et non-fonctionnels, du moins pour les modèles de composants existants. La principale originalité de nos travaux est de considérer le déploiement et les reconfigurations comme faisant partie du comportement du système, et ainsi de vérifier le comportement de tout un système à base de composants.

Cette thèse représente un pas vers la réalisation d'une boîte à outils de vérification comportementale concrètement utilisable. Cette boîte à outils est capable de construire des modèles de manière automatique, et fourni du retour sur des propriétés génériques et la détection d'erreurs. Les modèles ainsi générés permettent la définition et la vérification de nouvelles propriétés, qu'il est alors possible de comparer à une spécification.

4.1 Travaux futures

À court terme nous sommes dédiés à développer les outils, puis les utiliser avec des cas d'étude plus grands, qui sûrement nous donneront des retours pour améliorer notre approche.

À moyen et long terme, quelques directions de recherche qui peuvent être explorées sont :

Relation de pré-ordre

Une des questions qu'on laissé ouverte dans ce travail est de répondre à la question si un composant est fidèle avec sa spécification. Répondre à cette question nous permettra, sans devoir générer à nouveau le modèle du composant, déterminer si un composant peut remplacer un autre d'un manière sûr.

Il y quelques méthodes pour répondre a cette question; des équivalences par bisimulation permettraient de garantir que les propriétés de comportement sont préservées, mais sont des relations trop fortes dans le sens que plusieurs composants qui peuvent remplacer des autres d'une manière sûre dans le système seront refusés.

Une solution peut être l'utilisation de la relation de *compliance* définit dans Sofa [146]. Mais dans Sofa les comportements sont exprimés comme des expressions régulières, et la relation de *compliance* est basée sur l'inclusion des traces. Il n'est pas clair d'appliquer une approche similaire à nos sémantiques basées sur les équivalences de bisimulation.

De toute façon nous pensons que les idées qui sont inspiré la relation *compliance* dans Sofa, basées sur des obligations qu'un processus doit remplir, sont bien énoncées et nous voulons explorer leur applicabilité à notre approche. Cette exploration nous doit conduire vers une relation de pré-ordre, qui permettra à l'implantation de faire quelques choix entre les possibilités laissées ouvertes par la spécification, mais aussi compatible avec la composition en utilisant les réseaux de synchronisation.

Exprimer les propriétés

Dans le manuscrit complet, nous avons montré trois façons d'exprimer des propriétés : les automates abstraits [36, 10], ACTL [60], et le μ-calcul régulièr [129]. Les automates abstraits peuvent être vus comme une façon plus facile d'exprimer propriétés ; d'autre part, le μ-calcul régulière est le formalisme le plus expressif. ACTL peut se placer entre les deux.

Les trois façons demandent un utilisateur assez qualifié et elles sont loin de ne pas être sujet à l'erreur. Un travail intéressant pour approcher la distance entre la spécification des propriétés et les utilisateurs inexperts a été fait par Dwyer et al. [67]. Ils ont défini, en classifiant dans un espace temporel, plusieurs patrons (patterns) qui permettent d'exprimer la plus part des propriétés avec une syntaxe proche de langage naturelle.

Nous voulons proposer des extensions aux patrons de Dwyer pour exprimer des propriétés spécifiques aux composants. Par exemple, nous pouvons définir le patron `AfterDeployment` pour exprimer l'espace temporel après une réussite du déploiement. D'autres patrons peuvent êtres `NoErrors`, `ControlActions` et `FutureUpdate`, tous expriment des ensembles spécifiques d'actions.

Néanmoins, avant de définir ces patrons d'une manière précise, nous avons besoin de plus d'expérience et des résultats de cas réels d'études avec des systèmes distribués.

Nouvelles caractéristiques de Fractive

Fractive est développé en continu, avec des nouvelles caractéristiques comme les dernières : interfaces collectives avec des communications à tous (broadcast) ou à certains (multicast) composants; et aussi il y a toujours des discussions ouvertes comme : est-ce qu'un composant doit préserver sa queue des appels quand il est remplacé, ou même à quel moment peut-il être remplacé.

Notre approche doit évoluer avec les nouvelles caractéristiques de Fractive.

Part II

Thesis

Chapter 1

Introduction

I am pretty sure that many people, including the readers of this thesis, would swear at the screen and mumble the same advocated by Richard Sharpe in [150] when Windows crash:

> *If they built roads, bridges, cars, planes and ships like this software, the human race would be doomed*

Yet software is now at the heart of many of these products. Why are Airbus planes, with their fly-by-wire technology, not falling out of the sky as regularly as Windows crashes? Why do not the unmanned Paris Metro trains crash into each other every day?

Part of the reason is that such safety-critical software is developed using *formal methods*. Aircraft designers use mathematics to model the complex systems of lift and thrust needed to keep an Airbus in the sky. Bridge designers use mathematics to assess the stresses on the materials from which they can build bridges.

Formal methods are the mathematical foundation for software. A method is formal if it has a sound mathematical basis, typically given by a formal specification language. This basis provides means of precisely defining notions like consistency and completeness and, more relevant, specification, implementation and correctness. It provides the means of proving that a specification is realisable, proving that a system has been implemented correctly, and proving properties of a system without necessarily running it to determinate its behaviour.

It is the use of mathematics to specify, model, develop and reason about computing systems that is both its strength and its weakness. One weakness is because there are far too few programmers with a background in mathematics who are comfortable with the notation at the heart of formal methods and they are not enough to provide the code needed to exploit the success delivered by Moore's Law in hardware.

Moreover, to formally verify a system, i.e. using a mathematical-based approach, is a difficult task. To prove that a system holds a certain property is usually an undecidable problem. This happens because systems are designed to execute infinite cycles, and manipulate uncountable or infinite data sets (e.g. real numbers, non bounded integers or time). The system needs, using complex abstraction techniques [57], to be approximated to finite discrete models, where efficient algorithms of verification exists [68].

This is what this thesis is about. We want to provide methods and tools which enable people, not necessarily experts in formal methods, to verify the correctness of systems in a simple and straightforward way, hiding away as much as possible the complexity.

Formal methods are rich in literature and considerably varies depending on the system's target or even the development's phase. Pretend to develop an approach suitable for all of them would be to build a castle in the air. We focus, as the thesis title says, on systems built from distributed components.

Many works have been pretty successful in applying formal methods in various domains, including circuit design, embedded and synchronous software, or some classes of real-time systems. However, on distributed systems, the hard problems stem from parallel processing and asynchronous communication may lead to undesirable behaviours (e.g. deadlocks, starvation) on addition to increase the verification task complexity.

The author's thesis belongs to the project OASIS [3] at INRIA Sophia-Antipolis which assigns several resources in the development of a middle-ware for building distributed applications, named ProActive [28, 46]. Within this framework, we develop methods and tools for the formal analysis and verification of systems built using ProActive.

In addition, ProActive recently features components programing [29]. Components have shown as a gaining programing paradigm which considerable helps the implementation and maintenance of every day growing complex software systems. Indeed, components programming have shown as a solution to deal with the complexity in the design and coding of such systems. The components implementation made by the team OASIS features the same distributed nature of ProActive.

Hence the focus and motivation of this work. Our particular target are distributed components built using ProActive, but as we show along this thesis, our ideas can be applied to other kinds of distributed applications and component systems as well.

1.1 The need for reliable systems

On June 19th 2003, the prestigious magazine "The Economist" published an article about software development titled "Building a better bug-trap" [9]. It begins by stating:

> *People who write software are human first and programmers only second - in short, they make mistakes, lots of them. Can software help them write better software?*

The article continues

> *Our civilisation runs on software, Bjarne Stroustrup, a programming guru, once observed. Software is everywhere, not just in computers but in household appliances, cars, aeroplanes, lifts, telephones, toys and countless other pieces of machinery. In a society dependent on software, the consequences of programming errors ("bugs") are becoming increasingly significant. Bugs can make a rocket crash, a telephone network collapse or an air-traffic-control system stop working. A study published in 2002 [151] by America's National*

Institute of Standards and Technology (NIST) estimated that software bugs are so common that their cost to the American economy alone is $60 billion a year or about 0.6% of gross domestic product...

There are many cases [101] where, besides political impact, errors in system development and design has lead not only to huge financial consequences, but also to lost of human lives. Among the most known:

- On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off from Kourou, French Guiana. The rocket was on its first voyage, after a decade of development costing $7 billion. The destroyed rocket and its cargo were valued at $500 million. A board of enquiry investigated the causes of the explosion and in two weeks issued a report [48]. It turned out that the cause of the failure was a software error in the inertial reference system. Specifically a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer. The number was larger than 32,767, the largest integer storeable in a 16 bit signed integer, and thus the conversion failed.

- The bug in Intel's Pentium-II floating-point division unit [71] in the second semester of 1994 caused a loss of about US$475 million to replace faulty processors [7], and severely damaged Intel's reputation as a reliable chip manufacture.

- On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to track and intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks, killing 28 soldiers and injuring around 100 other people. A report of the General Accounting office [34] reported on the cause of the failure. It turned out that the cause was an inaccurate calculation of the time since boot due to computer arithmetic errors. The error was about 0.34 seconds, a Scud travels at about 1,676 meters per second, and so it travels more than half a kilometre in this time. This was far enough that the incoming Scud was outside the "range gate" that the Patriot tracked.

- A software flaw in the control part of the radiation therapy machine Therac-25 [110] caused the death of 6 cancer patients and others were seriously injured between 1985 and 1987 as they were exposed to an overdose of radiation.

- The automated system of the Denver's new international airport [59] was supposed to improve baggage handling by using a computer tracking system to direct baggage contained in unmanned carts that run on a track. Originally scheduled for completion in March 1994, the unfinished $234 million project helped to postpone the opening of the airport until February 1995. The delay reportedly cost the city roughly $1 million per day in operations costs and interest on bond issues, more than the direct cost of the project. Significant mechanical and software problems plagued the automated baggage handling system. In system tests, bags were misloaded, were misrouted, or fell out of telecarts, causing the system to jam.

The article from *The Economist* continues:

> *To make matters worse, as software-based systems become more pervasive and interconnected, their behaviour becomes more complex. Tracking down bugs the old-fashioned way: writing a piece of code, running it on a computer, seeing if it does what you want, then fixing any problems that arise; becomes less and less effective. "People have hit a wall," says Blake Stone, chief scientist at Borland, a company that makes software-development tools. Programmers spend far longer fixing bugs in existing code than they do writing new code. According to NIST, 80% of the software-development costs of a typical project are spent on identifying and fixing defects.*
>
> *Hence the growing interest in software tools that can analyse code as it is being written, and automate the testing and quality-assurance procedures. The goal, says Amitabh Srivastava, a distinguished engineer at Microsoft Research, is to achieve predictable quality in software-making, just as in carmaking. "The more you automate the process, the more reliable it is," he says. In short, use software to make software better.*

As Clarke stated in [52], hardware and software systems will inevitably grow in scale and functionality. Because of this increase in complexity, the likelihood of subtle errors is much greater. Moreover, as we mention in real cases above, some of these errors may cause catastrophic loss of money, time, or even human life. Then a major goal of software engineering is to enable developers to construct systems that operate reliably despite this complexity.

One way of achieving this goal is by using *Formal Methods*, which are mathematically based languages, techniques, and tools for specifying and verifying such systems. Use of formal methods does not *a priori* guarantee correctness. However, they can greatly increase our understanding of a system by revealing inconsistencies, ambiguities, and incompletenesses that might otherwise go undetected.

1.2 Formal Methods

An excellent introduction and overview about formal methods can be found in [155]. Formal methods provide frameworks in which people can specify, develop, and verify systems in a systematic, rather than ad hoc, manner.

Robert Floyd, in his seminal 1967 paper, "Assigning Meanings to Programs" [75] opened the field of program verification and formal methods. His basic idea was to attach so-called "tags" in the form of logical assertions to individual program statements or branches that would define the effects of the program based on a formal semantic definition of the programming language. Many researchers in formal methods of computing worldwide adopted this method. One of the most important influences was on C. A. R. Hoare, who in 1969, starting from Floyd's work, developed his calculus of pre and postcondition semantics for computer programs [95].

As we mention before, a formal method uses mathematics to specify, model, develop and reason about computing systems. A formal method also addresses a number of pragmatic considerations: who uses it, what it is used for, when it is used, and how

it is used. Most commonly, system designers use formal methods to specify a system's desired behavioural and structural properties.

Anyone involved in any stage of system development can make use of formal methods. They can be used in the initial statement of a customer's requirements, through system design, implementation, testing, debugging, maintenance, verification, and evaluation.

Formal methods are used to reveal ambiguity, incompleteness, and inconsistency. When used early in the system development process, they can reveal design flaws that otherwise might be discovered only during costly testing and debugging phases. When used later, they can help determine the correctness of a system implementation and the equivalence of different implementations.

The particular characteristics which can be described by different formal methods may vary considerably [20]. Roughly speaking, two brands of formal verification approaches can be distinguished: *deductive* and *model-based* methods.

With *deductive methods*, the correctness of systems is determined by properties in a mathematical theory. The verification problem is expressed as a theorem that has the form: *system specification* $\Rightarrow$ *desired property*. Trying to establish this result is referred to as *theorem proving*.

Model-based techniques are based on models describing the possible system behaviour in a mathematical precise and unambiguous manner. The system models are accompanied with algorithms that systematically explore all states (all possibles system scenarios) of the system model. In this way, it can be shown that a given system truly satisfies a certain property, referred to as *model-checking*.

Model Checking

Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model. Two general approaches to model checking are used today: by verifying if temporal logic formulas (expressing the property) hold on the system or by comparing (using a pre-order relation) the system with a specification for determining whether or not its behaviour conforms to the specification.

In contrast to theorem proving, model checking is completely automatic and fast. It also produces counterexamples, which usually represent subtle errors in design, and thus can be used as a debugging aid.

Theorem proving

Theorem proving is a technique where both the system and its desired properties are expressed as formulas in some mathematical logic. This logic is given by a formal system, which defines a set of axioms and a set of inference rules. Theorem proving is the process of finding a proof of a property from the axioms of the system. Steps in the proof appeal to the axioms and rules, and possibly derived definitions and intermediate lemmas.

In contrast to model checking, theorem proving can deal directly with infinite state spaces. It relies on techniques like structural induction to prove over infinite domains.

Its main drawback is that the verification process is usually slow, error-prone, labour-intensive to apply and requires a rather high degree of user experience.

Because of this drawback and since this work focuses on automatic and friendly tools for verification, we rather rely on model-based techniques.

1.3 The myths of formal methods

Research in formal methods has led to the development of powerful software tools that can be used for automating various verification steps. Investigations have shown that formal verification procedures would have revealed the exposed defects in e.g., the Ariane 5 missile, Intel's Pentium II processor and the Therac-25 therapy radiation machine.

The application of formal methods has a long history. There is a significant take up of formal methods in critical industries [94], however it has not been substantially adopted by the software engineering community at large [35].

This situation is hardly surprising since formal methods technology is largely perceived to consist of a collection of prototype notations and tools which are difficult to use and do not scale up easily; there are many widely held misconceptions about the use of formal techniques. Anthony Hall mention seven myths of formal methods [90], we recall two of them:

1. *Formal method can guarantee that software is perfect*

   Nothing can achieve perfection. Formal methods are not the panacea for system reliability, but it can considerably improve it. They should be seen as complementary and powerful methods to other well establish verification techniques such as debugging and testing. In particular for model-based approaches, we should remember the fact that:

   > *Any verification using model-based techniques is only as good as the model of the system*

   Hence why this work is mainly concentrated in system modelling. Its first half is concentrated in finding the most suitable format for modelling the system's behaviours, while the second half profits from the component semantics and structure for building models of such system as much automatic as possible.

2. *Formal methods require highly trained mathematicians*

   It is real that it has been a lot of work on the formal specification of systems which make formal methods easier to use (at least to some target systems), but we disagree at some level with A. Hall. Depending on the formalism, on the target system and on the properties being proved, the chosen formal method may require strong mathematical basis. In fact, just to chose the most suitable formal method approach for the target system would require at least a basic knowledge on the field.

   For the systems we want to address in this work, in addition to automation, we want to hide away the complex notations, logics and algorithms for formal verification behind friendly interfaces. In general we try to adapt established approaches

of system specifications (such as Architecture Description Languages (ADL) as we show in chapters 4 and 5) for the formal verification proposes.

1.4 The distributed systems

Model-checking has been shown as a powerful technique for verifying hardware, embedded systems and sequential or concurrent memory shared systems. This technique has several important advantages over mechanical theorem provers or proof checkers. The most important being that the procedure is highly automatic. Typically, specifications are expressed in a propositional temporal logic (such as CTL [68]) and the system is modelled as a *state-transition* graph (named Kripke structure). The successful use of model checking relies on efficient algorithms to check properties and a symbolic representation of the state graph using *ordered binary decision diagrams (OBDDs)* [44] which enables representation of realistic large systems.

However, when moving to distributed systems, i.e. communicating concurrent processes, the state-based modelling is not well suited. In the absents of shared memory, where the sates may be easily characterised by the state of the system's variables, it is difficult (if not impossible) to identify the state of the system, and therefore its modelling. In addition, the concurrency introduces interleaving and non-determinism which increase exponentially the size of the state-based model.

On the other hand, in communicating concurrent processes it is easier to distinguish the actions that each of the processes may do at a given moment, including actions that may represent communications between processes. Communicating actions should be done at the same time (synchronised) in all the processes involved in the communication. The formalism for modelling processes where we distinguish the actions that can be executed instead of its internal structure is known as *Labelled Transition Systems (LTSs)*.

Note that the LTS approach does not see the nature of the processes (such as their state of variables) but rather *observes* what a process may execute. This view has lead, starting from the seminal works of Milner on CCS [130] and Hoare on CSP [96], to a rich family of algebras which enables reasoning about communicating concurrent processes behaviour, known as *Process Algebras*.

We do use *process algebras* for modelling the behaviour of our systems. Being in an algebraic framework enables us to profit from equivalence relations, modular design and abstractions leading to smaller systems size while preserving their semantics. In chapter 2 we take a deeper view on the main work on process algebras and their features.

Thanks to later work which have taken the best from both approaches (state-based and process algebras), today we can profit in process algebras of symbolic representations using OBDDs [74] and model-checking using several temporal logics [125, 129, 127].

1.5 Components programming

Component programming has emerged as a programming methodology ensuring both re-usability and composability. In general words, a component is a self contained en-

tity that interacts with its environment through well-defined interfaces. Besides these interactions, a component does not reveal its internal structure.

Several component models have been proposed [145, 43, 64, 8] and some of them are currently being used in the industry. All of them have common features, such as encapsulation, and some of them have advanced features such as nested (hierarchical) composition and distribution.

In hierarchical component frameworks like Fractal, different components can be assembled together creating a new self contained component, which can be itself assembled to other components in a upper level of the hierarchy. Hierarchical components hide, at each level, the complexity of the sub-entities. The compositional aspect, together with the separation between functional and non-functional aspects, help the implementation and maintenance of complex software systems. Among the most interesting of those non-functional aspects are life-cycle and reconfiguration capabilities, allowing the control of the execution of a component, but also its dynamic evolution.

The main goal of this thesis is to build a formal framework to ensure that those compositions are correct when deploying component systems. We focus on Fractive [29], an implementation of the Fractal component model [43] using the middle-ware ProActive [28].

1.6 Thesis structure, initial assumptions and goals

We have discussed in the previous sections the need for computer systems reliability and we have introduced formal methods as a powerful technique to achieve this goal. We have identified some reasons why formal methods are not widely applied on software production and, at the same time, we have described some features that a solution must have to go over them (e.g. automation and friendly-use). We have stated as well our focus on distributed component systems and we have justified some initial decisions about which would be the better formal method approach to such systems.

Our aim is to ensure that an application built from distributed components is safe, in the sense that its parts fit together appropriately and behave together smoothly. Each component must be adequate to its assigned role within the system, and the update or replacement of a component should not cause deadlocks or failures to the system.

The usual notion of type compatibility of interfaces, in the spirit of OO method typing, is not sufficient; it does not prevent assembled components from having non compatible behaviours, that could lead to deadlocks, live-locks, or other kinds of safety problems.

The challenge is to build a formal framework of methods and tools for ensuring not only that compositions are correct when deploying those component system, but also further dynamic changes or reconfigurations. This framework should be at the same time formal enough to be used by the tools, and simple enough to be used by non-specialists; the tools should be as much automatic as possible, hiding away their logical and algorithmic complexity.

1.6.1 Thesis structure

This thesis is written to be as much self-contained as possible and it should be read in order. The latter because many concepts and ideas strongly depend on previous chapters and sections to be better understood.

In the next chapter we review the main works on the theory of process algebras that are relevant to our subject. We also review the description languages and tools that have been developed from such algebras. At the end of the chapter we take an overview of current works featuring behaviour's verification of component systems.

In section 3 we introduce a new intermediate format which is an adaptation of the *symbolic transition graphs with assignment* into the *synchronisation networks*: we extend the general notion of Labelled Transition Systems (LTS) and hierarchical networks of communicating systems (synchronisation networks) adding parameters to the communication events. Events can be guarded with conditions on their parameters. Our processes can also be parameterized to encode sets of equivalent processes running in parallel. The results of this work have been presented in [21, 26, 25, 17].

In section 4 we use this intermediate format to give behavioural specifications of hierarchical components. We assumed the models of the primitive components as known (given by the user or via static analysis). Using the component description, we built a *controller* describing the component's non-functional behaviour. The semantics of a component is then generated as the synchronisation product of : its LTSs sub-components and the controller. The resulting system can be checked against requirements expressed in a set of temporal logic formulas, or again as a LTS. The results of this works have been presented in [22, 23].

In section 5 we do a big step forward by moving to distributed components built using Fractive. Components become active in the same way than active objects: their membrane have a single non-preemptive control thread which serves, based on different serving policies, method requests from its unique pending queue. The requests to other components are done via a *rendez-vous* phase so there is delivery guarantee and order conservation of incoming calls. The responses (when relevant) are always asynchronous with replies by means of future references; their synchronisation is done by a *wait-by-necessity* mechanism.

Similar to synchronous components, we automatically incorporate the non-functional behaviour of the components within its controller, based on the component's definition. In addition, we incorporate the Fractive component features by automatically adding automata encoding the queues, future responses and serving policies (in particular default policies of serving functional versus non-functional requests based on the component life cycle status). Finally we show how our approach enables us to verify properties in all the component's phases, including the interplay between functional and non-functional events: for instance the (inevitable) reachability of responses from asynchronous method calls even during a reconfiguration phase. The results of this work have been presented in [24].

In section 6 we introduce the tools we have developed which implement the approaches proposed by this thesis. Finally, section 7 concludes our work and states future directions of research.

Chapter 2

State of the Art

This thesis has two main axes. First, it proposes models and methods to verify the behaviour of distributed systems using process algebras theories. Second, it analyses how to best apply those models and methods to the specific distributed systems built using components. Both axes are focussed on the main target of this thesis: to ease the verification process to the user, non specialist on formal verification, by providing automatic tools for verification.

On this context, we analyse the most important state of the art works that have influenced the ideas and decisions in the contribution of the thesis.

We start by introducing the process algebra theories which give the formal basis of our approach. Then we analyse the main description languages and tools that have been developed to verify systems based on such algebras. Finally we review the main works on verification of components-like systems.

2.1 Process Algebras: a semantics for concurrent (distributed) systems

Process Algebras allow a rather high level view on interacting systems. They regard all such systems as *processes* (objects in some mathematical domain) describing all the potential behaviours a program or system can execute using operators within an algebraic theory.

A family of different process algebras have been developed, which can be characterised by the use of equations and inequalities among processes expressions, and by an acceptance of synchronised communication as the primitive means of interaction among system components. Among the best-known are: Milner's CCS [131, 130], developed with the help of Hennessy and Park; Hoare's CSP [96, 42], developed in conjunction with Brookes and Roscoe; Hennessy's ATP [92, 91, 137], developed with Nicola; ACP [32, 14] developed by Bergstra, Klop and Baeten; Meije [62, 19], developed by Berry, Boudol and de Simone; and Milner's π-calculus [134, 133, 132, 149, 143], based on CCS but with mobile processes support.

The main distinctions between them are the constructions by which processes are assembled, the method by which processes expressions are endowed with meaning, and the notion of equivalence among process expressions. However, they all share the

following key ingredients:

- **Compositional Modelling**. Process algebras provide a small number of constructors for building up larger systems from smaller ones.
- **Operational Semantics**. Process algebras are typically equipped with structural operational semantics (SOS) that describe the single-step execution capabilities of systems. Using SOS, systems represented as terms in the algebra can be "compiled" into labelled transition systems.
- **Behavioural reasoning via equivalences and preorders**. Process algebras feature the use of behavioural relations as a mean for relating different systems given in the algebra. By means of this equational reasoning a system can be *verified*, i.e. establish that it satisfies a certain property.

On this section we concentrate on two process algebras: the Calculus of Communication Systems (CCS) and the π-calculus, developed by Milner who is without a doubt the central person on the process algebra history. In CCS the behaviour of interacting systems is described using a small set of 6 operators, and reasoning about such systems is possible through equivalence relations named *bisimilarity*. The π-calculus can be seen as an extension of CCS to support mobility, i.e. processes are mobile and the configuration of communications links may dynamically change.

2.1.1 Calculus of Communicating Systems (CCS)

CCS defines a small language whose constructors reflect simple operational ideas. The meaning of those constructors is given through operational semantics. The core of CCS is a congruence relation between closed *terms*, where the terms represent *processes*. A semantic process is understood to be a congruent class of terms.

This congruence relation is therefore interpreted as equality of processes. It is built upon the idea of *observing* a process: processes are equal iff they are indistinguishable in any experiment based upon observation.

Basic Language

CCS starts defining the *expressions* of the process language. Having this done, the operational semantics is presented as *Labelled Transition Systems*. This leads to the notion of *derivation tree*, which records the successive transitions or actions which may be performed by a given process.

CCS presupposes an infinite set $\mathcal{A}$ of names $a, b, c, \ldots$. Then $\bar{\mathcal{A}}$ is the set of *co-names* $\bar{a}, \bar{b}, \bar{c}, \ldots$; $\mathcal{A}$ and $\bar{\mathcal{A}}$ are disjoint and are in bijection via $(\bar{\ })$, $\bar{\bar{a}} = a$.

$\mathcal{L} = \mathcal{A} \cup \bar{\mathcal{A}}$ is the set of *labels* ranged over by l, l'. Labels will identify the actions which may be performed by a process. CCS also introduces the *silent action* $\tau \notin \mathcal{L}$; this special action is considered to be unobservable. Let $Act = \mathcal{L} \cup \{\tau\}$, ranged over by α, β.

A function $f : \mathcal{L} \to \mathcal{L}$ is called a *relabelling function* provided $f(\bar{l}) = \overline{f(l)}$, and is extended to Act by decreeing that $f(\tau) = \tau$. Finally a countable set $\mathcal{X}$ of process variables $X, Y, \ldots$ is presupposed. Then the set $\mathcal{E}$ of *processes expressions* $E, F, \ldots$ (also

called terms) is the smallest set including $\mathcal{X}$ and the following expressions - when E, E_i are already in $\mathcal{E}$.

$\alpha.E$, a *Prefix* ($\alpha \in Act$),
$\sum_{i\in I}$, a *Summation* (I an indexing set),
$E_0|E_1$, a *Composition*,
$E\backslash L$, a *Restriction* ($L \subseteq \mathcal{L}$),
$E[f]$, a *Relabelling* (f a relabelling function),
$\mathbf{fix}_j\{X_i = E_i : i \in I\}$, a *Recursion* ($j \in I$).

In the final form (Recursion) the variables X_i are *bound* variables. For any expression E, its *free* variables $fv(E)$ are those which occur unbound in E. E is *closed* if $fv(E) = \emptyset$; in this case E is a *process*. P, Q, R are used to range over the process $\mathcal{P}$.

Roughly, the meaning of the process constructions is as follows. $\alpha.P$ is the process which performs α and then behaves as P; $\sum_{i\in I} P_i$ is the process which may behave as any (but only one) of P_i; $P_0|P_1$ represents P_0 and P_1 performing concurrently, with possible communication; $P\backslash L$ behaves like P but with actions in $L \cup \bar{L}$ prohibited; $P[f]$ behaves like P but with the actions relabelled by f; finally, $\mathbf{fix}\{X_i = E_i : i \in I\}$ is the jth component of a distinguished "solution" of the recursive process equations $X_i = E_i (i \in I)$.

$\widetilde{E}$ stands for $\{E_i : i \in I\}$ when I is understood, and it is written $\widetilde{X} = \widetilde{E}$ for an I-indexed set of equations. Other abbreviations are $\sum \widetilde{E}$ for a Summation and $\mathbf{fix}_j\widetilde{X}\widetilde{E}$ for a Recursion. Also $\mathbf{fix}\widetilde{X}\widetilde{E}$ means $\{\mathbf{fix}_j\widetilde{X}\widetilde{E} : j \in I\}$. The simultaneous substitution of E_i for free occurrences of X_i in E is written $E\{\widetilde{E}/\widetilde{X}\}$, assuming that bound variables are changed where necessary to avoid clashes.

Operational Semantics

A *Labelled Transition System* (LTS) is of the form $(S, A, \{\xrightarrow{a} : a \in A\})$ where S is a set of *states*, A is a set of *actions* and each $\xrightarrow{a}$ is a subset of $S \times S$, called an *action relation* over S.

The semantics for $\mathcal{E}$ consist in the definition of the action relation $\xrightarrow{\alpha}$ over $\mathcal{E}$ in the LTS $(\mathcal{E}, Act, \{\xrightarrow{\alpha} : \alpha \in Act\})$. The action relations $\xrightarrow{\alpha}$ are defined to be the smallest set which obey the following rules, in which the action below the line is to be inferred from those above the line:

ACT: $\dfrac{}{\alpha.E \xrightarrow{\alpha} E}$

SUM$_j$: $\dfrac{E_j \xrightarrow{\alpha} E'_j}{\sum_{i\in I} E_i \xrightarrow{\alpha} E'_j}(j \in I)$

COM$_0$: $\dfrac{E_0 \xrightarrow{\alpha} E'_0}{E_0|E_1 \xrightarrow{\alpha} E'_o|E_1}$

COM$_1$: $\dfrac{E_1 \xrightarrow{\alpha} E'_1}{E_0|E_1 \xrightarrow{\alpha} E_0|E'_1}$

COM$_2$: $\dfrac{E_0 \xrightarrow{l} E'_0 \qquad E_1 \xrightarrow{\bar{l}} E'_1}{E_0|E_1 \xrightarrow{\tau} E'_0|E'_1}$

RES: $\dfrac{E \xrightarrow{\alpha} E'}{E\backslash L \xrightarrow{\alpha} E'\backslash L}(\alpha \notin L \cup \bar{L})$

REL: $\dfrac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$

REC$_j$: $\dfrac{E_j\{\mathbf{fix}\widetilde{X}\widetilde{E}/\widetilde{X}\} \xrightarrow{\alpha} E'_j}{\mathbf{fix}\widetilde{X}\widetilde{E} \xrightarrow{\alpha} E'_j}$

For Summation (SUM_j), note that the first action of $\sum_{i\in I} E_i$ determines which alternative E_j is selected, the others being discarded. For Composition, the rules COM_0 and COM_1 permit the interleaved performance of E_0 and E_1; the rule COM_2 permits a synchronising communication between E_0 and E_1 whenever they may perform complementary actions. The rule REC_j states that the actions of (the jth component of) the distinguished "solution" of $\widetilde{X} = \widetilde{E}$ are just those inferable by unwinding the recursion.

Extended Language

For many applications it is important to represent the passage of data values between processes. Let's assume that V is the set of data values, and that there are *value variables* $x, y, \ldots$ and *value expressions* e built from constants and standard functions (e.g. arithmetic or boolean operations) over V.

The construction $ax.E$ represents the input of an arbitrary value at a port with name a, in which x is a *bound* value variable, and the construction $\bar{a}e.E$ represents the output of the value e at the port co-named $\bar{a}$. Finally, it is convenient to introduce the conditional construction `if` e `then` E_1 `else` E_2.

The extended language is translated to the basic language by induction upon the structure of expressions. For each expression E without free value variables, its translated form $\widehat{E}$ is given as follows:

E	$\widehat{E}$
$ax.F$	$\sum_{v\in V} a_v.\widehat{F\{v/x\}}$
$\bar{a}e.F$	$\bar{a}_e.\widehat{F}$
`if` e `then` E_1 `else` E_2	$\begin{cases} \widehat{E}_1 & \text{if } e = true \\ \widehat{E}_2 & \text{if } e = false \end{cases}$
$A(e)$	A_e

Congruence of processes

As Milner states in [131], the most obvious equivalence of processes would be one that requires merely that they should possess the same transitions, but when considering deadlocks this proposal is rejected because it is too large, as he shows in an example where in two equivalent processes (using this congruence), one may lead to a deadlock while the other does not.

On the other hand, the equivalence should not be *too* restrictive, such as considering two processes P and Q equivalent just when their derivation trees are isomorphic. This would deny the equivalence of two processes even though, at each stage, the same actions are possible (which Milner shows as well through an example).

Then a well-suited equivalence for processes should be an intermediate notion, with the following property:

> P and Q are equivalent iff, for all $\alpha \in Act$ each α-success of P is equivalent to some α-successor of Q, and conversely.

This equivalence can be formally written as follows, using $\sim$ for the equivalence relation:

$$\begin{aligned} & P \sim Q \text{ iff, } \forall \alpha \in Act, \\ & \text{(i) Whenever } P \xrightarrow{\alpha} P' \text{ then, for some } Q', Q \xrightarrow{\alpha} Q' \text{ and } P' \sim Q' \\ & \text{(ii) Whenever } Q \xrightarrow{\alpha} Q' \text{ then, for some } P', P \xrightarrow{\alpha} P' \text{ and } P' \sim Q' \end{aligned} \tag{2.1}$$

However, there are many equivalence relations $\sim$ that satisfy (2.1). What is really wanted is the *largest* (or *weakest*, or most generous) relation $\sim$ which satisfies (2.1).

Definition 1 *Let $\mathcal{F}$ be the function over binary relations $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ (i.e. binary relations over processes) defined as follows: $\langle P, Q \rangle \in \mathcal{F}$ iff, for all $\alpha \in Act$,*

(i) Whenever $P \xrightarrow{\alpha} P'$ then, for some $Q', Q \xrightarrow{\alpha} Q'$ and $\langle P', Q' \rangle \in \mathcal{R}$
(ii) Whenever $Q \xrightarrow{\alpha} Q'$ then, for some $P', P \xrightarrow{\alpha} P'$ and $\langle P', Q' \rangle \in \mathcal{R}$

Definition 2 *$\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a strong bisimulation if $\mathcal{R} \subseteq \mathcal{F}(\mathcal{R})$.*

Is easy to demonstrate that $\mathcal{F}$ is monotone ($\mathcal{R} \subseteq \mathcal{R}' \Rightarrow \mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}(\mathcal{R}')$)

Definition 3 *P and Q are strongly equivalent or strongly bisimilar, written $P \sim Q$, if $\langle P, Q \rangle \in \mathcal{R}$ for some strong bisimulation $\mathcal{R}$:*

$$\sim = \bigcup \{\mathcal{R} : \mathcal{R} \textit{ is a strong bisimulation}\}$$

Milner proves in [131] that $\sim$ is the largest strong bisimulation and that it is an equivalence relation (reflexive, symmetric and transitive). In particular he shows that $\sim$ is the largest fixed point of $\mathcal{F}$; that is $\sim = \mathcal{F}(\sim)$.

Strong congruence provides a tractable notion of equality of processes, and allows many nontrivial equalities to be derived. Milner introduces several equational laws for $\sim$ grouped as:

1. *monoid laws*: simple properties of Summation (commutative, associative, idempotent and identity under Summation),
2. *static laws*: for Composition, Restriction and Relabelling; named static constructors because they are preserved by the transition; for example if $(P|Q)\backslash L \xrightarrow{\alpha} R$ then R must be of the form $(P'|Q')\backslash L$, and
3. *expansion law*: it relates the static constructors (Composition, Restriction and Relabelling) with the dynamic constructors (Prefix and Summation).

The strong bisimulation treats the unobservable action (τ) on the same basis as all other actions. Some properties which would be expected to hold if τ is unobservable, such as $\alpha.\tau.P = \alpha.P$ do not hold if $\sim$ is taken to mean equivalence.

Next, we introduce another equivalence congruence defined by Milner that relaxes the strong bisimulation by requiring that each τ action be matched by zero or more τ actions. This yields to a weaker notion of bisimulation named *weak* (or *observation*) equivalence.

Definition 4 *Let $t = \alpha_1 \dots \alpha_n \in Act*$. Then*

1. $\stackrel{t}{\longrightarrow} \stackrel{\text{def}}{=} \stackrel{\alpha_1}{\longrightarrow} \dots \stackrel{\alpha_n}{\longrightarrow}$;
2. $\hat{t} \in Act*$ *is the result of removing all τ's from t;*
3. $\stackrel{t}{\Rightarrow} \stackrel{\text{def}}{=} (\stackrel{\tau}{\longrightarrow})* \stackrel{\alpha_1}{\longrightarrow} (\stackrel{\tau}{\longrightarrow}) * \dots (\stackrel{\tau}{\longrightarrow})* \stackrel{\alpha_n}{\longrightarrow} (\stackrel{\tau}{\longrightarrow})*$.

$\stackrel{\hat{t}}{\Rightarrow}$ represents all sequences of actions with the same visible content as $\stackrel{t}{\longrightarrow}$. The Definition 1 is modified for weak equivalence as:

Definition 5 *Let $\mathcal{G}$ be the function over binary relations $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ (i.e. binary relations over processes) defined as follows: $\langle P, Q \rangle \in \mathcal{G}$ iff, for all $\alpha \in Act$,*

(i) Whenever $P \stackrel{\alpha}{\Rightarrow} P'$ then, for some $Q', Q \stackrel{\hat{\alpha}}{\Rightarrow} Q'$ and $\langle P', Q' \rangle \in \mathcal{R}$

(ii) Whenever $Q \stackrel{\alpha}{\Rightarrow} Q'$ then, for some $P', P \stackrel{\hat{\alpha}}{\Rightarrow} P'$ and $\langle P', Q' \rangle \in \mathcal{R}$

Definition 6 *$\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a weak bisimulation if $\mathcal{R} \subseteq \mathcal{G}(\mathcal{R})$.*

Definition 7 *P and Q are observation-equivalent or weakly bisimilar, written $P \approx Q$, if $\langle P, Q \rangle \in \mathcal{R}$ for some weak bisimulation $\mathcal{R}$:*

$$\approx = \bigcup \{\mathcal{R} : \mathcal{R} \text{ is a weak bisimulation}\}$$

Again, Milner proves in [131] that $\approx$ is the largest weak bisimulation and that is an equivalence relation (reflexive, symmetric and transitive). In particular he shows that $\approx$ is the largest fixed point of $\mathcal{G}$; that is $\approx = \mathcal{G}(\approx)$.

Note that $P \sim Q \Rightarrow P \approx Q$.

2.1.2 The π-calculus

The π-calculus [134, 133, 132, 149, 143] is a process algebra in which processes may change their configuration (structure) dynamically. Not only may the component agents (processes) of a system be arbitrarily linked, but a communication between neighbours may carry information which changes this linkage.

The π-calculus was introduced by Milner based on the work done by U. Engberg and M. Nielsen [72], who successfully extended CCS [130, 131] to include mobility while preserving its algebraic properties.

Basic definitions and syntax

π-calculus assumes a potentially infinite set $\mathcal{N}$ of *names*, ranged over by x, y, z, w, u, v. It also assumes a set of *agent identifiers* ranged over by C, where each agent identifier C has a nonnegative *arity* $r(C)$.

Names indistinctly encode all: links names (or communication ports), variables and ordinary data values.

Agents or *process expressions*, ranged over by $P, Q, R, \dots$, are defined as follows:

$$
\begin{array}{rlll}
P & ::= & \mathbf{0} & \text{(inaction)} \\
& | & \bar{x}y.P & \text{(output prefix)} \\
& | & x(y).P & \text{(input prefix)} \\
& | & \tau.P & \text{(silent prefix)} \\
& | & (y)P & \text{(restriction)} \\
& | & [x = y]P & \text{(match)} \\
& | & P|Q & \text{(composition)} \\
& | & P + Q & \text{(summation)} \\
& | & C(y_1, \ldots, y_n) & \text{(defined agent, } n = r(C)\text{)}
\end{array}
$$

inaction: the agent that cannot perform any action,

output prefix: the name y is sent along the name x, and thereafter the agent behaves like P,

input prefix: a name is received along the name x, and y is a placeholder for the received name. After the input the agent behaves like P but with the newly received name replacing y,

silent prefix: the agent performs the silent action τ and then behaves like P,

restriction: the agent behaves like P except that the actions at port $\bar{y}$ and y are prohibited (but communications between components within P are allowed),

match: the agent behaves like P if the names x and y are identical, and otherwise like $\mathbf{0}$,

composition: the agents P and Q execute in parallel. They can act independently, and may also communicate if one performs an output and the other an input along the same port,

summation: the agent can behave either like P or Q,

defined agent: Every agent has a definition $C(y_1, \ldots, y_n) \stackrel{\text{def}}{=} P$, where the names $y_1, \ldots, y_n$ are distinct and are the only names that may occur freely in P. Then $C(x_1, \ldots, x_n)$ behaves like P with x_i replacing y_i for each i. Recursion is provided in the equation definition, since P may contain any agent identifier, even C itself.

Operational Semantics

As in CCS, the semantics for π-calculus are given in terms of Labelled Transition Systems. However, in π-calculus there are two ways to treat input actions named *early instantiation* and *late instantiation*.

In the early approach, input transitions are in the form $x(y).P \xrightarrow{xu} P\{u/y\}$ meaning that the agent receives the name u and then behaves like P with the variable y instantiated to a value u, i.e. the variables are instantiated at the time of inferring the input action.

In the late approach, input transitions are in the form $x(y).P \xrightarrow{x(u)} P\{u/y\}$ meaning that the agent can receive a name u and then behave like $P\{u/y\}$. In that action u does not represent the value received, but rather it is a reference to the places in P where

the received name will appear. The name becomes instantiated only when inferring an internal communication.

The π-calculus semantics for the late approach is given through the following set of rules LATE for inferring transitions:

ACT: $\frac{}{\alpha.P \xrightarrow{\alpha} P}$ SUM: $\frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'}$

PAR: $\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$, $bn(\alpha) \cap fn(Q) = \emptyset$

L-COM: $\frac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P|Q \xrightarrow{\tau} P'|Q'\{y/z\}}$ CLOSE: $\frac{P \xrightarrow{\bar{x}(y)} P' \quad Q \xrightarrow{x(y)} Q'}{P|Q \xrightarrow{\tau} (y)(P'|Q')}$

RES: $\frac{P \xrightarrow{\alpha} P'}{(y)P \xrightarrow{\alpha} (y)P'}$, $y \notin bn(\alpha) \cup fn(\alpha)$ OPEN: $\frac{P \xrightarrow{\bar{x}y} P'}{(y)P \xrightarrow{\bar{x}(y)} P'}$, $y \neq x$

$P \xrightarrow{\alpha}_L Q$ is written to mean that the transition $P \xrightarrow{\alpha} Q$ can be inferred from LATE.

The set of rules EARLY is obtained from LATE by replacing the rule L-COM with the following two rules:

E-INPUT: $\frac{}{x(y).P \xrightarrow{xw} P\{w/y\}}$ E-COM: $\frac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{xy} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$

$P \xrightarrow{\alpha}_E Q$ is written to mean that the transition $P \xrightarrow{\alpha} Q$ can be inferred from EARLY.

Congruences of processes

Milner applies the same ideas of bisimulation from CCS to π-calculus. As he states in [133], several considerations about bound variables (in particular when they encode links in input transitions) lead to the following definition of simulation:

Definition 8 *A binary relation $\mathcal{R}$ on agents is a late (strong) simulation if $P\mathcal{R}Q$ implies that:*

(i) if $P \xrightarrow{\alpha} P'$ and α is τ, $\bar{x}z$, or $\bar{x}(y)$ with $y \notin fn(P,Q) \Rightarrow \exists Q' : Q \xrightarrow{\alpha} Q' \wedge P'\mathcal{R}Q'$

(ii) $P \xrightarrow{x(y)} P'$ and $y \notin fn(P,Q) \Rightarrow \exists Q' : Q \xrightarrow{x(y)} Q' \wedge \forall w : P'\{w/y\}\mathcal{R}Q'\{w/y\}$

The relation $\mathcal{R}$ is *late (strong) bisimulation* if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are late (strong) simulations. $P \dot{\sim}_L Q$ means that $P\mathcal{R}Q$ for some late (strong) bisimulation $\mathcal{R}$.

Note that the late simulation does not require anything from free-input actions. Instead, there is a strong requirement on bound-input actions: the resulting agents P' and Q' must continue simulating all instances w of the bound name. The term "late" refers to the fact that these w are introduced after the simulating derivate Q' has been chosen. The algebraic theory of $\dot{\sim}_L$ is explored in [133].

In the early input transition, the object represents the received value (therefore there is no need for clause (ii) in Definition 8). The natural bisimulation equivalence for early instantiation will use the actions rather than the extra requirement on bound-input actions:

Definition 9 *A binary relation $\mathcal{R}$ on agents is a early (strong) simulation if $P\mathcal{R}Q$ implies that:*

$$\textit{(i) if } P \xrightarrow{\alpha} P' : bn(\alpha) \cap fn(P,Q) = \emptyset \Rightarrow \exists Q' : Q \xrightarrow{\alpha} Q' \wedge P'\mathcal{R}Q'$$

The relation $\mathcal{R}$ is *early (strong) bisimulation* if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are early (strong) simulations. $P \dot{\sim}_E Q$ means that $P\mathcal{R}Q$ for some early (strong) bisimulation $\mathcal{R}$.

As for CCS, Milner gives in [133] several properties and algebraic laws for strong bisimilarity. He also briefly outlines a late weak bisimilarity analogous to CCS. Both late and early weak bisimulation equivalences, as well as several variants of π-calculus and bisimulation relations, are discussed in [143].

2.1.3 Networks of Communicating Automata

The main concept of communicating automata is to define the interactions between concurrent processes (communications, synchronisations) at a very high level of abstraction.

It was first introduced by Maurice Nivat in [138] as the result of a joint seminar in parallel and concurrent system semantics in the French company *Thomsom-CSF*. A complete version of this work was introduced by Maurice Nivat and André Arnold in [139]. An overview of the evolution can be found in [16] and a complete recompilation with examples can be found in [15].

In the work proposed by Arnold & Nivat, a system can be fully described as a set of interacting processes. At the base, a process is represented by a *transition system* which consists of a set of possible states for the process and a set of transitions, or state changes, which the system can execute.

Definition 10 Transition System *A transition system is a quadruple $\mathcal{A} =< S, T, \alpha, \beta >$ where:*

- *S is a finite or infinite set of states,*
- *T is a finite or infinite set of transitions, α and β are two mappings from t to S which take each transition t in T to the two states $\alpha(t)$ and $\beta(t)$, respectively the source and the target of the transition t.*

A transition system is *finite* if S and T are finite. Unless explicitly stated otherwise, it is assumed that only finite transition systems are considered.

In particular, each process in the Arnold & Nivat's approach is described using a *Labelled Transition System* (LTS) formally defined by Arnold [15] as:

Definition 11 Labelled Transition System (by Arnold & Nivat) *A transition system labelled by an alphabet A is a 5-tuple $\mathcal{A} =< S, T, \alpha, \beta, \lambda >$ where*

- *$< S, T, \alpha, \beta >$ is a transition system,*
- *λ is a mapping from T to A taking each transition t to its label $\lambda(t)$*

Intuitively, the label of a transition indicates the action or the event which triggers the transition.

The allowed interactions between the processes forming a system are given through a *synchronisation constraint* defined as:

Definition 12 Synchronisation constraint *Let $A_1, \ldots, A_n$ be alphabets representing actions or events, a synchronisation constraint is a subset of the Cartesian product $A_1 \times \ldots \times A_n$.*

Each element of the Cartesian product is a *synchronisation vector* representing a global action of the system of processes. This synchronisation constraint in the Arnold & Nivat proposal is constant. It does not vary during the evolution of the system, in particular it does not depend on the state of the system or in one of its processes.

A transition system encoding the behaviour of a system itself is obtained through the *synchronous product*. The *synchronous product* combines the LTSs of the processes forming the system taking into account the synchronisation constraints. Before giving a formal definition of the synchronous product, Arnold defines the *free product* of transition systems, where there is no interaction between the processes as:

Definition 13 Free Product of Transition Systems *Consider n transitions systems $\mathcal{A}_i =< S_i, T_i, \alpha_i, \beta_i >, i = 1, \ldots, n$. The free product $\mathcal{A}_1 \times \ldots \times \mathcal{A}_n$ of those n transition systems is the transition system $\mathcal{A} =< S, T, \alpha, \beta >$ defined by:*

$$\begin{aligned} S &= S_1 \times \ldots \times S_n, \\ T &= T_1 \times \ldots \times T_n, \\ \alpha(t_1, \ldots, t_n) &= < \alpha_1(t_1), \ldots, \alpha_n(t_n) >, \\ \beta(t_1, \ldots, t_n) &= < \beta_1(t_1), \ldots, \beta_n(t_n) >, \end{aligned}$$

Then the *synchronous product* is defined as:

Definition 14 Synchronous Product *If $\mathcal{A}_i, i = 1, \ldots, n$, are n transition systems labelled by alphabets $\mathscr{A}_i$, and if $I \subseteq \mathscr{A}_1 \times \ldots \times \mathscr{A}_n$ is a synchronisation constraint, the synchronous product of the $\mathcal{A}_i$ under I, written $< \mathcal{A}_1, \ldots, \mathcal{A}_n; I >$, is the transition subsystem of the free product of the $\mathcal{A}_i$ containing only the global transitions $< T_1, \ldots, t_n >$ whose vectors of labels $< \lambda_1(t_1), \ldots, \lambda_n(t_n) >$ are elements of I.*

In other words, the synchronous product allows only those global transitions corresponding to action vectors included in the synchronisation constraint.

2.1.4 The transformations Lotomaton

Lotomaton [107, 135] is an extension to LOTOS to express *contexts*. LOTOS [104] is a specification language to describe systems strongly based on Process Algebra concepts and operators (we give an overview of LOTOS in section 2.2.3). Because of this strong relation and since we have not introduced LOTOS yet, we introduce here the main concepts of Lotomaton using process algebras even if it was developed specifically to LOTOS.

A *context* represents an environment where a system, specified using process algebras, will execute. A context itself is specified with process algebras operators but leaving

some "holes" in the specification. Those holes will be eventually filled with the processes that will execute on this context (environment).

A *context* can be simply viewed as a process expression (the context) interacting with a set of *undefined* sub-expressions (the processes). By consequence, the behaviour of contexts cannot be given as a simple LTS acting over actions, but as a LTS acting over other LTSs. The LTS defining the behaviour of a context is called a *transducer*.

A transducer may act as a controller of LTSs by "consuming" their actions, i.e. the transducer transitions are fired by the actions executed on its argument LTSs. In this case the transducer is a *controller automaton* and the LTSs its *argument automata*. Informally:

Definition 15 A transducer $P = \langle A, \mathcal{R} = \{B_1, \ldots, B_N\}\rangle$ *is an automaton* A *that may consume the actions executed by a set of behaviours* $\mathcal{R} = \{B_1, \ldots, B_n\}$. *Note that* $\mathcal{R}$ *can be empty.*

The transducers allow expressing the behaviour of systems in changing (dynamic) contexts. The semantics of transducers and its application to LOTOS can be found in [107].

2.1.5 Symbolic Transition Graphs

M. Hennessy and H. Lin [93] extend the standard notion of transition graphs to *symbolic transition graphs* which are a more abstract description of processes in terms of *symbolic actions*

Symbolic transition graphs are parameterized on a number of syntactic categories. The first two are a countable set of *variables*, $Var = \{x_0, x_1, \ldots\}$, and a set of values V. $Eval$, ranged over by ρ, represents the set of *evaluations*, i.e. the set of total functions from Var to V. A substitution is a partial injective mapping from Var to Var whose domain is finite. Sub represents the set of substitutions and this set is ranged over by σ.

They also presume a set of *expressions*, Exp, ranged over by e, which includes Var and V. Each e has an associated set of free variables, $fv(e)$, and it is assumed that both evaluations and substitutions behave in a reasonable manner when applied to expressions; the applications of ρ to e, denoted $\rho(e)$, yields to a value while the application of a substitution, denoted $e\sigma$, yields another expression with the property that $fv(e\sigma) = \sigma(fv(e))$ where the latter is defined in the obvious manner. $BExp$ is a set of boolean expressions, ranged over by b, with similar properties.

Essentially the symbolic transition graphs are arbitrary directed graphs in which nodes are labelled by a set of variables, intuitively the set of free variables of that node. The branches are labelled by *guarded actions*, pair of boolean expressions and actions. An action may be an *input* action of the form $c?x$, where c is from a set of channels, $Chan$; an output action of the form $c!e$; or a *neutral* action such as τ. So let $SyAct$, ranged over by α, represent the set of symbolic actions; it has the form:

$$SyAct = \{c?x, c!e | c \in Chan\} \cup \tau$$

The set of free and bound variables of these actions are defined in the obvious manner: $fv(c!e) = fv(e), bv(c?x) = \{x\}$, otherwise both $fv(\alpha)$ and $bv(\alpha)$ are empty. The set of guarded actions is given by:

$$GuAct = \{(b, \alpha) | b \in BExp, \alpha \in SyAct\}$$

Definition 16 Symbolic Transition Graphs. *A symbolic transition graph is a directed graph in which each node n is labelled by a set of variables $fv(n)$ and every branch is labelled by a guarded action such that if a branch labelled by (b, α) goes from node m to n, which we write as $m \xrightarrow{b,\alpha} n$, then $fv(b) \cup fv(\alpha) \subseteq fv(m)$, and $fv(n) \subseteq fv(m) \cup bv(\alpha)$.*

A symbolic transition graph may be looked upon as a particular austere representation of the abstract syntax of a value-passing process algebra. Central to the approach of Hennessy and Lin is the development of both *late* and *early symbolic operational semantics* where symbolic actions such as $c?x$, $c!e$ and their associated residual are associated with *open terms*.

If these terms are interpreted by assigning values to the free variables, then concrete operational semantics can be given in terms of the concrete actions $c?v$ and $c!v$ and this leads to a concrete bisimulation equivalence between terms (late and early version). Indeed, Hennessy and Lin give late and early concrete operational semantics of CCS - closed terms. If ρ is used to range over assignments of values to free variables, Hennessy and Lin obtain relations of the form:

$$\rho \models t \sim_E u \quad \text{and} \quad \rho \models \sim_L u$$

Intuitively these mean that with respect to the assignment ρ, t is early/late bisimulation equivalent to u.

Hennessy and Lin define two *symbolic bisimulation equivalences*, a late and early version. These are parameterized in boolean expressions having relations of the form $\simeq^b_E$ and $\simeq^b_L$ between open terms. For example, $t \simeq^b_E u$ indicates that regarding to the early version of the symbolic operational semantics, t and u are bisimulation equivalent relative to the boolean expression b (i.e. in every interpretation which satisfies b, the processes t and u are bisimulation equivalent). They show that:

> $t \simeq^b_i u$ if and only if for every assignment ρ which satisfies
> the boolean b $\rho \models t \sim_i u$, where i is either E or L.

Finally they provide algorithms for both late and early symbolic bisimulations. In particular the algorithms return the weakest boolean for which $t \simeq^b_i u$, i.e. reducing bisimulation equivalence to the logical equivalence of boolean expressions.

2.1.6 Symbolic Transition Graph with Assignment

H. Lin [111] extends the notion of *Symbolic Transition Graph* (STG for short) by allowing *assignments* to be carried in transitions. Lin postulates that many intuitively simple processes can not be pictured as finite STG.

For example, given the following definition:

$$P(x) \stackrel{\text{def}}{=} c!x.P(x+1)$$

the process $P(0)$ first output, along the channel c, the integer 0, then 1, 2, 3, ... The STG for $P(x)$ has countably many states representing $P(x+1)$, $P(x+2)$, $P(x+3)$, ..., with edges $P(x+n) \xrightarrow{true, c!x+n} P(x+n+1)$ for $n \geq 0$, which is clearly an infinite graph if we let x range over all the Integers.

The main purpose of Lin in his work is to generalise the notion of symbolic transition graph so that processes like $P(x)$ above can be associated with finite state graphs. This is achieved by introducing *assignments* into labels. An edge now takes the form $n \xrightarrow{b, \overline{x}:=\overline{e}, \alpha} n'$, where, besides a boolean condition b and an abstract action α, there is also an assignment $\overline{x} := \overline{e}$. Roughly it means if b is evaluated to *true* at node n then the action α can be fired, and, after the transition, the free variable $\overline{x}$ at node n' will have the values of $\overline{e}$ *evaluated at* n. With such extension the graph for $P(x)$ has only one node.

Lin calls such a graph *symbolic transition graph with assignment* (STGA for short). Note that a STG becomes a special case of STGA with trivial assignment (identity mapping).

Let ι range over a set of *base types*. Lin presupposes the following syntactic categories:

$v, \ldots \in Val_\iota$:	a set of data values
$x, \ldots \in DVar_\iota$:	a countable set of data variables
$e, \ldots \in DExp_\iota$:	a set of data expressions
$b, \ldots \in BExp$:	a set of boolean expressions
$c, \ldots \in Chan$:	a set of channel names

It is assumed that both Val_ι and $DVar_\iota$ are included in $DExp_\iota$, and that $e = e' \in BExp$ for any $e, e' \in DExp_\iota$. $BExp$ is equipped with the usual operators $\wedge, \vee, \neg, \Rightarrow$ and $\forall$.

An *evaluation* $\rho \in Eval$ is a type-respecting mapping from Var to Val. An application of ρ to a data expression e , denoted $\rho(e)$, always yields a value from Val and similarly for boolean expressions; $\rho(b)$ is either true or false.

A *substitution* σ is a type-respecting mapping from data variables to expressions and $e\sigma$ denotes the result of applying σ to the expression e.

An *assignment* θ has the form $\overline{x} := \overline{e}$, with $\overline{x}$ and $\overline{e}$ having the same length and type.

An *action* is either a silent action τ, an input action $c?x$, or an output action $c!e$, where $c \in Chan$. Actions are ranged over by α. The set of free and bound variables of actions are given by $fv(c!e) = fv(e)$, $bv(c?x) = \{x\}$, and empty otherwise. The set of channel names used in an action is defined by $chan(c?x) = chan(c!x) = \{c\}$ and $chan(\tau) = \emptyset$. A *guarded action with assignment* is a triple (b, θ, α) where b is a boolean expression, θ an assignment, and α an action.

Definition 17 Symbolic Transition Graph with Assignments *A symbolic transition graph with assignments (STGA for short) is a rooted directed graph where each node n has an associated finite set of free variables $fn(n)$ and each edge is labelled by a guarded action with assignment. A STGA is* well-formed *if whenever $(b, \overline{x} := \overline{e}, \alpha)$ is the*

label of an edge from n to m, written $n \xrightarrow{b,\overline{x}:=\overline{e},\alpha} m$, *then* $fv(b,\overline{e}) \subseteq fv(n), fv(\alpha) \subseteq \{\overline{x}\}$, *and* $fv(m) \subseteq \{\overline{x}\} \cup bv(\alpha)$.

The transition $n \xrightarrow{true,\theta,\alpha} m$ is written simply as $n \xrightarrow{\theta,\alpha} m$, and θ is omitted when it is the identity assignment on $fv(n)$.

Lin distinguishes an *eager* approach to build a STGA from a process description language in which substitutions are performed when moves are inferred. In a recursively defined process term $P(\overline{e})$ with definition clause $P(\overline{x}) \stackrel{\text{def}}{=} t$ the way to infer moves is to substitute $\overline{e}$ for $\overline{x}$ in t and look for moves from the result term. A disadvantage with this approach is that very often it results in infinite state graphs. An example is the process $P(0)$ where $P(x) \stackrel{\text{def}}{=} c!x.P(x+1)$. Though simple, using this approach for the symbolic graph $P(0)$ will have infinite number of nodes representing $P(0), P(0+1), P(0+1+1), \ldots$.

Lin prefers a *lazy* approach: the substitutions necessary in inferring moves from recursively defined terms are postponed. They are carried in the transitions and will be performed later when processes are compared for bisimulations. For regular value-passing *CCS* [130], it is not difficult to see that graphs generated with this new approach are always finitely, because any process term involves only finite recursive definition clauses and each such clause rises a finite subgraph.

Then he introduces a late ground operational semantics rules with respect to an evaluation ρ supplying values for free node variables, and he defines a late ground bisimulation symmetric relation between two graphs with respect to two evaluations (one for each graph).

Lin also introduces a more abstract operational semantics to STGAs without referring to evaluations, called *symbolic* operational semantics and defined in *terms*. It is called "symbolic" because boolean and data operations do not get evaluated, instead they are symbolically carried in transitions. Having this symbolic operational semantics, Lin defines (late) *symbolic bisimulation* as in the case of STG [93].

Finally Lin provides an algorithm for late symbolic bisimulation of STGAs which returns a predicate equation system. The problem of deciding bisimulation of two STGAs is reduced to the problem of checking validity for the greatest solution of a closed equation system.

2.2 Languages

Together with the development of process algebra theories during the eighties and nineties, appears some tools that allow automatic or semi-automatic reasoning about concurrent systems based on those algebras (we review some of such tools in section 2.3).

These tools need input languages describing the system's behaviour to analyse. For this propose several languages where developed, some of them in a general approach and others for some specific domain areas such as telecommunications or real-time. They are also present in different levels of abstractions, usually classified in *low-level models* such as format FC2 [36, 118], Kripke structures [68] and Petri nets [6]; *intermediate formats* such as Input/Output automata [113], Symbolic Transition Systems [93], IF v2.0 [114]

and NTIF [80]; and *high-level* languages such as Promela [98, 85], LOTOS [104] and E-LOTOS [105].

In this section we overview three of them which we believe are the most powerful and/or most used description languages.

2.2.1 FC2

The FC2 format [36, 118] was originally designed to interface several preexisting verification tools [117]. In this way these heterogeneous tools could be further developed independently, while being used in cooperation with their complementary features.

The format allows description of labelled transition systems and networks of communicating automata. While the format is not "syntax-friendly" (as it represent objects which are supposedly obtained by translation or compilation), it is still reasonably natural: automata are tables of states, states being each in turn a table of outgoing transitions with target indexes; networks are vectors of references to subcomponents (i.e., to other tables), together with synchronisation vectors (legible combinations of subcomponent behaviours acting in synchronised fashion). Subcomponents can be networks themselves, allowing hierarchical descriptions.

In addition a permissive labelling discipline allows a variety of annotations on all distinct elements: states, transitions, automata and networks as a whole. It is through this labelling that behavioural action labels are provided of course, but also structural information for source code retrieval, logical model-checking annotation and even private hooked informations. Annotative labels are dealt as regularly as possible in the syntax, in simple form at predictable location, so that they can be treated smoothly by any tool at parsing time, often by simply disregarding them if they do not address the tool's specific functionalities. The actual labelling contents are stored in tables forming the objects headers, so that only integers referencing to table entries are actually present in object bodies themselves (automata or networks). Finally, labels can be structured by simple operators (sum, product and several others) to allow richer information. We introduce the syntax of FC2 in section 6.5.1.

FC2 simple example

```
nets 1
hook "main">0
net 0
struct "t1" logic "initial">0
hook"automaton"
vertice 2
vertex0 struct"v0" edges2
behav"a" result 0
behav"b" result 1
v1s"v1"E2
b"a"r0
b"b"r1
```

For example, this FC2 object contains a single net (indeed an automaton, as indicated in the net hook). The automaton has two vertices with two edges each. All information attached appears here directly in place, though it could have been tabulated. The text for the first vertex appears in long form (more readable), while the second vertex is in a short, compact, form.

2.2.2 Promela

Promela (Process Meta Language) [98, 85] is a language designed to describe distributed systems, specifically data communication protocols ones. The language allows

dynamic creation of concurrent processes. Communication via message channels can be defined to be synchronous (i.e., rendezvous), or asynchronous (i.e., buffered).

Promela is a verification modelling language. It provides means for making abstractions of protocols (or distributed systems in general) that suppress details that are unrelated to process interaction.

Promela programs consist of *processes*, message *channels*, and *variables*. Processes specify behaviour, channels and global variables define the environment in which the processes run.

Processes and variables must be declared before they can be used. Variables can be declared either locally, within a process type, or globally. A process can only be declared globally in a `proctype` declaration. `Proctype` declarations *cannot be nested*.

A processes declaration in Promela has the form:

```
proctype pname ( chan In, Out; byte id )
   { statements  }
```

The body of a process declaration, `statements`, defines the behaviour of the process. In Promela there is no difference between conditions and statements, even isolated boolean conditions can be used as statements. The execution of every statement is conditional on its executability. Statements are either executable or blocked. The executability is the basic means of synchronisation. A process can wait for an event to happen by waiting for a statement to become executable. For instance, instead of writing a busy wait loop:

```
while (a != b)
   skip   /* wait for a==b */
```

one can achieve the same effect in Promela with the statement

```
(a == b)
```

A condition can only be executed (passed) when it holds. If the condition does not hold, execution blocks until it does.

There are sixteen types of statements:

`assert`	`assignment`	`atomic`	`break`
`declaration`	`d_step`	`else`	`expression`
`goto`	`receive`	`selection`	`skip`
`repetition`	`send`	`timeout`	`unless`

The execution of a statement is conditional on its enabledness (or "executability"). Statements are either enabled or blocked. Of the above listed statements, assignments, declarations, assert, skip, goto and break are always enabled. If a statement is blocked, execution at that point halts until the statement becomes enabled.

`assert( expression )` aborts the program if the expression returns a zero result; otherwise it is just passed.

`atomic { statements }` attempts to execute the statements in one indivisible step; i.e., without interleaved execution of other processes. An atomic statement is enabled if its first statement is. `dstep { statements }` has the same effect as `atomic`.

A `selection` statement begins with the keyword `if`, followed by a list of one or more options, and ends with the keyword `fi`. Every option begins with the flag `::` followed by any sequence of statements. An option can be selected if its *first* statement (the *guard*) is enabled. A selection blocks until there is at least one selectable branch. If more than one option is selectable, one will be selected at random. The special guard `else` can be used (once) in selection and repetition statements and it is enabled precisely if all other guards are blocked.

A `repetition` statement is similar to a `selection` statement, but is executed repeatedly until either a `break` statement is executed or a `goto` jump transfer control outside the cycle. The keywords of `repetition` statement are `do` and `od`.

`goto label` transfers control to the statement labelled by `label` which has to occur in the same procedure as the `goto`.

`skip` has no effect and is mainly used to satisfy syntactic requirements.

A `timeout` statement becomes enabled precisely when every other statement in the system is blocked.

{ `statements-1` } `unless` { `statements-2` } starts execution in `statements-1`. Before each statement in `statements-1` (including the first one) is executed, *enabledness* of `statements-2` is checked and if it is, execution of `statements-1` is aborted and control is transfered to `statements-2`; otherwise, control remains in `statements-1`. If `statements-1` terminates, statements-2 is ignored.

Communication between processes

`send` and `receive` statements are used to communicate processes. Message channels are used to model the transfer of data from one process to another. They are declared either locally or globally, for instance:

```
chan qname = [16] of { short }
```

This declares a channel that can store up to 16 messages of type short. Channel names can be passed from one process to another via channels or as parameters in process instantiations. A channel behaves as a *FIFO queue*

The statement `qname!expr` sends the value of the expression `expr` to the channel `qname`, that is: it appends the value to the tail of the channel. The statement blocks if the channel is full.

The statement `qname?msg` retrieves a message from the head of the channel `qname` and stores it in a variable `msg`.

Note that a channel size can be set to 0 to define a rendez-vous communication. A channel with size 0 can pass single messages, but cannot store them.

Processes in Promela are instantiated through a `run` operation of the form:

```
run pname(Transfer, Device[0], 0)
```

It first assigns the actual parameters to the formal ones and then executes the statements in the body. Each process instance has a unique, positive instantiation number. A process-instance remains active until the process' body terminates (if ever).

A process declaration prefixed with the `active [N]` modifier causes N instances of the process to be active in the initial system state. Formal parameters of instances activated through the `active` modifier are initialised to 0.

```
init { statements }
```

The process `init`, if present, is instantiated once, and is often used to prepare the true initial state of a system by initialising variables and running the appropriate process-instances.

2.2.3 LOTOS

LOTOS (Language of Temporal Ordering Specification) is one of the two Formal Description Techniques (FDT) [104, 103] developed within ISO (International Standards Organisation) for the formal specification of open distributed systems, and in particular for those related to the Open Systems Interconnection (OSI) computer network architecture [102]. It was developed by FDT experts from ISO/TC97/SC21/WG1 ad-hoc group on FDT/Subgroup C during the years 1981-86. The basic idea that LOTOS developed from was that systems can be specified by defining the temporal relation among the interactions that constitute the externally observable behaviour of a system. Contrary to what the name seems to suggest, this description technique is not related to temporal logic, but is based on process algebraic methods. Such methods were first introduced by Milner's work on CCS [130, 131], soon to be followed by many closely related theories that are often collectively referred to as process algebras, e.g. [31, 96]. More specifically, the component of LOTOS that deals with the description of process behaviours and interactions has borrowed many ideas from [130, 96].

LOTOS also includes a second component, which deals with the description of data structures and value expressions. This part of LOTOS is based on the formal theory of abstract data types, and in particular the approach of equational specification of data types, with an initial algebra semantics. Most concepts in this component were inspired by the abstract data type technique ACT-ONE [69], although there are a number of differences.

Even when LOTOS has been developed particularly for OSI, it is an FDT generally applicable to distributed, concurrent, information processing systems.

In LOTOS a distributed, concurrent system is seen as a *process*, possibly consisting of several sub-processes. A sub-process is a process in itself, so that in general a LOTOS specification describes a system via a *hierarchy of process definitions*. A *process* is an entity able to perform *internal, unobservable actions*, and to interact with other processes, which form its *environment*. Complex interactions between processes are built up out of elementary units of synchronisation called *events*, or (*atomic*) *interactions*, or simply *actions*.

Events imply process synchronisation, because the processes that interact in an event (they may be two or more) participate in its execution at the same moment in time. Such synchronisations may involve the exchange of data. Events are *atomic* in the sense that they occur instantaneously, without consuming time. An event is thought of as occurring at an interaction point, or *gate*, and in the case of synchronisation without data exchange, the event name and the gate name coincide.

The environment of a process P, in a system S, is formed by the set of processes of S with which P interacts, plus an unspecified, possibly human, *observer* process, which is assumed to be always ready to observe anything observable the system may do. And, to be consistent with the model, observation is nothing but interaction.

Basic LOTOS is a simplified version of the language employing a *finite* alphabet of observable actions. This is because observable actions in basic LOTOS are identified only by the name of the gate where they are offered, and LOTOS processes can only have a finite number of gates

The structure of actions will be enriched in *full* LOTOS by allowing the association of data values to gate names, and thus the expression of a possibly infinite alphabet of observable actions.

Basic LOTOS only describes process synchronisation, while full LOTOS also describes interprocess value communication. In spite of this remarkable difference, we initially concentrate on basic LOTOS because within this proper subset of the language we can appreciate the expressiveness of all the LOTOS process constructors (operators) without being distracted by interprocess communication.

The typical structure of a basic LOTOS *process definition* is given in Figure 2.1.

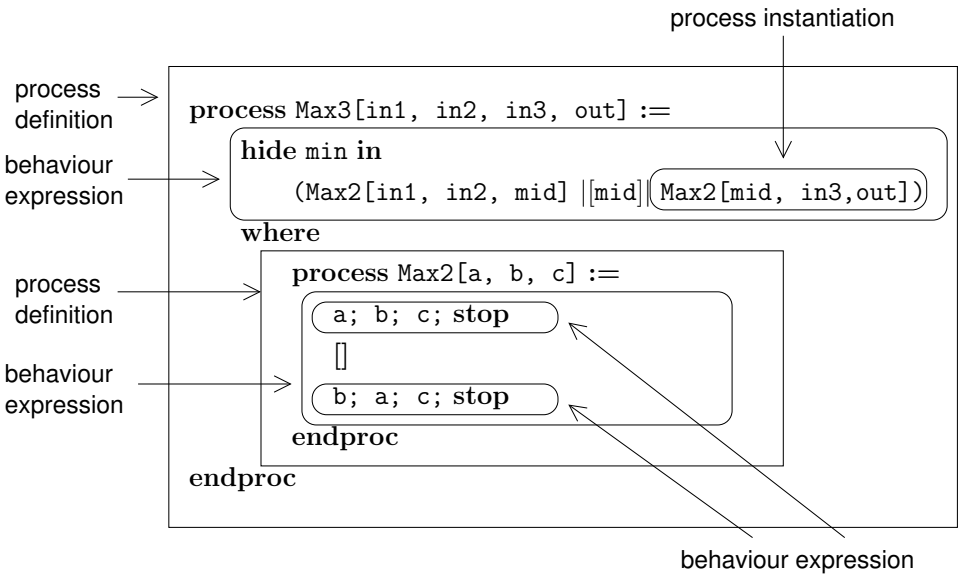


Figure 2.1: Example of a process definition in LOTOS

An essential component of a process definition is its `behaviour expression`. A behaviour expression is built by applying an operator (e.g., `[]`) to other behaviour expressions. A behaviour expression may also include *instantiations* of other processes (e.g. `Max2`), whose definitions are provided in the `where` clause following the expression. Given behaviour expression B, it is also called for convenience "a process", even when no process name is explicitly associated with the behaviour expressed by B.

The complete list of basic-LOTOS behaviour expressions is given in Table 2.1, which includes all basic-LOTOS operators. Symbols B, B1, B2 in the table stand for any behaviour expression. Any behaviour expression must match one of the formats listed in column **Syntax**.

Name	**Syntax**
inaction	**stop**
action prefix	
- unobservable (internal)	**i**;B
- observable	g;B
choice	B1 [] B2
parallel composition	
- general case	B1 |[$g_1, \ldots, g_2$]| B2
- pure interleaving	B1 ||| B2
- full synchronisation	B1 || B2
hiding	**hide** $g_1, \ldots, g_n$ **in** B
process instantiation	p [$g_1, \ldots, g_n$]
successful termination	**exit**
sequential composition (enabling)	B1 >> B2
disabling	B1 [> B2

Table 2.1: Syntax of behaviour expressions in LOTOS

By inspecting Table 2.1 we may observe that basic LOTOS includes *nullary* operators (e.g. inaction), *unary* operators (e.g. action prefix) and *binary* operators (e.g. parallel composition), that is, operators applicable to, respectively, none, one and two *behaviour expressions*.

The completely inactive process is represented by **stop**. It cannot offer anything to the environment, nor it can perform internal actions.

The unary prefix operator produces a new *behaviour expression* out of an existing one, by prefixing the latter with an action (gate name) followed by a semicolon.

If B1 and B2 are two behaviour expressions then B1 [] B2 denotes a process that behaves either like B1 or like B2. The choice offered is resolved in the interaction of the process with its environment. If (another process in) the environment offers an initial observable action of B1, then B1 may be selected, and if the environment offers an initial observable action of B2, then B2 may be selected. If an action is offered from the environment that is initial to both B1 and B2, then the outcome is undetermined.

In B1 |[S]| B2 with S a list of gates common to both B1 and B2, the parallel composition expression is able to perform any action that either component expression is ready to perform at a gate not in S (excluding successful termination), or any action that both components are ready to perform at a gate in S. This implies that when process B1 is ready to execute some action at one of the synchronisation gates (in S), it is forced, in the absence of alternative actions, to wait until its "partner" process B2 offers the same action.

B1 || B2 is similar to B1 |[S]| B2, but S is the set of "all the gates" common to both processes, i.e. both B1 and B2 should synchronise actions in all their shared gates. On the contrary, B1 ||| B2 is similar to B1 |[S]| B2, but S is an empty set.

The hiding operator, **hide** $g_1, \ldots, g_n$ **in** B, renames all the actions on the gates $g_1, \ldots, g_n$ of B to the unobservable action **i**.

Process instantiation instantiates a process with the given parameters. Recursion is achieved in LOTOS by allowing the instantiation of a process within its own *behaviour expression* definition.

exit is a nullary operator whose propose is to perform the successful termination of the process, after which it becomes the dead process **stop**.

The interpretation of the expression `B1 >> B2` is that if `B1` terminates successfully, and not because of a premature deadlock, then the execution of `B2` is enabled.

The disabling operator, `B1 [> B2` was introduced in LOTOS to encode a disturbing event in the "normal" course of actions (e.g. disconnection or abortion of a connection). It means that `B1` may be interrupted by the first action of `B2`, in which case the control is transfered to `B2`; or `B1` successfully terminates, in which case `B2` disappears (is disabled).

The representations of values, value expressions and data structures in LOTOS are derived from the specification language for abstract data types (ADT) ACT ONE [69]. It does not indicate how data values are actually represented and manipulated in memory, but only defines the essential properties of data and operations that any correct implementation (concrete data type) is required to satisfy. The reader interested in further details about ADT may refer to [69].

While in basic LOTOS an observable action coincides with a gate name, in full LOTOS (or, simply, LOTOS) it is formed by a gate name followed by a list of zero or more values offered at that gate: $\mathtt{g} < \mathtt{v}_1 \ldots \mathtt{v}_n >$. For example:

```
g <TRUE, "tree", 3>
```

is the observable action offering the boolean value TRUE, character string "`tree`", and natural number `3` at gate `g`. Since the offered values may range over infinite sets (e.g. the natural numbers), an infinite number of observable actions is expressible in full LOTOS.

Having the facilities for defining and describing values in LOTOS, behaviour expressions may now depend on *conditions* on values. Such conditions are expressed as equations that relate two value expressions: the condition is met if the two expressions evaluate to the same value, in the data type environment of that condition (the condition can be also an expression of type boolean).

Full LOTOS offers the possibility to parameterise process definitions not only in terms of formal gates (as is the case with basic LOTOS) but also in terms of a parameter list, which is a list of variable declarations.

The LOTOS specification [104] defines the its full syntax as well as its semantics on terms of SOS rules.

E-LOTOS[105] is an enhanced version of LOTOS introducing the notion of time (needed for instance to model real-time systems).

2.2.4 Unified Modelling Language

In the description of system's behaviours we cannot leave behind the Unified Modelling Language (UML) [142], which is a widely adopted language for designing systems in the industry.

UML provides seven diagrams capturing the variety of interactions and discrete behaviours of single entities within a model as it "executes" over time:

The use case diagram captures the requirements of a system by means of communicating with users and other stakeholders about what the system is intended to do.

An activity diagram is used to display the sequence of activities. Activity Diagrams show the workflow from a start point to the finish point detailing the many decision paths that exist in the progression of events contained in the activity.

A state machine diagram models the behaviour of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events.

A sequence diagram is a form of interaction diagram which shows objects as lifelines running down the page and with their interactions over time represented as messages drawn as arrows from the source lifeline to the target lifeline. Sequence diagrams are good at showing which objects communicate with which other objects and what messages trigger those communications. Sequence diagrams are not intended for showing complex procedural logic.

A communication diagram , formerly called a collaboration diagram, it is an interaction diagram that shows similar information to sequence diagrams but its primary focus in on object relationships.

Timing diagrams are used to display the change in state or value of one or more elements over time. It can also show the interaction between timed events and the time and duration constraints that govern them.

An interaction overview diagram is a form of activity diagram in which the nodes represent interaction diagrams. Interaction diagrams can include sequence, communication, interaction overview and timing diagrams.

All those diagrams are accompanied in the UML specification [142] with an informal semantics given in natural language.

UML is a language for the specification of systems at design time and, in the experience of the author, an excellent language for documentation as well. It provides means for describing both the structure of an application and a high level view of how this application should behave.

However, despite its success as being an unified and visual notation in the industry, it is recognised that UML still lacks from formal semantics [41, 73]. The Object Constraint Language (OCL [140]) completes the syntactic rules of the UML metamodel and improves the precision of the notation, but it is not enough to check and to validate UML models [147]. Although, since the apparition of UML 2.0, many works address this lack of formal semantics like in [66] where the authors propose a formalisation of sequence diagrams and the verification of their coherence with the state machines using π-calculus.

2.3 Verification tools

As we mention early in the section above, several tools for the reasoning of system's behaviour have been developed. Some of the most know are FC2Tools [36, 10], Labelled Transition System Analyzer [122], SPIN [98, 97], dSPIN [63] and CADP [81].

In this section we review three of them which we believe are the best suited to verify distributed systems.

2.3.1 FC2Tools

FC2Tools [36, 10] is a set of construction, reduction, analysis and diagnostics tools for communicating systems described using the FC2 format.

The verification tools comprise a number of stand-alone tools, each implementing some well-defined functionalities. Tools may be used in succession through the common FC2 file description format.

The set of tools is made of: Autograph editor, for the graphical edition and display of automata and networks of communicating automata; `fc2explicit`, for manipulation of enumerated finite state machines; `fc2implicit`, for manipulation of symbolic finite state machines; and `fc2link` to merge systems (usually large hierarchical systems) described in separated FC2 files.

The main *functional modules* of the toolset are:

graphical description of the network and behaviour of communicating agents The graphical editor Autograph allows to draw such descriptions much in the usual fashion of process algebraic terms, and then produces FC2 format representations.

linking of multifile descriptions Large hierarchical system descriptions can be split between different files (for instance as different Autograph windows). The tabulated naming informations in resulting FC2 files need not be consistent across files, and so merging these partial descriptions into a single file for later analysis takes some bookkeeping care.

construction of "some form of " global model Model-based automatic verification relies on expansion of network into a global state-transition model (synchronisation product of section 2.1.3). Two main implementation techniques can be used here: the extensional approach with a classical representation of expanded automata with enumerated states and transitions; the symbolic approach, based on implicit representation by Binary Decision Diagrams of state sets(only), while representation of the full transition relation is avoided, and remain parted by possible events.

reduction of the model FC2Tools supports reduction by weak, branching and strong bisimulation equivalence.

specification of properties and model-checking FC2Tools finds the existence of deadlock, livelock or divergent states. More refined properties are expressed in FC2Tools as abstraction automata. They are labelled transition systems with logical predicates in their labels, and with acceptance states. Each acceptance

state defines one *abstract action*, representing a set of traces (a regular language) from the actions of the model being checked. Note that abstraction automata can be specified graphically using the Autograph editor.

diagnostics FC2Tools provides a diagnostic (or counterexample) when analysing deadlocks, livelocks or reachability properties expressed as abstraction automata.

toplevel object management Successive object transformations can be applied while intermediate representations are kept and gathered on demand in a graphical environment, for later reuse.

In section 3.3 we analysis a distributed system using FC2Tools. Other case studies using FC2Tools can be found in [38, 119, 87, 135].

2.3.2 SPIN

SPIN [98, 97] is a generic model-checking tool that supports the design and verification of asynchronous process systems. Given a set of correctness claims and a system description, SPIN verifies whether or not those claims hold in the system.

SPIN accepts design specifications written in the verification language Promela [85, 98], and it accepts correctness claims specified in the syntax of standard Linear Temporal Logic (LTL) [123].

SPIN translates each process template into a finite automaton. The global behaviour of the concurrent system is obtained by computing an asynchronous interleaving product of automata, one automaton per asynchronous process behaviour. The resulting global system behaviour is itself again represented by an automaton. This interleaving product is often referred to as the state space of the system, and, because it can easily be represented as a graph, it is also commonly referred to as the global reachability graph.

To perform verification, SPIN takes a correctness claim that is specified as a LTL formula, converts that formula into a Büchi automaton (as showed by Vardi and Wolper in [153]), and computes the synchronous product of this claim and the automaton representing the global state space. The result is again a Büchi automaton. If the language accepted by this automaton is empty, this means that the original claim is not satisfied for the given system. If the language is nonempty, it contains precisely those behaviours that satisfy the original temporal logic formula.

SPIN uses the correctness claims to formalise erroneous system behaviours, i.e. behaviours that are undesirable. A positive claim requires to prove that the language of the system (i.e. all its executions) is included in the language of the claim. A negative claim, on the other hand, requires to prove that the intersection of the language of the system and of the claim is empty. The size of the state space for a language inclusion proof is at most the size of the Cartesian product of the (automata representing) system and claim, and at least the size of their sum. The worst-case state space size to prove emptiness of a language intersection is still the size of the Cartesian product of system and claim, but, in the best case, it is zero. Note that if no initial portion of the invalid behaviour represented by the claim appears in the system, the intersection

contains no states. SPIN, therefore, works with negative correctness claims and solves the verification problem by language intersection.

The entire computation, starting from the individual concurrent components and a single Büchi automaton representing the correctness claim, is done by SPIN in one single procedure, using a nested depth-first search algorithm [55, 99]. The algorithm terminates when an acceptance cycle is found (which then constitutes a counterexample to a correctness requirement), or, when no counterexample exists, when the complete intersection product has been computed.

The nested depth-first search algorithm does not preserve the capability to detect all possible acceptance cycles that may appear in the reachability graph. It can, however, be proved to detect at least one such cycle if one or more cycles exists. Because acceptance cycles in SPIN constitute counterexamples to correctness claims, establishing either their absence or their presence always suffices for the purposes of verification.

To avoid the *state explosion* problem, SPIN uses a partial order reduction method [144] to reduce (*per formula*) the number of reachable states that must be explored to complete a verification. The reduction is based on the observation that the validity of an LTL formula is often insensitive to the order in which concurrent and independently executed events are interleaved in the depth-first search. Instead of generating an exhaustive state space that includes all execution sequences as paths, the verifier can generate a reduced state space, with only representatives of classes of execution sequences that are indistinguishable for a given correctness property.

In addition, SPIN uses state compression and hashing techniques to reduce the use of the memory.

SPIN is considered to be one of the most efficient and most widely used LTL model checking systems.

2.3.3 CADP

CADP [81] is a toolbox for protocol engineering. It offers a wide range of functionalities, ranging from interactive simulation to the most recent formal verification techniques.

The CADP toolbox contains several closely interconnected components: Aldébaran, BCG, Caesar, Caesar.adt, Open/Caesar and XTL. All these components are accessible through a unified graphical user-interface named Eucalyptus. It accepts three different input formalisms:

- high-level protocol descriptions written in LOTOS [104]. The toolbox contains two compilers Caesar and Caesar.adt. They translate LOTOS descriptions into C code which can be used for simulation, verification and testing purpose.

- low-level protocol descriptions specified as Labelled Transition Systems (LTS), i.e., finite state machines with transitions labelled by action names.

- As an intermediate step, the CADP toolbox accepts networks of communicating automata [16], i.e. finite state machines running in parallel and connected together using many operators including LOTOS parallel composition, synchronisation vectors, label renaming and hiding operators.

The CADP toolbox allows to cover most of the development cycle of a protocol by offering an integrated set of functionalities. These functionalities are interactive or random simulation, partial and exhaustive deadlock detection, test sequences generation, verification of behavioural specifications with respect to a bisimulation relation, verification of branching-time temporal logic specifications.

All the validation and verification tools are based on a same principle consisting in the exploration of an LTS describing the exhaustive behaviour of the protocol under analysis. This LTS can be accessed through several representations: The explicit representation consists in the exhaustive list of the states and transitions of the LTS. A compact format (BCG) is available to encode explicit representations efficiently. The implicit representation consists in a C library providing a set of functions allowing a dynamic exploration of the LTS. It is well adapted to perform "on-the-fly" verification, avoiding the generation of the whole LTS.

The main components of the CADP toolset are:

bcg_min & bisimulator [128, 126, 30] allow the comparison and the reduction of LTSs modulo various equivalence relations (such as strong bisimulation, observational equivalence, delay bisimulation, or τ^*a bisimulation, branching bisimulation, and safety equivalence) and preorder relations (such as simulation preorder and safety preorder).

BCG (Binary-Coded Graphs) is both a format for the representation of explicit LTSs and a collection of libraries and programs dealing with this format. Compared to ASCII-based formats for LTSs, the BCG format uses a binary representation with compression techniques resulting in much smaller (up to 20 times) files. BCG is independent from any source language but keeps track of the objects (types, functions, variables) defined in the source programs. The following tools are currently available for this format: `bcg_io` performs conversions between the BCG format and a dozen of other formats (among them FC2 files describing a single automaton); `bcg_open` establishes a gateway between the BCG format and the Open/Caesar environment; `bcg_draw` provides a bi-dimensional graphical representation of BCG graphs with an automatic layout of states and transitions; and `bcg_edit` which is an interactive editor which allows to modify manually the display generated by `bcg_draw`.

Caesar [83] is a compiler which translates LOTOS descriptions into LTSs. Caesar proceeds in several steps, first translating the LOTOS description to compile into an intermediate Petri Net model, which provides a compact representation of the control and data flows. Then, the LTS is produced by performing reachability analysis on this Petri net. Caesar only handles LOTOS specifications with static control features, which is usually sufficient for most applications. The current version of Caesar allows the generation of large LTSs (some million states) within a reasonable lapse of time. The efficient compiling algorithms of Caesar can also be exploited in the framework of the Open/Caesar environment.

Caesar.adt [78] is a compiler that translates the data part of LOTOS descriptions into libraries of C types and functions. Each LOTOS sort or operation is trans-

lated into an equivalent C type or function. One must indicate to Caesar.adt which LOTOS operations are "constructors" and which are not (fairly obvious, in practise). Caesar.adt does not allow non-free constructors ("equations between constructors"). Translation of large programs (several hundreds of lines) is usually achieved in a few seconds. Caesar.adt can be used in conjunction with Caesar, but it can also be used separately to compile and execute efficiently large abstract data types descriptions.

Open/Caesar [77] is an extensible programming environment for the design of applications working with the implicit representation of LTSs. It provides a front-end with functions and types defined in the Open/Caesar's API. The API allows the use, among others, of the back-end tools including: the `caesar` and `caesar.adt` compilers, bisimulator, the `bcg_open` gateway for explicit graphs, the `exp.open` [109] gateway for networks of communicating automata, distributor [82] for generating the state space on a distributed environment, and evaluator [129] for checking temporal formulas.

XTL (eXecutable Temporal Language) is a functional-like programming language designed to allow an easy, compact implementation of various temporal logic operators. These operators are evaluated over an LTS encoded in the BCG format. Besides the usual predefined types (booleans, integers, etc.) The XTL language defines special types, such as sets of states, transitions, and labels of the LTS. It offers primitives to access the informations contained in states and labels, to obtain the initial state, and to compute the successors and predecessors of states and transitions. The temporal operators can be easily implemented using these functions together with recursive user-defined functions working with sets of states and/or transitions of the LTS.

Evaluator performs an on-the-fly verification of temporal properties on given Labelled Transition Systems. The result of this verification (TRUE or FALSE) is displayed on the standard output, possibly accompanied by a diagnostic.

The temporal logic used as input language for evaluator is called regular alternation-free μ-calculus. It is an extension of the alternation-free fragment of the modal μ-calculus [106, 70] with action predicates and regular expressions over action sequences. It allows direct encodings of "pure" branching-time logics like CTL [68] or ACTL [60], as well as of regular logics like PDL [116]. Moreover, it has an efficient model checking algorithm, linear in the size of the formula and the size of the LTS model. CADP provides macros to translate action-based CTL formulas (ACTL [60]) and Temporal Logic Patterns [67] to regular alternation-free μ-calculus. A more elaborate version of this logic, able to express temporal properties involving data values, has been defined and studied in [125]; however, the current version of evaluator does not handle the data-based version of the logic.

Several case studies on verification of system using CADP has been published, the most recent are [27, 152, 115, 84].

2.4 Components Related Work

Most component frameworks available today only have tools for checking the static type compatibility of interfaces. Work on behaviour compatibility is quite recent, and not yet available on industrial platforms. We review here the most known research works and tools on verification of components-like systems.

2.4.1 Wright

Wight [13] provides a formal basis for specifying the interactions among architectural components. Wright is built around the basic architectural abstractions of *components*, *connectors* and *configurations*. Figure 2.2 shows a simple client-server system as would be described using Wright specifications.

```
System SimpleExample
        component Server =
                port provide [provide protocol]
                spec [Server specification]
        component Client =
                port request [request protocol]
                spec [Client specification]
        connector C-S-connector =
                role client [client protocol]
                role server [server protocol]
                glue [glue protocol]
Instances
        s: Server
        c: Client
        cs: C-S-connector
Attachments
        s.provide as cs.server
        c.request as cs.client
end SimpleExample
```

Figure 2.2: A simple client-server system description in Wright

A Wright component describes a localised, independent computation. It has two parts, the *interface* and the *component-spec*. An interface consists of a finite number of *ports*. Each port represents an interaction in which the component may participate. It also indicates both the properties that the component must have if it is viewed through the lens of that particular port, and the expectations of the component about the system with which it interacts.

The component-spec describe what the component actually does. It carries out the interactions described by the ports and shows how they are tied together to form a coherent whole.

A connector represents an interaction among a collection of components. It is defined by a set of *roles* and a *glue* specification. Each role specifies the behaviour of a single participant in the interaction. The glue of a connector describe how the participants work together to create an interaction, this is, how the computations of the components are composed to form a larger computation.

Finally a configuration is defined by two parts: *instances* and *attachments*. The instances define the actual entities that will appears in the configuration. The attachments define the topology of the configuration, by showing which components participate in which interactions. The last is done by associating a (instantiated) component's port with a (instantiated) connector's role.

Behaviours in Wright are described using interacting protocols. The protocols of the roles, ports and glues are defined using a variant (sub-set) of CSP [96], including processes and communication events ($e?x$ and $e!x$), prefixing ($e \rightarrow P$), alternative ($P \,\Box\, Q$, *external choice*: the choice is made by the environment), decision ($P \sqcap Q$, *internal choice*: the choice is made by the process itself) and parallel ($P||Q$) operators. The alphabet of a process P, i.e. the set of events it can communicates, is written as αP.

In addition to the standard notation of CSP, Wright introduces three notational conventions:

- A successfully terminating process, noted by "§" ($\S \equiv \surd \rightarrow STOP$, where $\surd$ is the success event)
- Scope of processes, noted by "**let** Q = *process-exp* **in** R". This defines process Q to be *process-expr* within the scope of process R.
- An operator ":" to label as l all the events in a process P (except the event $\surd$), noted by "$l : P$". Additionally Σ is the set of all unlabelled events.

For instance, the connector C-S-connector in Figure 2.2 might be written as:

connector C-S-connector =
  role Client = (request!x → result?y → Client) ⊓ §
  role Server = (invoke?x → return!y → Server) □ §
  glue = (Client.request?x → Server.invoke!x → Server.return?y
    → Client.result!y → **glue**) □ §

Note that the choice operators allow to distinguish between *obligations* (□) to provide some services (such as Server) or *choices* (⊓) to use some services (such as Client).

The semantics of a connector is then given as the parallel interaction of the glue and the roles, where the alphabets of the roles and glue are arrange so that the desired coordination occurs. Wright defines this parallel composition as the *meaning of a connector description*:

Definition 18 *The* **meaning of a connector description** *with roles $R_1, R_2, \ldots, R_n$, and glue Glue is the process:*

$$Glue||(\mathrm{R}_1 : R_1||\mathrm{R}_2 : R_2|| \ldots ||\mathrm{R}_n : R_n)$$

where R_i is the (distinct) name of role R_i, and

$$\alpha Glue = \mathrm{R}_1 : \Sigma \cup \mathrm{R}_2 : \Sigma \cup \ldots \cup \mathrm{R}_n : \Sigma \cup \{\surd\}$$

The behaviour of a port is given as an interacting protocol as well, for instance:

component DataUser =
    port DataRead = get $\rightarrow \sqcap$ §
    other ports...

When associated with the roles, the port protocol takes the place of the role protocol in the resulting system. The semantics of an attached connector is the protocol resulting from this replacement, formally:

Definition 19 *The* **meaning of attaching ports** $P_1 \dots P_n$ *as roles* $R_1 \dots R_n$ *of a connector with glue Glue is the process:*

$$Glue||(\mathrm{R}_1 : P_1||\mathrm{R}_2 : P_2|| \dots ||\mathrm{R}_n : P_n)$$

The main motivation of Wright to do this separation between roles and ports is to enable the reuse of connectors.

Since connectors define interaction between components, the question that naturally arises is whether a given port of a component can be used in a given role of a connector such that the components safely communicate.

This "compatibility" notion between ports and roles is captured in Wright by means of a "refinement relationship". In CSP a refinement is based on the characterisation of a process as the triple (A, F, D) of alphabet, failures and divergences. A process P *is refined by* process Q, written $P \sqsubseteq Q$, if their alphabets are the same, the failures of P are a superset of the failures of Q, and the divergences of P are a superset of the divergences of Q.

Wright defines compatibility between a port and a role based on the behaviour of the port *over the traces described by the role.* They start by defining a *deterministic* version of a role R, by replacing all the non-deterministic choices from R by deterministic choices, formally defined in terms of the traces of R as follow:

Definition 20 deterministic version of R

$$det(R) = (\alpha R, \{(t, s) | t \in traces(R) \wedge \forall e : s \bullet t^\frown\langle e\rangle \notin traces(R)\}, \{\})$$

where $^\frown$ is the catenation CSP operator and $\{declarations|predicate \bullet expression\}$ denotes the set of values defined by *expression* ranging over the values of variables defined in *declarations* that also satisfy *predicate.*

Then compatibility is defined (using "\" as set difference) by:

Definition 21 P **compat** R *("P is compatible with R") if:*

$$R_{+(\alpha P \setminus \alpha Q)} \sqsubseteq (P_{+(\alpha R \setminus \alpha P)}||det(R)_{+(\alpha P \setminus \alpha R)}).$$

where $P_{+B} = (P||STOP_B)$ is the augmented alphabet of process P by the set B ($STOP_B$ is the STOP process over alphabet B).

Informally, a port P is compatible with the role R, both with matched augmented alphabets, if R refines the process resulting form of the parallel composition of the port

P and the deterministic version of the role $det(R)$. Since $det(R)$ is deterministic, any internal choice made by P is still present in the parallel composition, except those that would have resulted in a trace that is prevented by R, and no internal choice is induced as a result of the interaction with R.

Compatibility ensures absence of deadlock if the connector is deadlock free and conservative as demonstrated in [13]. A connector is conservative if the glue traces are a subset of the possible interleaving of role traces.

Given the finite specification of processes (ports, roles and glues), Wright uses FDR [76] to automatically check the compatibility relations and conservatism, and therefore to conclude about deadlock-freedom. They have develop a tool [5] to translate the Wright specifications (written in either an ASCII or Latex representation) into a CSP specification directly used as the FDR input language. The use of model-checking tools [68] to verify temporal properties of the processes and to check relationships between processes is straight-forward in a per-connector basis (global properties can not be verify since global constructor behaviours are not given).

As [13] states, hierarchy in Wright would be easily addressed by defining that a subsystem must be a refinement of the element it represents (in the architecture), once events internal to the subsystem have been hidden. However, this issue has not been deeply studied in Wright.

Finally, Wright inherits the CSP limitation to systems with static process structure. That is, the set of possible processes must be known at system definition time: new processes cannot be created or passed as parameters in a running system. However, we believe that dynamic update of a component is supported by checking that the ports definition of the new component is compatible with the role to be attached. The roles anyway keep being static.

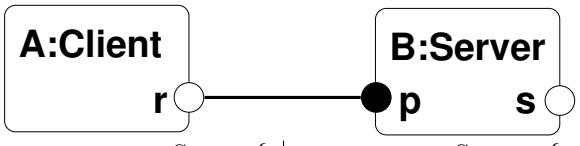
2.4.2 Darwin (Tracta)

Darwin [120] allows distributed programs to be specified as a hierarchical construction of components. Composite component types are constructed from the primitive computational components and these in turn can be configured into more complex composite types.

Components interact by accessing services. Each inter-component interaction is represented by a binding a required service and a provided service. Darwin has both a graphical and textual representation. An example of a Darwin specification is shown in Figure 2.3 in both representations.

Darwin views components in terms of both the services they provide to allow other components to interact with them and the services they require to interact with other components. In Figure 2.3 the convention is that filled-in circles represent services provided by a component and empty circles represent services required by a component.

The purpose of Darwin is to construct composite component types from both instances of basic computational components and other composite components, resulting in a hierarchical structured system. Composite components are formed in Darwin by declaring instances of components and binding the services required by one component to the services provided in another as shown in Figure 2.3.




```
component Server {
  require s;
  provide p;
}
component Client {
  require r;
}
```

```
component System {
inst
  A:Client;
  B:Server;
bind
  A.r -- B.p;
}
```

Figure 2.3: Darwin example

The Darwin specification of a system architecture is used as a framework for structuring behavioural specifications during design and analysis; and it can be used to drive system building during construction. A component does not need to know the global names of external services or where they are to be found in the distributed environment. Components may thus be specified, implemented and tested independently form the system they are part.

Behavioural descriptions are given to Darwin components using the Tracta [86] approach. Tracta uses Labelled Transition Systems (LTSs) as the underlying formalism, with behavioural specifications given in FSP [122] (Finite State Processes), a process algebra. Specifically, behaviour is attached to the software architecture in Darwin by giving a behavioural specification for each primitive component in the hierarchy.

The behaviour of composite components is computed from that of their constituent parts. For this process, all the necessary information related to the structure and interconnections of components is extracted from the architectural description of the system. In terms of LTSs, the behaviour of a composite is the parallel composition of its sub-components LTS. Formally, Tracta defines the parallel composition between processes P and Q, noted $P||Q$, by the following transactional semantics:

$$\frac{P \xrightarrow{a} P'}{P||Q \xrightarrow{a} P'||Q}\ a \notin \alpha Q \qquad \frac{Q \xrightarrow{a} Q'}{P||Q \xrightarrow{a} P||Q'}\ a \notin \alpha P$$

$$\frac{P \xrightarrow{a} P'\ Q \xrightarrow{a} Q'}{P||Q \xrightarrow{a} P'||Q'}\ a \neq \tau$$

where αP denotes the set of action names (alphabet) of process P. This parallel composition is both commutative and associative, therefore the order in which processes are composed is insignificant.

Informally, processes communicate by synchronisation on actions common to their alphabets with interleaving of the remaining actions.

Binding services in Darwin correspond to relabelling with common names the corresponding actions in LTSs. Additionally, Darwin encapsulates (hide) all interactions among its sub-components that do not form part of the interactions with the environment. In order to do that, Tracta defines two operators *relabelling* and *hiding* similar to the same operators in CCS [130].

For each composite, Tracta applies relabelling based on the bindings between the sub-components defined in the Darwin specification. Then a LTS is obtained by applying the parallel composition rules among its sub-components. The resulting LTS is minimised with respect to *weak semantic equivalence* [130], where the internal interactions of the sub-components have been hidden. Finally the minimised LTS can checked against local properties and it is used when computing the parallel composition at the next level of hierarchy.

Tracta distinguishes between *safety* and *liveness* properties.

Safety properties are checked using the approach introduced by Cheung and Kramer in [49]. They are specified by describing expected scenarios as deterministic LTSs without τ transitions. Then a system Sys satisfies a property P if Sys can only generate traces which, when restricted to the alphabet of P, are acceptable to P.

Liveness properties are specified as *Büchi automata*. Since Tracta works with LTS (no specific information is stored on states), it distinguishes accepting states of Büchi automata by means of special transitions that are added to the automata. In particular for a property expressed as a Büchi automaton B, the following conditions should hold in order to be verified:

1. B is deterministic
2. A transition is defined at each state of B for each action in its alphabet
3. The choices in the system analysed are assumed to be fair.

A system satisfies a property expressed as a Büchi automaton if the automaton accepts all infinite executions of the system.

Tracta is supported by the tool "Labelled Transition Systems Analyser" (LTSA) [122]. Darwin specifications are supported by the tool "Software Architect's Assistant (SAA)" [136]. The Tracta team, at the time of this thesis, is working on developing a tool to automatically translate the Darwin description of a system architecture as the input to the LTSA tool.

2.4.3 SOFA

SOFA [145] is a distributed component model and implementation. In the SOFA component model, an application is viewed as a hierarchy of nested software components.

Components in SOFA are instantiated from a *template*, which can be interpreted as a *component type*. A template T is a pair $< F, A >$ where F is a template *frame* and A is a template *architecture*. SOFA components interact with the environment through typed *interfaces*. The frame F defines the set of individual interfaces that any component which is an instance of T will posses. An interface can be instantiated as a *provides-interface* or a *requires-interface*, the first encoding services the component provides and the second services the component requires.

One or more architectures can be associated to a frame. Basically, for a template $T =< F, A >$ its frame F describes the template's specification by providing a *black-box* view on T, and the architecture A describes a particular template's implementation by providing a particular *grey-box* view on T.

An architecture A describes the structure of an implementation of a frame F by:

1. instantiating direct subcomponents of A and
2. specifying the subcomponents interconnections via interface ties

There are four kinds of interfaces ties within a template $T =< F, A >$:

1. *binding* a requires-interface to a provides-interface between two subcomponents,
2. *delegating* a provides-interface of F to a subcomponent's provides-interfaces,
3. *subsuming* a subcomponent's requires-interface to a requires-interface of F, and
4. *exempting* an interface of a subcomponent from any ties (the interface is not employed in A).

An architecture can also be specified as *primitive*, which means the there are no subcomponents and its structure/implementation will be provided in an underlying implementation language.

```
interface IDBServer {
    void Insert(in string key, in string data);
    void Delete(in string key);
    void Query(in string key, out string data);
};

frame DatabaseBody {
    provides:
        IDBServer d;
        ICfgDatabase ds;
    requires:
        IDatabaseAccess da;
        ILogging lg;
        ITransaction tr;
};
```

```
frame Database {
    provides:
        IDBServer dbSrv;
    requires:
        IDatabaseAccess dbAcc;
        ILogging dbLog;
};

architecture Database version v2 {
    inst TransactionManager Transm;
    inst DatabaseBody Local;
    bind Local:tr to Transm:trans;
    exempt Local:ds
    subsume Local:lg to dbLog;
    subsume Local:da to dbAcc;
    delegate dbSrv to Local:d;
};
```

Figure 2.4: Example of a SOFA specification in CDL

The Component Definition Language (CDL) is used to describe interfaces, frames and architectures of SOFA components. It is based on OMG IDL [141]: the language extends the features of IDL to allow specification of software components. An example of SOFA specification in CDL is shown in Figure 2.4, the full CDL syntax is provided in [4]

The behaviour of SOFA components is modelled via event sequences (traces) on the component's interfaces (connections). The event sequences are approximated and represented by regular expressions called *behaviour protocols*.

Every event is syntactically written as

$$< \text{event prefix} >< \text{connection\_name} > . < \text{local\_event\_name} >< \text{event suffix} >$$

The pair <connection_name>.<local_event_name> encodes the event "local_event_name" in the interface "connection_name". The event prefix ($!, ?$ or τ)

expresses whether an event is emitted (requirement), absorbed (provides) or is an internal event. The event suffix expresses whether an event is a *request* (↑) or a *response* (↓) to an event request. The abbreviation "$?m\{\alpha\}$" stands for "$?m\uparrow;\alpha;!m\downarrow$", "$?m$" stands for "$?m\uparrow;!m\downarrow$" and "$!m$" stands for "$!m\uparrow;?m\downarrow$".

Definition 22 ***A* behaviour protocol** *(protocol for short) P over an alphabet S, is a regular-like expression which (syntactically) generates a set of traces over S, the language $L(P)$.*

A behaviour protocol may contain the basic operators of regular expressions [100] as well as the enhanced and composed operators listed bellow (A, B denotes a protocol):

- *Basic operators*

  sequencing $A;B$ the set of traced formed by concatenation of a trace generated by A and a trace generated by B,

  alternative $A+B$ the set of traces which are generated either by A or by B,

  repetition $A*$ zero or more concatenation of a trace generated by A.

- *Enhanced operators*

  and-parallel $A|B$ an arbitrary interleaving of event tokens from traces generated by A and B,

  or-parallel $A||B$ stands for $A+B+(A|B)$,

  restriction A/G the events NOT in the set G are omitted from the traces of $L(A)$ [1]

- *Composed operators*

  composition $A\ \Pi_X\ B$ the set of traces each formed as an arbitrary interleaving of event tokens from a pair of traces (α,β), (where $\alpha \in L(A)$ and $\beta \in L(B)$), such that, for every event $x \in X$, if x is prefixed by ? in α and by ! in β (or vice versa), any occurrence of $?x, !x$ resp. $!x, ?x$ as the result of the interleaving is merged into τx in the resulting trace (the pair of events becomes an internal event),

  adjustment $A|T|B$ the set of traces each formed as an arbitrary interleaving of event tokens from a pair of traces (α,β), where $\alpha \in L(A)$ and $\beta \in L(B)$, with the exception of event tokens from T which have to appears in α and β in the same order (representing "synchronisation points"). If the interleaving produces $\ldots x, x \ldots$ for an $x \in T$, then x, x is merged into $\ldots x \ldots$ in the resulting trace (the pair becomes a single event).

In Principe, the semantics of the adjustment operator $|T|$ is inspired by the generalised parallel operator defined in CSP [96], while the semantics of the composition operator Π_X by the parallel composition in CCS [130]. The formal definition of both operators is given in [146].

[1] please do not confuse the restriction operator of SOFA with the restriction operator $E\backslash L$ of CCS, where the actions in the set L are forbidden (omitted) from E

The composition operator is suitable for expressing joint behaviour of components communicating via bound interfaces. The adjustment operator is suitable for the comparison of the behaviour of components in the following way: If the protocol B in $A|T|B$ comprises only events tokens from T, it can be seen as an obligation for the protocol A in the sense that A should handle the event tokens from T in the same way that B does. Since a trace $\beta \in L(B)$ comprises event tokens from T only, either $\alpha|T|\beta$ generates α (if $\alpha \in L(A)$ contains all the tokens of β in the same order that appears in β), or it does not yield any trace. As consequence, $L(A|T|B) \subseteq L(A)$.

Behaviour protocols are integrated in the CDL and associated to the SOFA natural abstraction units (interface, frame and architecture) as indicated in Table 2.2.

CDL concept	**behaviour protocol**	**alphabet S** (the event tokens on:)		
		S_{ext}		**S_{int}**
		S_{prov}	**S_{req}**	
frame F	frame protocol	F's provides interfaces	F's requires interfaces	$\emptyset$
architecture A (of frame F)	architecture protocol	F's provides interfaces	F's requires interfaces	interfaces bound in A
provides interface I_p	interface protocol	I_p	$\emptyset$	$\emptyset$
requires interface I_r	interface protocol	$\emptyset$	I_r	$\emptyset$

Table 2.2: Behaviour protocols and CDL association

Frame protocol is a behaviour specifying the acceptable interplay of method invocations on the provides-interfaces and reactions on the requires-interfaces of the frame. The frame protocol is given by the system's designer in the CDL as the example shown in Figure 2.5.

```
frame Database {
    provides:
        IDBServer dbSrv;
    requires:
        IDatabaseAccess dbAcc;
        ILogging dbLog;
    protocol:
        !dbAcc.Open ;
            ( ?dbSrv.Insert { ( !dbAcc.Insert ;
                !dbLog.LogEvent )* }
            +
            ?dbSrv.Delete { ( !dbAcc.Delete ;
                !dbLog.LogEvent )* }
            +
            ?dbSrv.Query { !dbAcc.Query* }
        )*;
        !dbAcc.Close
};
```

Figure 2.5: Behaviour protocol of frame DatabaseBody

For a template $T =< F, A >$, *architecture protocol* is a behaviour protocol describing the interplay on the method invocations on the interfaces of F and the outermost

interfaces of the subcomponents in A. The architecture protocol is generated automatically combining the frame protocols of the subcomponents via the composition operator Π_X. In a architecture protocol, the set X of Π_X is composed of all the events on the interfaces appearing in the bind clauses of the corresponding architecture and can be inferred from the architecture specification.

Interface protocol is a behaviour specifying the acceptable order of method invocations on an interface. It is intended to simplify a component design as it represents the behaviour of a component on a single interface only. The interface protocol is given in the CDL specification.

SOFA propose a top-down design. It starts by substituting the primitive top component by another refined composite, whose architecture behaviour is defined by combining the frame protocols of its subcomponents. This refinement is recursively repeated through the hierarchy until the inner-most primitive component whose architecture behaviour protocol is determined by its implementation. In this refinement process, at each level the hierarchy defined by a template $T =< F, A >$, arises the question whether the architecture protocol A fits in the frame protocol F, i.e. whether the architecture A implements the frame specification F.

In SOFA, this relation is called *protocol conformance* [146, 11]. An architecture protocol P_A refines a frame protocol P_F if P_A conforms P_F. The conformance relation is given bellow:

Definition 23 Harmonious alphabets *two alphabets S_A and S_B are harmonious if for all event e such that $e \in S_A$ and $e \in S_B$, e is a provides (resp. requirement, resp. internal) in both alphabets S_A and S_B.*

Definition 24 Protocol compliance *Let L_A and L_B be protocols with harmonious alphabets S_A and S_B respectively. Let S be another protocol harmonious with S_A and S_B such that:*

$$S_{prov} \subseteq S_{A,prov} \cup S_{B,prov} \wedge S_{req} \subseteq S_{A,req} \cup S_{B,req} \wedge S_{int} \subseteq S_{A,int} \cup S_{B,int}$$

The behaviour L_A is compliant with the behaviour L_B on S if:

$$L_B/S_{prov} \subseteq L_A/S_{prov} \wedge (L_B/S_{prov})|S_{prov}|(L_A/S) \subseteq L_B/S$$

Behaviour protocols associated with a CDL incorporate connections at different levels of abstraction and name unification may be required for reasoning about the protocols. SOFA introduce the notion of *qualification of a protocol P_X* with respect to a CDL abstraction Y (denoted as ${}^{Y}P_X$), which means that any name of an interface instance/connection in P_X associated with the CDL abstraction X is modified (unified) to that used in Y for the same interface instance/connection.

Definition 25 Protocol Conformance *Let $T =< A, F >$ be a template with the frame protocol P_F and the architecture protocol P_A. The protocol P_A conforms to the protocol P_F if P_A is compliant with ${}^{A}P_F$ on S, where S is the alphabet associated with F.*

Informally, the definition expresses that the architecture protocol of A cannot generate traces not allowed by the frame protocol of F (under the assumption that the

provides-interfaces known in F are used in A in a way allowed by the frame protocols). At the same time, the architecture protocol can be "less demanding" on the requires-interfaces.

The CDL compiler provided in the SOFA implementation [4], automatically generates the architecture protocols and tests the interface, frame, and architecture protocol conformance.

Errors detection

In [12] SOFA extends the behaviour protocols to capture faulty computations caused by a "bad" component composition. SOFA splits faulty computations in two categories: 1) At some point the computation cannot continue - *no continuation* error which includes two specific error types: *bad activity* and *no activity*; 2) A computation is infinite (*divergency* error).

To capture errors, SOFA introduces *error tokens*: $\varepsilon n\uparrow$, $\varepsilon n\downarrow$, $\varepsilon\oslash$ and $\varepsilon\propto$ (where n is an event name); and *erroneous traces* of the form $w < e >$, where w is a trace formed of non-error event tokens and e stands for the error token reflecting the type of the error occurred: $\varepsilon n\uparrow$, $\varepsilon n\downarrow$ for bad activity, $\varepsilon\oslash$ for no activity and $\varepsilon\propto$ for divergency.

1. *bad activity* is produced when a component tries to emit $!n\uparrow$ or $!n\downarrow$ (where n is an event name), but the component at the other side of the respective binding is not ready to accept (to issue $?n?\uparrow$ resp. $?n\downarrow$), i.e. no suitable trace is defined in B's behaviour.

2. *non activity* is produced when neither A or B is able to emit an event, and at least one of A and B cannot stop.

3. *divergence* is produced when A and B can emit events, but after each event, at least one of the components cannot stop.

The set of erroneous traces induced when composing two components A and B, noted $ER(L_A, L_B, X)$, is formally defined in [12]. SOFA defines a *consent* operator ∇_X for languages L_A, L_B, where X consists of all the events from connections between components A and B. Formally:

$$L_A \nabla_X L_B = (L_A \Pi_X L_B) \cup ER(L_A, L_B, X)$$

Dynamic update

An important feature of SOFA is to allow the update of a component during runtime, i.e it can dynamically change the implementation of a component by a new one. The architecture protocol of the new component should *conforms* to the frame protocol. Besides this conformance relation, SOFA states in [12] that during the update of a component C, it has to be ensured: 1) *component passivity*: no thread is executing a method of an interface of C; 2) *update atomicity*: no other component (external to C) in the system calls a method of an interface of C.

To ensure 1) and 2), SOFA allows the system's designer to use a special update token π, defining when an update can take place. An update of a component C is possible

after a given trace prefix tp, if $tp <?\pi\uparrow>$ is also a prefix of a trace generated by the frame protocol of C.

Component passivity is ensured by the SOFA implementation at runtime (still, this can take place only if the update is possible). Atomic update is ensured by static analysis of the behaviour protocols.

Because atomically update is a matter of communication among the components on a particular level of nesting (the components from an architecture Z), it is not a property of a single component, but a property of Z. Let Z consist of frames $F_1, \ldots, F_n$ with frame protocols $P_{F_1}, \ldots, P_{F_n}$. The atomicity is verified in two steps:

1. *Local atomicity* is tested for every frame protocol. A protocol is locally atomic if in all the traces generated by the protocol between the pair of corresponding update tokens ($?\pi\uparrow$ and $!\pi\downarrow$), no other tokens occurs.

2. If (1) succeeds, the protocols $P_{F_1}, \ldots, P_{F_n}$ are composed using the consent operator which yields the language of the architecture protocol P_Z

If (1) succeeds, there cannot be an accepting token of the form $?e$ between a pair of update tokens. Thus any attempt to call a method on an interface of a frame during an update results in a bad activity captured by a trace of the form $w <?\pi\uparrow; \varepsilon n\uparrow>$ in (2).

Summarising, SOFA provides a concrete formal framework of component behaviours. The conformance operator, the detections of errors (via the consent operator) and the explicit introduction of update tokens, provide a formal means for capturing component (dynamic) substitutability as well as adherence of a component implementation to the behaviour specification.

Chapter 3

Behaviour Specifications

3.1 Introduction

In behavioural verification of systems, the first natural question that arises is which would be the best suitable language/format to model its behaviour.

Our target systems are real concurrent and distributed asynchronous processes built within a component framework (we go deeper on the analysis of such systems in chapters 4 and 5). Then we want a well suited language for describing the behaviour of such systems.

We want as well this language to be based on process algebra theories [130, 96, 33] that would enable us to profit from its main features: *operational semantics* to describe unambiguously the system behaviour and check their properties, *equivalences and pre-orders* to provide a behavioural relation between different systems and *compositional modelling* to model larger systems up from their smaller pieces that compose them. The latter is specially important when looking for scalability.

Moreover, we want this language to be both intuitive and expressive enough for specifying the behaviour of our target system, as well as the target language for static analysis tools (then implementations can be checked against formal specifications). In addition it should be formal enough as the input language for state of the art automatic model-checking tools.

Several languages featuring process algebras have been proposed, besides the seminal work of Milner [130] and Hoare [96] we can mention TCSP [42], ACP [32], Meije [61], μCRL [88] or LOTOS [104] among many others (some of them reviewed in section 2.2).

A good candidate as a behavioural language would be a process algebra with at least value-passing features, or even encoding dynamic process and channel creation and reconfiguration.

Promela [98, 97] fits well on these requirements except that it does not support nested (hierarchical) process. This restriction does not allow us to directly describe our *hierarchical* systems using Promela.

Another excellent candidate is LOTOS [104], however LOTOS is just too expressive to be subject to automatic decision procedures (taken by our automatic tools), and would not give us models and algorithms usable in practical tools. The data part specification, based on abstract data types (ADT) [69], is complex and error-prone.

It requires the intervention of a qualified user, drastically reducing its usability and automatic reasoning.

Our approach is to extend the general notion of labelled transition systems (LTS) and hierarchical networks of communicating systems (synchronisation networks) by adding parameters to the communication events. Rather than a "language", we prefer speak of an "intermediate format", for avoiding confusions due to the wide use of the term in the computer field.

The labelled transition systems of Arnold & Nivat can be naturally associated with the CCS process algebras as introduced by Milner [130]. In CCS a process p is something that can execute some action a, or react to an event a, and transform itself to a new process p', which is indeed a labelled transition $p \xrightarrow{a} p'$.

The synchronisation product introduced by Nivat is both simple and powerful, because it directly addresses the core of the problem. One of the main advantage of using its high abstraction level is that almost all parallel operators (or interaction mechanism) encountered so far in the process algebra literature become particular cases of a very general concept: synchronisation vectors.

We will take benefits of both approaches: the simplicity and universality of synchronised LTSs, and the conciseness of symbolic graphs.

We start by slightly simplifying the Arnold & Nivat's definition of labelled transition systems. Then we introduce the synchronisation constraint as part of a *synchronisation network*. Contrary to synchronisation constraints, the network allows dynamic changes between different sets of synchronisation vectors through a *transducer* LTS. We give a synchronisation product definition semantically equivalent to the one given by Arnold & Nivat.

At a next step, we use the Hennessy and Lin's approach for adding parameters in the communications events of both transition systems and synchronisation networks. These communication events can be guarded with conditions on their parameters. Our agents can also be parameterized to encode sets of equivalent agents running in parallel.

We shall see later that the format we obtained here is too powerful to be used directly in existing verification tools. We shall restrict the domain of parameters to be simple enumerable types: booleans, integers, intervals, finite enumerations or structured objects; and define instantiations of the system based on finite abstraction of such parameters in a spirit similar to value-passing CCS [130].

Following we give the formal definition of our intermediate language that we call *Parameterized Networks of Communicating Automata*. In section 3.3 we describe a case study we use to validate our intermediate format as a specification language. Finally, section 3.4 concludes with the main results of our approach.

3.2 Parameterized Networks of Communicating Automata

We aim at combining the value-passing and the synchronisation product approaches. We define an intermediate format featuring parameterized processes, value-passing communication, behaviours expressed as symbolic labelled transition systems, and data-values of simple countable types.

3.2.1 Theoretical Model

We start with an unspecified set of communications actions *Act*, that will be refined later. We give as well a definition of labelled transition systems which is simpler, but semantically equivalent to Arnold & Nivat's (section 2.1.3, Definition 10)

Definition 26 LTS. *A labelled transition system is a tuple* $(S, s_0, L, \rightarrow)$ *where* S *is the set of states,* $s_0 \in S$ *is the initial state,* $L \subseteq Act$ *is the set of labels,* $\rightarrow$ *is the set of transitions :* $\rightarrow \ \subseteq S \times L \times S$. *We write* $s \xrightarrow{\alpha} s'$ *for* $(s, \alpha, s') \in \ \rightarrow$.

Then we define **Nets** in a form inspired by [15], that are used to synchronise a finite number of processes. A Net is a form of generalised parallel operator, and each of its arguments are typed by a **Sort** that is the set of its possible observable actions.

Definition 27 Sort. *A Sort is a set* $I \subseteq Act$ *of actions.*

A LTS $(S, s_0, L, \rightarrow)$ can be used as an argument in a Net only if it agrees with the corresponding Sort ($L \subseteq I_i$). In this respect, a Sort characterises a family of LTSs which satisfy this inclusion condition.

Our definition of Nets has the same expressive power than the synchronisation constraints of section 2.1.3 (Definition 12), though you can consider it has more "syntax oriented".

Nets describe dynamic configurations of processes, in which the possible synchronisations change with the state of the Net. They are Transducers, in a sense similar to the Lotomaton expressions [107, 135]. Each state of the transducer corresponds to a given configuration of the network in which a given set of synchronisations is possible; some of those synchronised actions can trigger a change of the transducer's state. Transducers are encoded as LTSs which labels are synchronisation vectors, each describing one particular synchronisation of the process actions:

Definition 28 Net. *A Net is a tuple* $< A_G, I, T >$ *where* A_G *is a set of global actions,* I *is a finite set of Sorts* $I = \{I_i\}_{i=1,\ldots,n}$, *and* T *(the transducer) is a LTS* $(T_T, s_{0_t}, L_T, \rightarrow_T)$, *such that* $\forall \overrightarrow{v} \in L_T, \overrightarrow{v} =< l_t, \alpha_1, \ldots, \alpha_n >$ *where* $l_t \in A_G$ *and* $\forall i \in [1, n], \alpha_i \in I_i \cup \{idle\}$.

We say that a Net is *static* when its transducer contains only one state. Note that a synchronisation vector can define a synchronisation between one, two or more actions from different arguments of the Net. When the synchronisation vector involves only one argument, its action can occur freely.

The semantics of the Net is given by the synchronisation product:

Definition 29 Synchronisation Product. *Given a set of LTS* $\{LTS_i = (S_i, \ s_{0_i}, L_i, \rightarrow_i)\}_{i=1\ldots n}$ *and a Net* $< A_G, \{I_i\}_{i=1\ldots n}, (S_T, s_{0_T}, L_T, \rightarrow_T) >$, *such that* $\forall i \in [1, n], L_i \subseteq I_i$, *we construct the product LTS* $(S, \ s_0, \ L, \ \rightarrow)$ *where* $S = S_T \times \prod_{i=1}^{n}(S_i)$, $s_0 = s_{0_T} \times \prod_{i=1}^{n}(s_{0_i})$, $L = A_G$, *and the transition relation is defined as:*

$\rightarrow \triangleq \{s \xrightarrow{l_t} s' |\ s =< s_t, s_1, \ldots, s_n >, s' =< s'_t, s'_1, \ldots, s'_n >,$
$\exists\ s_t \xrightarrow{\overrightarrow{v}} s'_t \in \rightarrow_T, \overrightarrow{v} =< l_t, \alpha_1, \ldots, \alpha_n >, \ \forall i \in [1, n], \ (\alpha_i \neq idle \wedge s_i \xrightarrow{\alpha_i} s'_i \in \rightarrow_i) \vee (\alpha_i = idle \wedge s_i = s'_i)$

Note that the result of the product is a LTS, which in turn can be synchronised with other LTSs in a Net. This property enables us to have different levels of synchronisations, i.e. a hierarchical definition for a system.

Next we enrich the format with parameters at the level of LTS and of Nets.

Definition 30 Parameterized Actions *are actions having parameters in the form* $\alpha(\overrightarrow{x})$*, where* α *is an action and* $\overrightarrow{x}$ *a vector of parameter terms; or the non-observable action* τ*.*

A parameterized LTS is a LTS with parameterized actions, with a set of parameters (defining a family of similar LTSs) and variables attached to each state. Additionally, the transitions can be guarded and have a resulting expression which assigns the variables associated to the target state:

Definition 31 pLTS. *A parameterized labelled transition system is a tuple* $pLTS = (K, S, s_0, L, \rightarrow)$ *where:*

$K = \{k_i\}$ *is a finite set of parameters,*

S *is the set of states, and each state* $s \in S$ *is associated with a finite vector of variables* $\overrightarrow{v_s}$*,*

$s_0 \in S$ *is the initial state,*

$\rightarrow \subseteq S \times L \times S$ *is the set of transitions and*

$L = (b, \alpha(\overrightarrow{x}), \overrightarrow{e})$ *is the set of labels (parameterized actions), where* b *is a boolean expression,* $\alpha(\overrightarrow{x})$ *is a parameterized action, and* $\overrightarrow{e}$ *is a finite set of expressions.*

In the label $l = (b, \alpha(\overrightarrow{x}), \overrightarrow{e})$ of a transition $s \xrightarrow{l} s'$, the free variables of b, $\overrightarrow{x}$ and $\overrightarrow{e}$ are a sub set of the free variables of $\overrightarrow{v_s}$ (source state) union the global variables; and the dimension of the vector $\overrightarrow{e}$ should be the same than $\overrightarrow{v_{s'}}$ (target state). The latter because, as we will see in section 3.2.3, $\overrightarrow{e}$ infers to the variables of the target state ($\overrightarrow{v_{s'}} = \overrightarrow{e}$).

Definition 32 Parameterized Sort. *A Parameterized Sort is a set* pI *of parameterized actions.*

Definition 33 *A* **pNet** *is a tuple* $< pA_G, H, T >$ *where :* pA_G *is the set of global parameterized actions,* $H = \{pI_i, K_i\}_{i=1..n}$ *is a finite set of holes (arguments). The transducer* T *is a pLTS* $(K_G, S_T, s_{0_T}, L_T, \rightarrow_T)$*, such that* $\forall \overrightarrow{v} \in L_T$, $\overrightarrow{v} =< l_t, \alpha_1^{k_1}, \ldots, \alpha_n^{k_n} >$ *where* $l_t \in pA_G$ *,* $\alpha_i \in pI_i \cup \{idle\}$ *and* $k_i \in K_i$*.*

The K_G of the transducer is the set of global parameters of the pNet. Each hole in the pNet has a sort constraint pI_i and a parameter set K_i, expressing that this "parameterized hole" corresponds to as many actual arguments as necessary in a given instantiation. In a synchronisation vector $\overrightarrow{v} =< l_t, \alpha_1^{k_1}, \ldots, \alpha_n^{k_n} >$, each $\alpha_i^{k_i}$ corresponds to the α_i action of the k_i-nth corresponding argument LTS.

We do not define a product of pLTS that would give some kind of "late" or "symbolic" semantics of our generalised pNets. Instead, we define instantiations of the parameterized LTS.

Before this, we give a definition of a subset of our intermediate format, which corresponds to a graphical language for static pNets. This language will be used for all graphical examples in this thesis.

3.2.2 Graphical Language

We provide a graphical syntax for representing *static* Parameterized Networks, that is a compromise between expressiveness and user-friendliness. This graphical syntax principally aims to better explain our approach by visualising the behaviours described using our formalism. Although the graphical syntax can be used as a behavioural design language, it does not claim to be better than existing approaches which can be more adequate depending on the aims (such as textual LOTOS or graphical UML).

We use a graphical syntax similar to the Autograph editor [10], augmented by elements for parameters and variables: a *pLTS* is drawn as a set of circles and states respectively representing states and transitions. The states are labelled with the set of variables associated with it ($\overrightarrow{v_s}$) and the edges are labelled by $[b]\ \alpha(\overrightarrow{x}) \rightarrow \overrightarrow{e}$ (see Definition 31).

An *static pNet* is represented by a set of named boxes, each one encoding a particular Sort of the pNet, and an enclosing box. These boxes can be filled with a pLTS satisfying the Sort inclusion condition. Each box has a finite number of *ports* on its edge, represented as labelled bullets, each one encoding a particular parameterized action of the Sort.

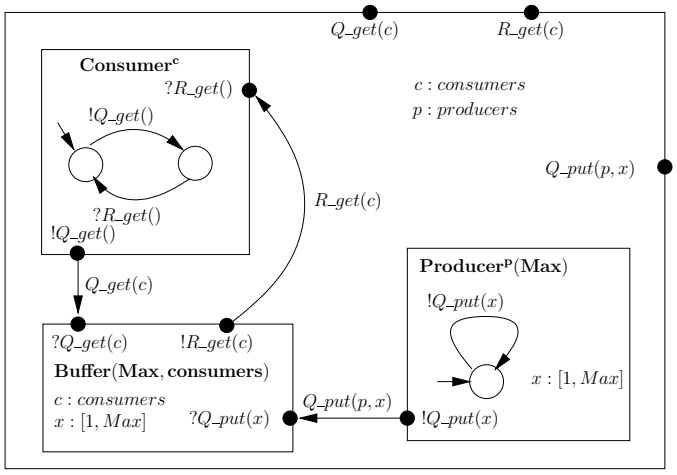


Figure 3.1: Parameterized consumer-producer system

Figure 3.1 shows an example of such a parameterized system. It is composed of a single bounded buffer (with a capacity of Max) and a bounded quantity of consumers ($\#consumer$) and producers ($\#producer$). Each producer feeds the buffer with a quantity (x) of elements at once. Each consumer requests a single element from the buffer ($!Q\_get()$) and waits for the response ($?R\_get()$).

Figure 3.1 also introduces the notation to encode sets of processes; for example, **Consumer**$^{\mathbf{c}}$ encodes a set composed of a **Consumer** process for each value in the domain of c. Therefore, each element in the domain of c is related (identifies) to an individual **Consumer** of the set.

The edges between ports in Figure 3.1 are called links. Links in a Net express synchronisation between internal boxes or with external processes. They also can be between ports of different instantiations of the same box. A synchronisation may be required between more than two actions (i.e. involving more than two ports), this is

represented by an ellipse with multiple arriving/outgoing links from/to the ports of the boxes (processes) whose actions must be done simultaneously (an example of such is shown later in Figure 3.5). Each link encodes a transition in the Transducer LTS of the *pNet*.

When the initial state is parameterized with an expression, it can be indicated which evaluation of the expression (for which value of the variables) is to be considered as the initial state.

The various elements of the graphical language described here are naturally translated into pLTSs and pNets. A *drawing* in our language may contain an arbitrary composition of pNets and pLTSs. The ports in the outer box in Figure 3.1 are actions we choose to be visible. The **Buffer** box in the figure represents a hole in the *pNet* to be filled, we left it unspecified on this section for clarity but we go deeper in the *pLTS* describing its behaviour in section 6.5.2.

Our graphical approach is valid only for *static pNets*: their transducers have only one state. If we had to represent dynamic pNets, we would have to add the transducer LTS in the drawing of the Net.

3.2.3 Instantiation

In this work's framework, we do not give a more precise definition of the language of parameterized actions, and we shall not try to give a direct definition of the synchronisation product of pNets/pLTSs. Instead, we shall instantiate separately a pNet and its argument pLTSs (abstracting the domains of their parameters and variables to finite domains, before instantiating with all possible values of those abstract domains), then use the non-parameterized synchronisation product (Definition 29). This is known as the early approach to value-passing systems [130, 133].

The abstraction of the parameter domains should be *sound* in the sense that every universal property that can be proved using the abstraction will also hold true for the original system.

Several work has been done in the abstraction domain derived from the *abstract interpretation* theories introduced by Cousot [57, 56, 58], but most of them, for instance in [51, 89, 112], are based on state-transformer languages and *Kripke structures* [68]. We rather prefer the approach proposed by Riely and Cleaveland [53, 148] based on abstraction of value-passing transition systems.

Riely and Cleveland take the notions and notations from *abstract interpretation* and introduce an *abstraction function* α between two value sets applicable to process terms and LTSs. They also devise a corresponding *concretization function* γ with the property that α and γ form a Galois insertion on the preordered domains. They define an abstract semantics to be *sound* if the set of values that it assigns to an expression includes every value that the concrete semantics might assign to that expression. Then they prove that sound value interpretations induce sound process semantics, and sound process semantics, in their setting, implies preservation of a large family of both safety and liveness temporal properties (assuming reasonable hypothesis on the values of variables occurring in the formulas). This theorem licenses the use of sound abstract value interpretations for verifying system properties, using both preorder checking and model

checking.

For instance, we can abstract the system described in Figure 3.1 to have 2 consumers and 2 producers. In both cases, one encodes every consumer/producer as an individual entity, and the second encodes the remaining consumers/producers. We also set buffer capacity to 3. An instantiation using these parameter domains is shown in Figure 3.2.

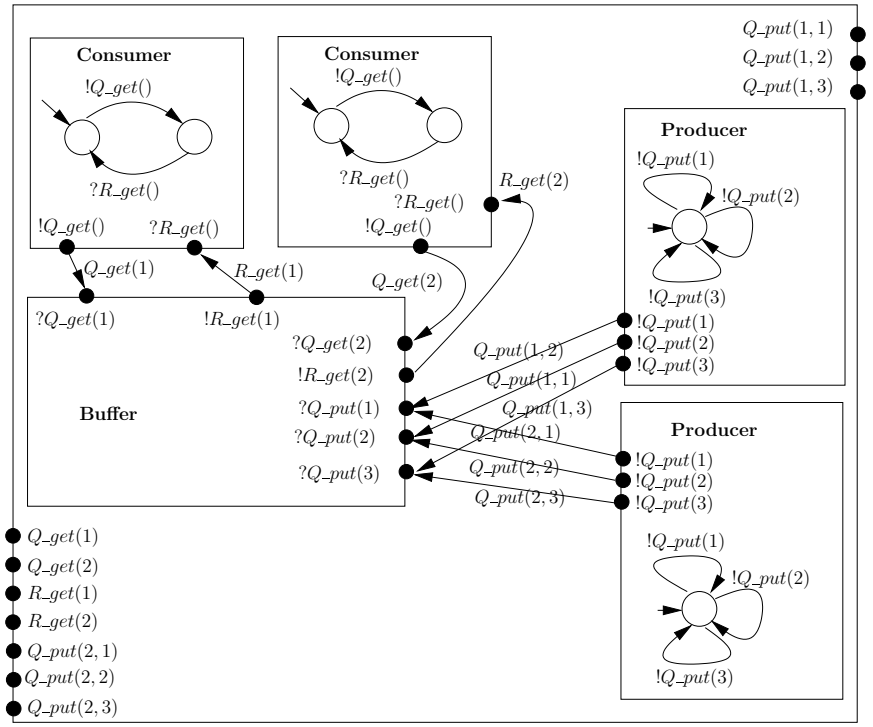


Figure 3.2: Instantiated consumer-producer system

We have developed a tool, described in section 6.5 for automatically generating finite systems from parameterized ones given their finite abstract domains.

3.3 Case study: The Chilean Electronic Invoices System

As an early work before tackling component behaviours, we have used our intermediate format to build a formal specification of the electronic invoices system used in Chile [17]. We used this case study to illustrate and validate our approach for the specification and verification of distributed applications. In this respect, this example has a number of interesting features: first it is a real-world application, whose (informal) requirements have been published on the Chilean tax agency web server [65]. It is a large example, that fits well with the idea of parameterized models: it has a complex component structure (17 automata in 4 levels of hierarchy), it will run with thousands of instances of the components and the smallest instantiation, with two actors of each type, already has a pretty large state space (over 10^{12}). Last, even if not a standard "critical system", it is naturally the kind of distributed applications for which both security and safety have a very strong economical impact.

3.3.1 System description

In this section, we informally describe the electronic invoice system recently realised in Chile, as published in September 2002; for a detailed explanation, please look at [65].

The Chilean law requires any commercial transaction done in Chile to be supported by a legal document previously authorised by the tax agency (Servicio de Impuestos Internos, from now on **SII**). There are several types of documents depending on the transaction such as the invoice for sales, or the forms for the transportation of goods. For a specific taxpayer and document type, each emitted document is assigned a unique number named *id*. Before emitting a document, it must be authorised by SII: this is done through an authorisation stamp specific to: (1) a set of documents, (2) a document type and (3) a taxpayer. The taxpayer obtains authorisation stamps via the SII Web site. We call the emitter of an invoice a "vendor" and its receptor a "buyer", even if those may be simply two different *roles* of the same taxpayer.

Every generated document must be sent to SII before sending it to the buyer and before the transport of goods (if relevant). All documents must include a digital seal, generated from the document data and the authorisation stamp.

SII has created a Web site where the buyer can verify if an invoice has been authorised and verify whether the emitter has sent the same invoice to SII than the buyer has received.

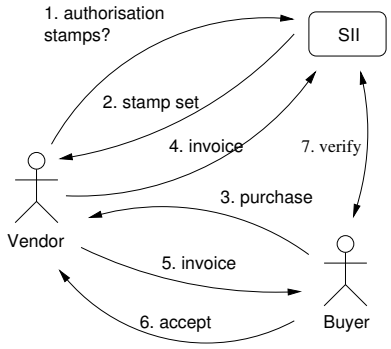


Figure 3.3: Normal Scenario

The most common scenario is shown in Figure 3.3. In step 1 the vendor asks for authorisation stamps. SII responds with a stamp set (step 2). Once a buyer has made a purchase (step 3), the vendor generates an invoice, sends it first to SII (step 4), then sends it to the buyer (step 5). In this scenario, the buyer will accept the invoice (step 6) and later it will verify the validity of the invoice with SII (step 7).

An electronic invoice is *well emitted* if it respects the format specifications defined by SII; if this is not the case, SII will refuse it and the invoice will be considered as never emitted. On the buyer's side, if the transaction has never been realised or if there are errors in the invoice information, the buyer may refuse the invoice and consider it as never received. Then it is the duty of the emitter to send a cancellation of the invoice to SII.

Note that Figure 3.3 is just a drawing meant to explain the application, not a part of the specification. In fact the available specification is informal, and consists in a natural language (Spanish) description of the protocols and of document formats. Our

first task then was to identify within this informal specification the parts relative to the communication between the various subsystems, and to extract a list of semi-formal requirements.

3.3.2 System properties

Some of the behavioural properties that the system should respect are listed below. The published requirements [65] focus more on the format and contents of the electronic invoices than on the system's behaviour. Properties 1, 3, 4, and 6 appear explicitly in the requirements. Properties 2, 5 and 7 do not, but they appeared as natural and useful extensions of the specification.

1. A taxpayer cannot emit invoices if it has not received stamps from SII. More specifically, a taxpayer can emit as many invoices as the quantity of stamps received from SII.

2. SII gives the right answers to the invoice status request: not present when it has not been sent to SII, present when it has been sent, and cancelled when it has been cancelled by the vendor.

3. Every invoice refused by a buyer must be cancelled by the vendor.

4. An invoice id can be used only once.

5. It is not possible to cancel an invoice which has not been emitted before.

6. Every invoice sent to a buyer, should be sent to SII first.

7. Every emitted invoice finishes being accepted by the buyer or cancelled in SII.

3.3.3 Formalisation

The intention here is not to describe all aspects of the system specification. We rather concentrate on the behaviour of the system, the communications between the distributed processes and their temporal properties.

- We assume that the communication channels are reliable.

- Security aspects (authentication, integrity) and document format verification are supposed to be treated elsewhere. All the processes in the system are trusted.

- There are only two types of documents, invoices and cancellations and only two types of authorisation stamps, one for invoices and another for cancellations. The only specific value to be considered for a document is its identification number (id).

In this section we concentrate in only two *pNets* describing the system, the interested reader will find the complete system formalisation in [26].

The Vendor system

Figure 3.4 shows the network that defines the behaviour of the Vendor. We use a richer language in the labels than those introduced in section 3.2.2 such as prefixing the communication actions with the process to whom/from where is done (Ex: $!Stock.stamp()$), having the identifier of processes between square parenthesis (Ex: $!AI[id].cancelSii()$) or setting the domain of variables (Ex: $x : int$ or $b : buyerSet$). This is for a better understanding but does not involve any extra semantics when instantiating.

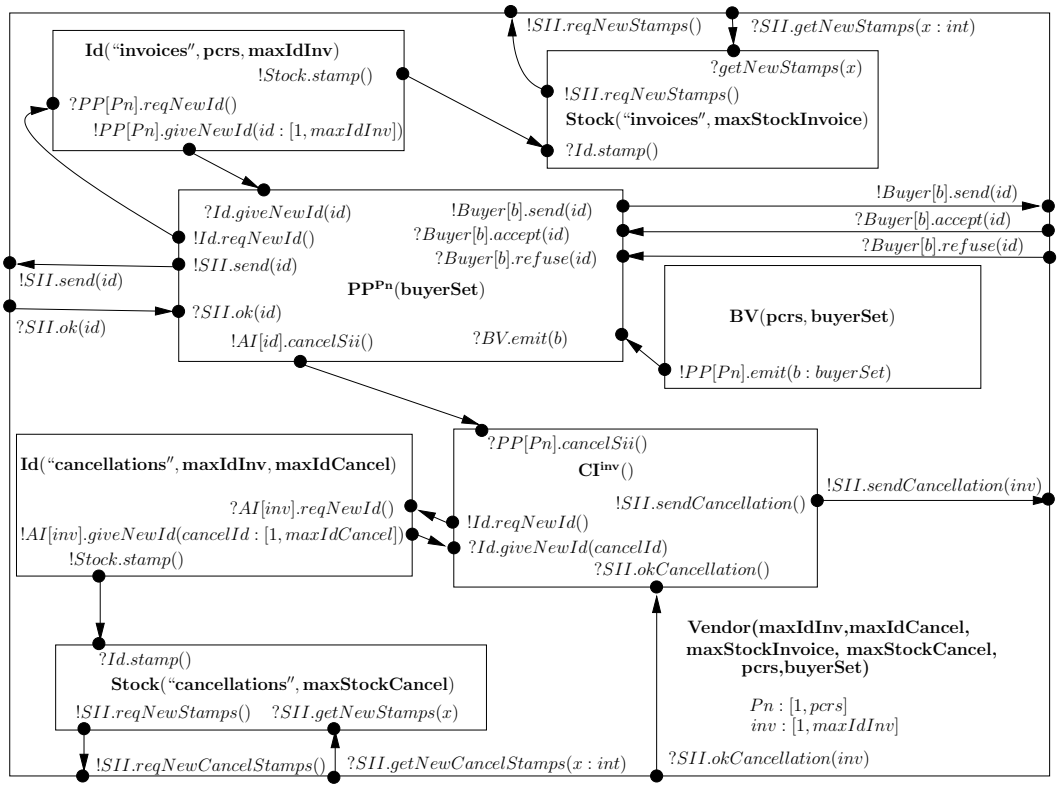


Figure 3.4: The Vendor system

The vendor has two pairs of **Stock** and **Id** processes: one pair for invoices and the other for cancellations. The **Stock** process manipulates a stock of stamps. It provides stamps for the generation of documents and requests new stamps to SII. The **Id** process assigns a unique sequential number to each new document (once a stamp has been provided by the **Stock** process). There is one single **BV** process that initiates new purchases. The purchase process (**PP**) takes care of the main life's cycle of a purchase. It is parameterized with the variable *pcrs*, which encodes the number of purchases that can be treated simultaneously (section 3.2.2 explains the notation P^n for processes). There is a cancellation process (**CI**) for each invoice *id* (which can possibly be cancelled). The **PP** process sends requests to the **Id** invoices process for new invoices *id*s while the **CI** process does so with the **Id** cancellations process. Note that even when the **Id** and **Stock** processes seem to be the same for invoices and cancellations, they could be instantiated with different domain of variables, resulting in different finite non-parameterized processes.

The reception and verification process at SII

The network **Reception** in Figure 3.5 specifies the part of the SII process in charge of receiving the documents (invoices and cancellations), and answering requests about the status of an invoice.

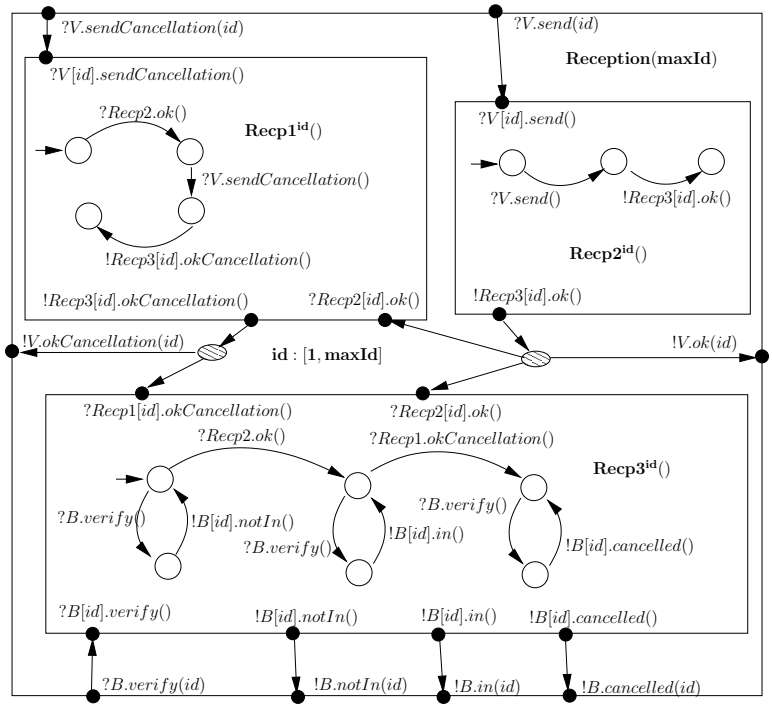


Figure 3.5: The reception and verification process

The process in Figure 3.5 is parameterized by the *id* of the invoice and is composed by three automata sets whose elements take care of one specific document *id* (of a given vendor). The top right automaton (**Recp2**) takes care of receiving an invoice, the top left automaton (**Recp1**) takes care of receiving a cancellation document and the bottom automaton (**Recp3**) gives the status of an invoice when requested. All three processes are parameterized by *id*.

The responses to an invoice status for a given *id* are or-exclusive: the invoice is not present at SII ($!B.notIn(id)$), the invoice has been sent to SII ($!B.in(id)$), or the invoice has been cancelled by the vendor($!B.cancelled(id)$).

In **Recp3**, initially an invoice is considered as not received. Upon reception, its status is changed to be present through a message sent by the **Recp2** ($!Recp3[id].okIn()$). Then, if a cancellation arrives for an invoice, its status is changed to be cancelled through a message sent by **Recp1** ($!Recp3[id].okCancellation()$). Note that the reception of a cancellation is only possible after the reception of the invoice to be cancelled (only after the transition $?Recp2.ok()$ is made).

The Global System

The global behaviour of the system is formed by an arbitrary number of vendors, buyers and a single SII as shown in Figure 3.6.

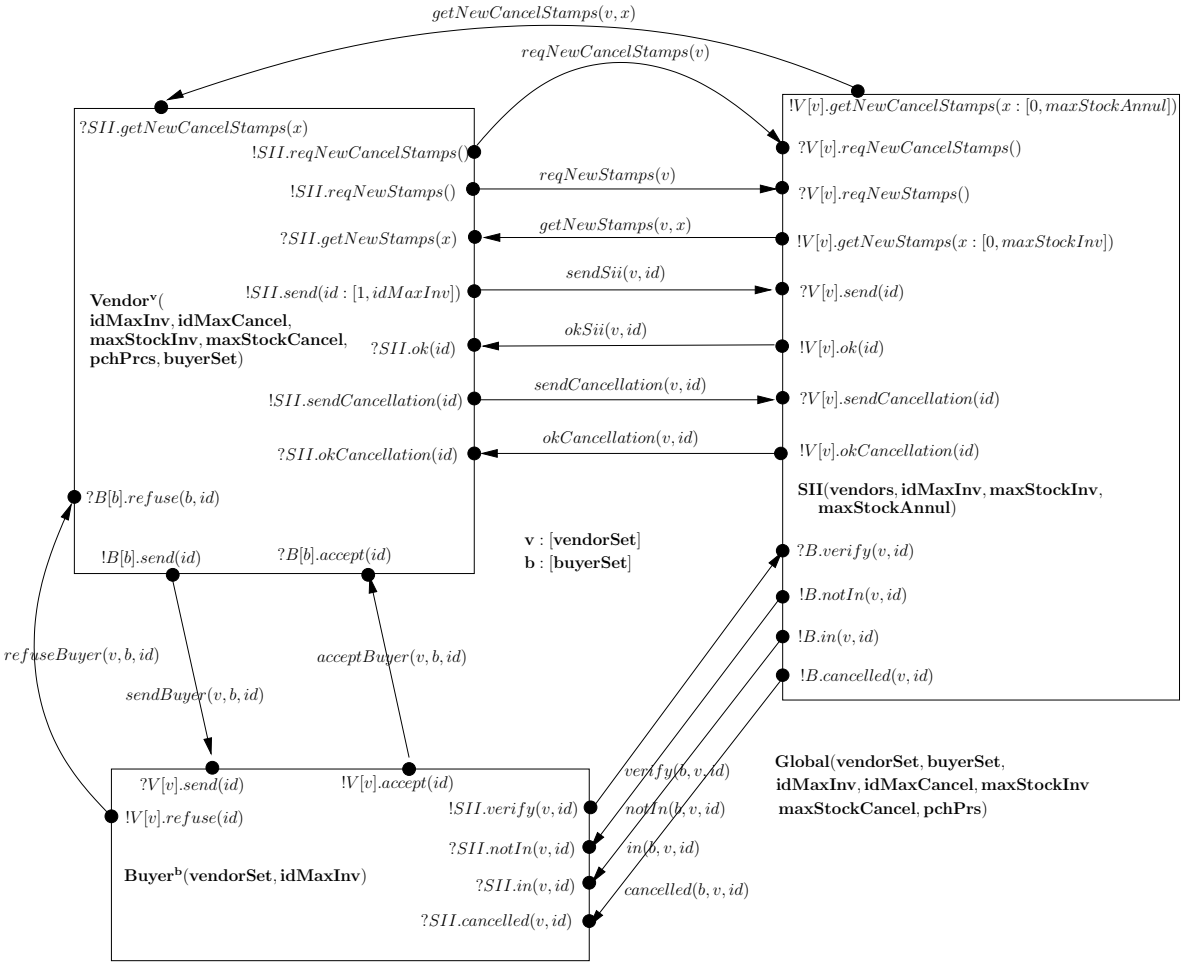


Figure 3.6: The global SII system

The synchronisation links are labelled so they become visible. Those links reflect the possible communications events, such as: requesting new stamps from the vendor v ($reqNewStamps(v)$), sending the invoice id from the vendor v to SII ($sendSii(v, id)$) or refusing the invoice id sent from the vendor v to the buyer b. The global system is fully described using 17 pLTSs structured by 7 pNets in 4 levels of hierarchy.

3.3.4 Properties Verification

The verification was done over the global synchronisation product of the instantiated processes and networks which form the system. The instantiation is made with the variable domains as described below.

Data domains

As we introduce in section 3.2.3, a finite instantiation of a parameterized model is an abstraction in the sense of [53, 148]. Starting with first order (countable) data domains, we define abstractions in which the abstract domain has values corresponding to a finite number of distinguished concrete values, plus one or more extra values representing the rest of the concrete domain. These abstractions define Galois insertions [53]. Such an abstraction will preserve a given formula if it has enough abstract values in the abstract domain of each parameter in the formula, to represent each distinguished value of the parameter in the formula.

We observe that all the properties listed in section 3.3.2 involve at most one buyer and/or one vendor. This does not mean that the property should be valid for only one specific vendor/buyer in the set of all the possible vendors/buyers, but for every possible combination of vendors and buyers as individual entities. Therefore, to verify the properties, it is sufficient to instantiate the system with two vendors and two buyers. In both cases, one encodes every vendor/buyer as an individual entity, and the second encodes the remaining vendors/buyers.

To have *many* invoices, as Property 1 states, we instantiate the maximal number of invoices to three (invoice $id \in [1..3]$): two encoding two particular invoices and the third encoding the rest of them. The stamps for invoices in the model are unbounded, only the stamp's stock capacity needs to be bound to get an instantiation. Since SII gives infinitely often authorisation stamps, the system can work with a minimal stock capacity of 1. However, we choose to set its capacity to 3 (the vendor can get as much as 3 stamps from SII at once) for having the scenario, among others, in which the vendor spends all the *id*s received from SII in a single request for authorisation stamps.

Since all the invoices can be potentially cancelled, we need at least the same quantity of cancellation *id*s as the quantity of invoices, therefore we instantiate the maximum number of cancellations to three. Following the same reasoning than the stamps for invoices, we also set the capacity of the cancellation stamp stock to 3.

Finally, we instantiate the purchase processes that a vendor can manipulate simultaneously to two: one encoding an individual process and the other encoding all the remaining processes that may be running during the life's cycle of the system.

Summarising, for verifying our 7 properties it is sufficient to instantiate the system with the variables values shown in Figure 3.7.

Max. Invoices	Max. Cancellations	Invoice stamps stock	Cancellation stamps stock	Purchase processes	Buyers	Vendors
3	3	3	3	2	$\{Vendor1, Vendor2\}$	$\{Buyer1, Buyer2\}$

Figure 3.7: Instantiation of data domains

Verification methodology

We have chosen in this case study to specify reachability properties, expressing scenarios that are desirable or not, by *abstraction automata*, a form of pLTSs with terminal states in which labels are predicates over parameterized actions. We believe this is simpler for non-specialists than having different formalisms for models and for properties. Alas this is not enough, and there are properties that cannot be checked this way, typically fairness or inevitability properties. For those we use directly the action-based temporal logic ACTL [60].

Reachability properties The use of *abstraction automata* for expressing and verifying reachability properties was advocated in the framework of the FC2Tools [36]. They are labelled transition systems with logical predicates in their labels, and with acceptance states. Each acceptance state defines one *abstract action*, representing a set of traces (a regular language) from the actions of the model we want to check.

From the original (concrete) system and the abstraction automaton (expressing the property), FC2tools builds a product LTS, whose actions are the labels in the acceptance states of the abstraction automaton encoding the property. If an action is present in the product LTS, then one of the corresponding concrete sequence is possible in the concrete system. The presence of an abstract action in the product system naturally proves the satisfiability of the corresponding formula, while its absence proves the negation of this formula.

For instance, Property 2 says: "SII gives the right answers to the invoice status request, i.e.: non present when it has not been sent to SII, present when it has been sent, and cancelled when it has been cancelled by the vendor".

This is a reachability property since it can be reformulated as *SII does not give wrong answers*, i.e. a scenario that should not be possible.

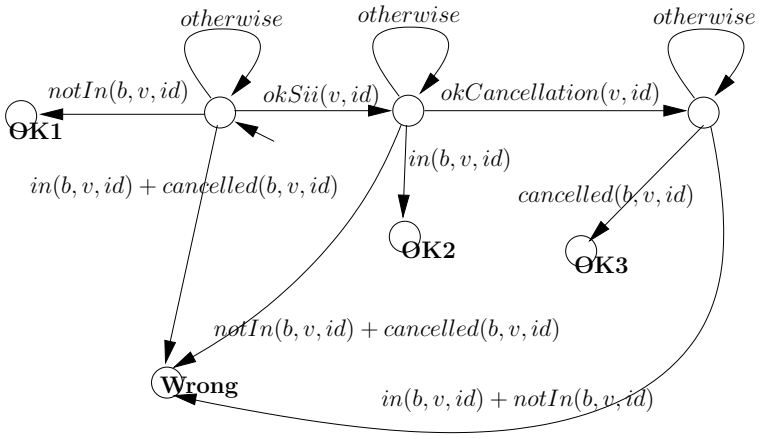


Figure 3.8: Abstraction automaton encoding Property 2

The abstraction automaton expressing this property is shown in Figure 3.8. In this automaton, the *otherwise* action means any other action different from the actions in the outgoing edges of the same state. In addition, the automaton not only expresses

that the responses are right (otherwise the state **Wrong** is reached) but also that they are possible (states **OK1**, **OK2** and **OK3** are reachable).

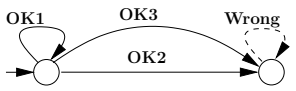


Figure 3.9: Property 2 verification result

We have used the FC2Tools to check this property : from the instantiated Net of the system, the tools build a global system minimised by weak bisimulation, then build its product with the property automaton, resulting in the LTS in Figure 3.9. In the LTS, the actions **OK1**, **OK2** and **OK3** are possible from the initial state, which means that the paths from the initial state to those acceptance states in the abstraction automaton (see Figure 3.8) are possible from the initial state in the instantiated system. Therefore we have proved that all the responses from SII to an invoice status request are possible. Additionally, since there are no **Wrong** actions possible in the initial state in the result, we conclude that the path from the initial state to the state labelled as **Wrong** in the abstraction automaton is not possible from the initial state of the instantiated system. The accurate reading of this **Wrong** action in Figure 3.8 is: a non-desired behaviour can happen if, in the system, we start from a state different from the initial one. Since we want to verify the property in the initial state, we have proved that SII does not give wrong answers to invoice status requests.

Temporal logic formulas The *abstract automaton* method of the FC2Tools is only usable for reachability properties. For other kinds of formulas, including fairness and inevitability properties, we use the EVALUATOR tool from the CADP tool-set [81]. EVALUATOR performs an on-the-fly verification of properties expressed as temporal logic formulas on a given Labelled Transition System (LTS). The used temporal logic is called regular alternation-free μ-calculus [129]. It allows direct encodings of "pure" branching-time logics including the action-based version of CTL, called ACTL [60].

For our case study we prefer to use the macros provided in CADP to express properties in ACTL, that may be more familiar to the reader, and we use EVALUATOR to verify the formula. The result of this verification is a *true* or *false* answer, and a diagnostics.

For instance, Property 3 is an inevitability property since it requests a scenario that must happen in a finite time, in all possible futures, under a condition. We reformulate it in a more precise way :

"If an invoice *id*, emitted by a vendor v to a buyer b is $\boxed{\text{refused by the buyer}}$
it will eventually be $\boxed{\text{cancelled by the vendor}}$ "

in which the boxed actions correspond respectively to the $refuseBuyer(v, b, id)$ and $sendCancellation(v, id)$ actions in the model.

We express this property using the following ACTL formula:

$$AG(refuseBuyer(Vendor1, Buyer1, 1) \Rightarrow AF\ sendCancellation(Vendor1, 1)) \quad (3.1)$$

To check this property, we have used our instantiation tool to produce a hierarchical Net, instantiated with the values of Figure 3.7, then the FC2Tools to compute a flat, minimised LTS for this system (in the FC2 format). This system was passed to the EVALUATOR model-checker, together with the formula. The result was positive: this formula holds from the initial state of the system.

The seven properties listed in section 3.3.2 have been successfully verified using this methodology. Two of them were specified using ACTL formulas. For a complete description see [26].

Improving the model through properties verification

The model we have introduced in section 3.3.3 was not the initial model we designed, but was improved after the verification of the properties listed in section 3.3.2. Some of the properties were not valid in this initial model and we had to review our formalisation to correct some parts. This verification and review not only improved the model but also the informal requirements defined by SII. For instance Property 5 says " It is not possible to cancel an invoice which has not been emitted before". Though it sounds obvious, we have not included this condition in the initial model since it is not explicitly written in the informal requirements. When verifying Property 2, we got undesired behaviours which exposed this lack in the initial model and so in the informal requirements. In fact, because of this experience, Property 5 was added to the verification list for having a more reliable formalisation. Without a formal verification as described in this paper, a programmer can easily overlook this condition during the implementation phase resulting in an application with potential and difficult to discover errors.

During this reviewing process, the instantiation tool proved very useful as a debugging tool of the system specification. We have done instantiations for small domains of variables to search the reasons why the properties were not valid in the system. Due to the size of the global LTS, those smaller instantiations were much easier to analyse than the complete instantiations.

3.3.5 Avoiding the state explosion

Once the system is instantiated, we should avoid generating directly the global LTS by brute force, i.e. without any pre-processing before calculating the global synchronisation product. This would lead us directly to the well-know *state explosion* problem. Some techniques that we exploit are the following: compositional hierarchy, parameterized representation and per-formula basis are shown during this section to limit the state explosion.

Nevertheless, we have produced small instantiations of the system by brute force. The idea is to have experiments showing how the variable domains affect the system size, to make an initial analysis, and to compare these results later with various techniques.

The results of this brute force instantiations are shown in Table 3.1. In the table, the numbers are the *states/transtitions* of the LTS (reduced by weak bisimulation) generated by the synchronisation product of the synchronisation networks describing the behaviour of the main actors (Vendor, Buyer, SII and the global system).

N^r	$idMaxInv$	$idMaxCancel$	$maxStockInv$	$maxStockCancel$	$pchPrcs$	$buyerSet$	$vendorSet$	**Vendor**	**Buyer**	**SII**	**Global**
1	1	1	1	1	1	1	1	140/860	6/9	64/348	752/2,816
2	3	1	1	1	1	1	1	5,404/37,162	216/972	16,384/168,960	58,960/290,208
3	1	3	1	1	1	1	1	140/860	6/9	64/348	752/2,816
4	1	1	5	1	1	1	1	420/3,456	6/9	64/476	2,256/10,896
5	1	1	1	5	1	1	1	420/3,432	6/9	64/476	2,256/10,752
6	1	1	1	1	5	1	1	51,428/347,944	6/9	64/348	278,256/1,199,648
7	1	1	1	1	1	3	1	404/2,500	6/9	64/348	3,088/11,824
8	1	1	1	1	1	1	2	140/860	36/108	4,096/44,544	565,504/4,235,264
9	3	3	1	1	1	1	1	8,812/63,710	216/972	16,384/168,960	90,064/462,208
10	1	1	1	1	1	1	3	140/860	216/972	262,144/4,276,224	*unknown*
11	2	2	2	2	2	2	2	4,950/38,697	1,296/7,776	*unknown*	*unknown*

Table 3.1: Brute force instantiations

An analysis of the values in Table 3.1 allows us to verify some aspects of the specification and potentially discover errors in it. In fact, there are some suspicious values observed in the instantiations 1 and 3: as we can observe, they share the same states/transitions numbers even when they are instantiated by different variable domains for the cancellation ids. However, as we explain in section 3.3.3, for each emitted invoice id, there is a process, and only one, in charge of the potential cancellation of it. Since this process, named **CI**, is the only one that requests ids for cancellation documents, and because this process is instantiated only once (invoices id = 1), there will be only one request for each cancellation id (independent of the number of cancellation ids available) and so the first and third instantiations are equivalent. Following the same reasoning, it is natural to observe different values for the instantiation number 9.

We also see in Table 3.1 how the variables impact the size of the instantiated processes. For example, the number of purchase processes strongly affect the size of the Vendor process (and so the global product), which is expected since it defines concurrent processes; but it does not affect at all the other process sizes since the purchase processes parameter is only relative to the Vendor. We also observe that the number of Vendors is the parameter that most drastically affects the size of the global system.

Note, that up to this point of the discussion, the tool to instantiate parameterized systems can be useful as an early debugging tool. The **unknown** values are due to memory constraints in the production machine, that did not enable us to generate the brute force product.

Structural actions hiding

When verifying properties, usually we do not need to observe all the events in the system. At each synchronisation product, we can hide the actions that are not involved in a specific property and which are not required to synchronise at upper levels of the system. This technique, in conjunction with minimisation, gives promising results.

We propose, on a per-property basis, to hide all the actions that are not explicitly in the property we want to prove.

For instance, let us recall Property 1: *A taxpayer could not emit invoices if it has not received stamps from SII. More specifically, a taxpayer can emit as many invoices as the quantity of stamps received from SII.* This property is shown formalised as an abstraction automaton in Figure 3.10.

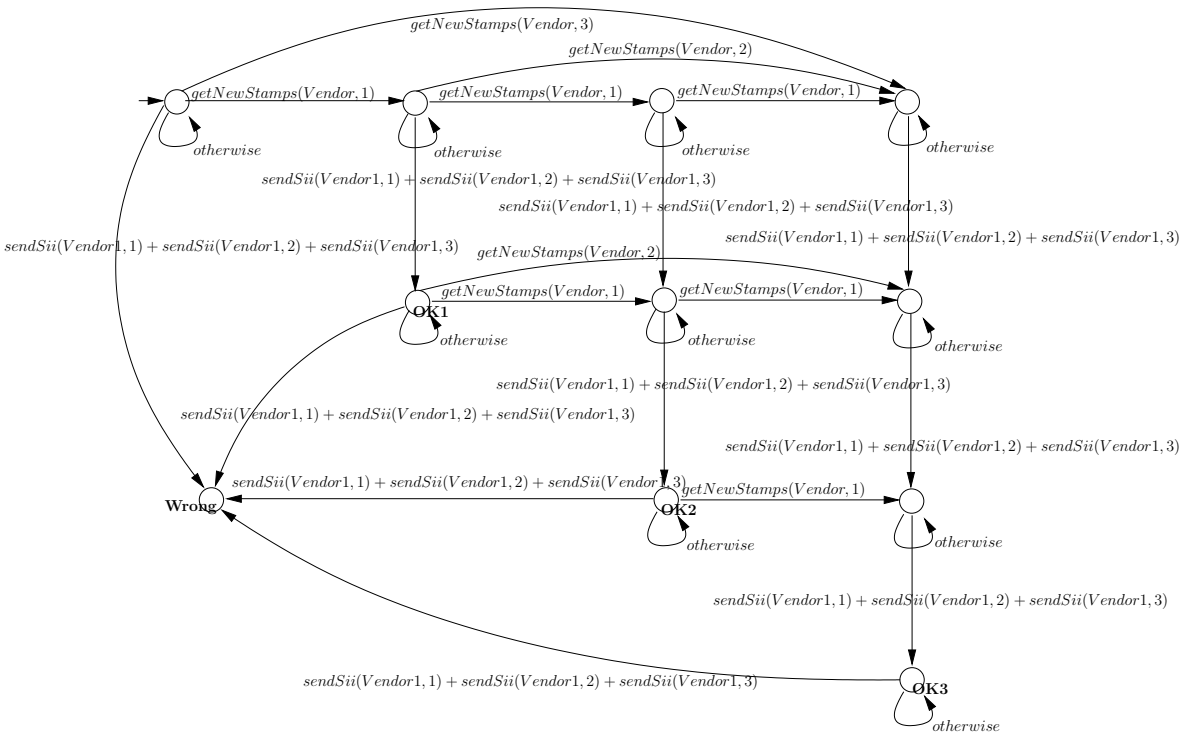


Figure 3.10: Abstraction automaton encoding Property 1

When minimising a system with hidden actions, the more actions you hide, the stronger reductions you obtain. For a given domain of variables, we can know how many links will exist in between two agents when instantiated.

Figure 3.10 has two groups of explicit actions: $getNewStamps(v, id)$, with $v = \{Vendor1\}$ and $id = \{1, 2, 3\}$; and $sendSii(v, id)$, with $v = \{Vendor1\}$ and $id = \{1, 2, 3\}$. The idea is to hide any other action that is not concerned by the property in the system. To hide the other actions means to consider any other action as the non-observable action τ.

Together with hiding, we successively generate the synchronisation product and we minimise it using weak bisimulation equivalence. We build the products by incrementally choosing at each level the pair of processes that share the most actions to be synchronised.

For a given domain of variables, we can know how many links will exist in between two agents when instantiated. For instance in the Vendor the communication $!PP[Pn].giveNewId(id)$ from the **Id** process to the **PP** process will be instantiated to a number of $domain(id) \times domain(p)$ communication links. Given the domain of variables, we propose to synchronise first, at each level, the pair of processes whose synchronisation product will have more hidden actions (i.e. there is no other pair where we can hide more communications than in this one).

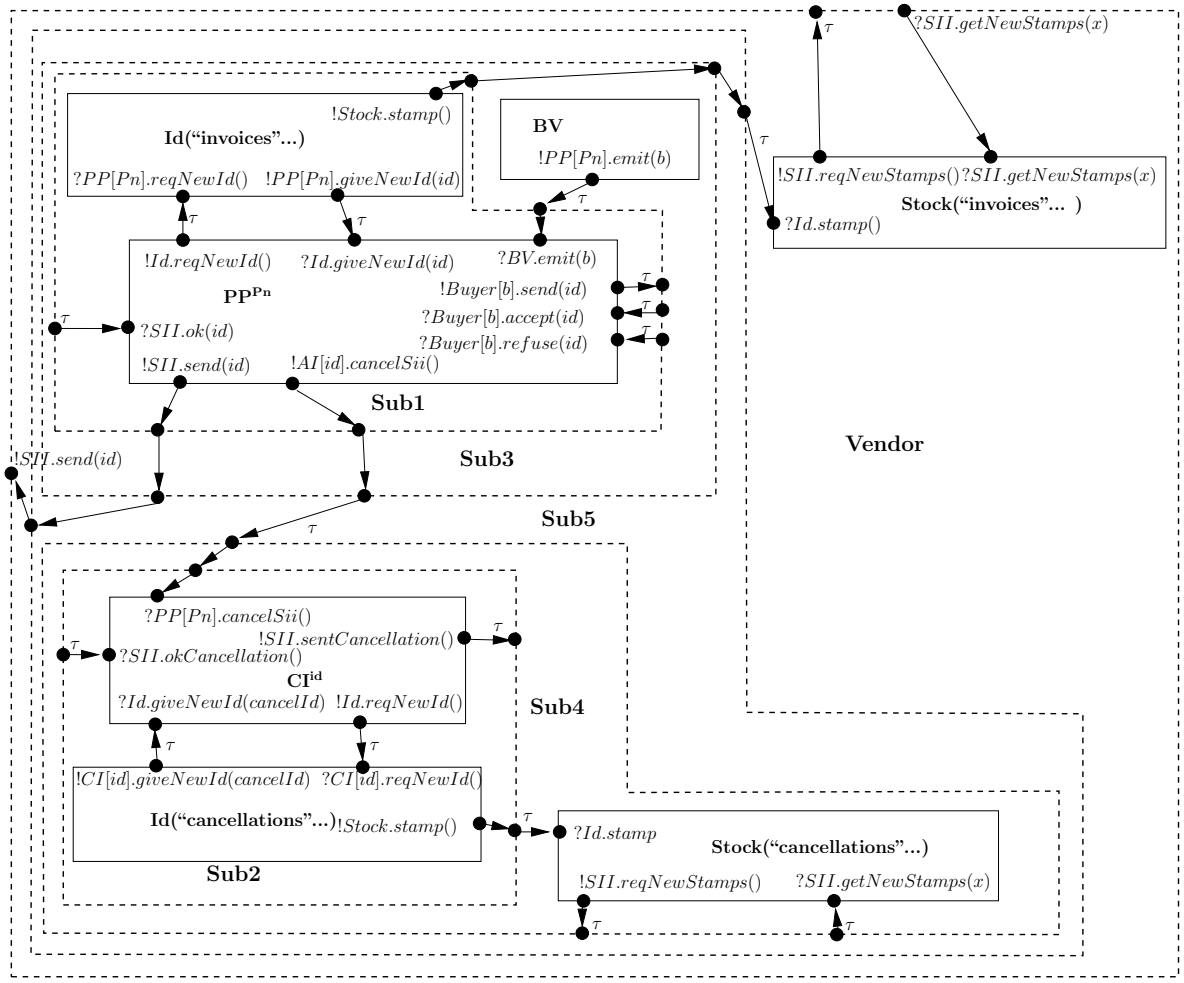


Figure 3.11: Vendor with structural hiding

Figure 3.11 graphically shows the composition using this technique for the Vendor and for the variable domains in Table 3.2 (significantly larger than those on Table 3.1). In Table 3.2, the numbers are the *states/transtitions* of the LTS generated by the synchronisation product of the synchronisation networks of each composition defined by the dashed lines in Figure 3.11 and using strong and weak bisimulation.

Note that the synchronisation product order depends on the variable domains. A different variable domain, such as in Table 3.1, will require a different order. We observe that this method enables us to scale up in the size of variable domains, compared to brute force instantiations (Table 3.1).

N^r	$idMaxInv$	$idMaxCancel$	$maxStockInv$	$maxStockCancel$	$pchPrcs$	$buyerSet$	**Sub1**				**Sub2**			
							Strong		Branch		Strong		Weak	
							before	after	before	after	before	after	before	after
1	3	3	3	3	2	2	1,832/4,036	492/1,139	1,652/3,688	189/484	494/1,119	140/302	383/900	20/36
2	3	3	5	5	2	5	10,754/24,415	492/1,433	9,674/22,285	189/664	494/1,119	140/302	383/900	20/36
3	3	3	5	5	3	5	518,704/1,712,810	4,810/21,414	469,009/1,572,215	1,165/6,319	494/1,119	140/302	383/900	20/36
							Sub3				**Sub4**			
							Strong		Weak		Strong		Weak	
							before	after	before	after	before	after	before	after
1	3	3	3	3	2	2	34/40	34/36	21/27	14/16	560/3,130	8/20	80/412	8/12
2	3	3	5	5	2	5	34/52	34/36	21/39	14/16	840/5,554	8/20	120/744	8/12
3	3	3	5	5	3	5	6,709/22,976	3,962/11,748	1,614/6,144	274/836	840/5,554	8/20	120/744	8/12
							Sub5				**Vendor**			
							Strong		Weak		Strong		Weak	
							before	after	before	after	before	after	before	after
1	3	3	3	3	2	2	95/189	31/61	44/43	8/7	124/674	124/550	32/136	20/68
2	3	3	5	5	2	5	95/189	31/61	44/43	8/7	186/1,119	186/1,013	48/254	28/126
3	3	3	5	5	3	5	6,178/23,771	605/2,166	406/1,131	67/156	3,630/29,187	3,630/25,557	402/2,710	227/1,388

Table 3.2: Vendor minimisation with structural hiding

Grouping by variables

The technique of structural actions hiding, described above, looks very promising when applied to the Vendor, as shown by the results in Table 3.2. If we try to apply the same reasoning to the global system as a whole, the first synchronisation product that we should make is the one shown in Figure 3.12.

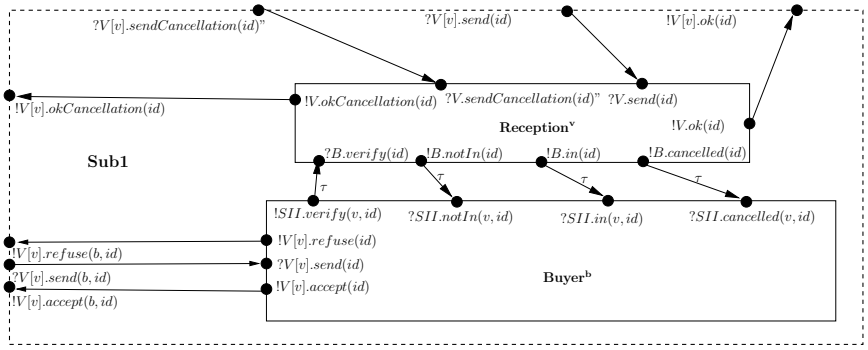


Figure 3.12: First composition for the global system

However, for any of the variable domains in Table 3.2, we run out of memory when generating the synchronisation product in Figure 3.12.

We propose a new method that benefits from the parameterized structure of the system. The idea is to group processes that share a common parameter. For instance, in Figure 3.5 is shown the structure of the **Reception** process. It is defined by a pNet that synchronises three processes (**Recp1**, **Recp2** and **Recp3**), each one parameterized by *id*. However, when instantiated, those synchronisations are made only between the three processes with the same value of *id*. So an instantiation of the **Reception** process is the interleaving of the synchronisation product of the three processes for each value of *id* in the instantiation. Therefore, for any instantiation we have the following equivalence:

$$Recp1^{id}|Recp2^{id}|Recp3^{id} \sim (Recp1|Recp2|Recp3)^{id} \tag{3.2}$$

Thus we apply hiding and minimisation to $(Recp1|Recp2|Recp3)$ before instantiating the id parameter. Naming the synchronisation product $Recp1|Recp2|Recp3$ as $SimpleReception$, and following the same reasoning, we can conclude the following strong equivalence:

$$\begin{aligned} System \sim ((IntBuyer^{b}|SimpleReception|CI)^{id}|PPPn| \\ BaseVendor|Id(invoices)|Stock(cancellations) \\ |GiveStamps(cancellations)|Id(cancellations) \\ |Stock(invoices)|GiveStamps(invoices))^{v} \end{aligned} \tag{3.3}$$

Mixing methods

Remember Property 1, which we are using to show our approach to generate the global LTS limiting as much as possible the *state explosion* problem: *A taxpayer could not emit invoices if it has not received stamps from SII. More specifically, a taxpayer can emit as many invoices as the quantity of stamps received from SII.*

Applying first a grouping by variables and then the structural actions hiding (for this property and the variable domains in Table 3.3), the global system is arranged as in Figure 3.13. The sizes of the intermediary synchronisation product and the global LTS, are shown in Table 3.3.

N^r	$idMaxInv$	$idMaxCancel$	$maxStockInv$	$maxStockCancel$	$pchPrcs$	$buyerSet$	$vendorSet$	**Sub1**				**Sub2**			
								Strong		Weak		Strong		Weak	
								before	after	before	after	before	after	before	after
1	2	2	2	2	2	2	2	280/1,065	280/1,065	280/1,065	80/328	595/2,490	595/2,490	176/792	128/576
2	3	3	3	3	2	2	2	280/1,065	280/1,065	280/1,065	80/328	595/2,595	595/2,595	176/824	128/608
								Sub3				**Sub4**			
								Strong		Weak		Strong		Weak	
								before	after	before	after	before	after	before	after
1	2	2	2	2	2	2	2	18,521/59,992	1,360/4,947	4,757/13,392	375/1,448	228/573	216/536	71/178	57/150
2	3	3	3	3	2	2	2	384,835/1,630,516	10,545/51,650	47,385/178,932	1,861/10,246	1,488/5,669	1,389/5,305	325/1,328	251/1,048
								Sub5				**Sub6**			
								Strong		Weak		Strong		Weak	
								before	after	before	after	before	after	before	after
1	2	2	2	2	2	2	2	167/423	167/389	47/124	30/70	6/13	6/13	6/13	1/1
2	3	3	3	3	2	2	2	725/3,035	725/2,751	146/676	93/379	8/20	8/20	8/20	1/1
								Sub7				**Sub8**			
								Strong		Weak		Strong		Weak	
								before	after	before	after	before	after	before	after
1	2	2	2	2	2	2	2	312/996	204/661	42/86	25/55	2,061/7,253	24/47	51/84	6/5
2	3	3	3	3	2	2	2	2,776/10,564	1,977/7,544	278/687	169/449	19,304/81,931	34/67	253/538	8/7
								Sub9				**Sub10**			
								Strong		Weak		Strong		Weak	
								before	after	before	after	before	after	before	after
1	2	2	2	2	2	2	2	72/354	72/354	18/66	12/45	144/492	144/492	24/51	12/33
2	3	3	3	3	2	2	2	136/740	136/740	32/136	20/88	272/1,004	272/1,004	40/100	20/68
								Global							
								Strong		Weak					
								before	after	before	after				
1	2	2	2	2	2	2	2	20,736/141,696	20,736/120,960	144/792	144/792				
2	3	3	3	3	2	2	2	73,984/546,176	73,984/472,192	400/2,720	400/2,720				

Table 3.3: Global system grouped by variables and structural hiding

This combination of techniques has enabled us to scale up to a variable domains size that we could not handle before due to the *state explosion* problem. All the verification

of properties, described in section 3.3.4, were done in the global LTS generated using this methodology.

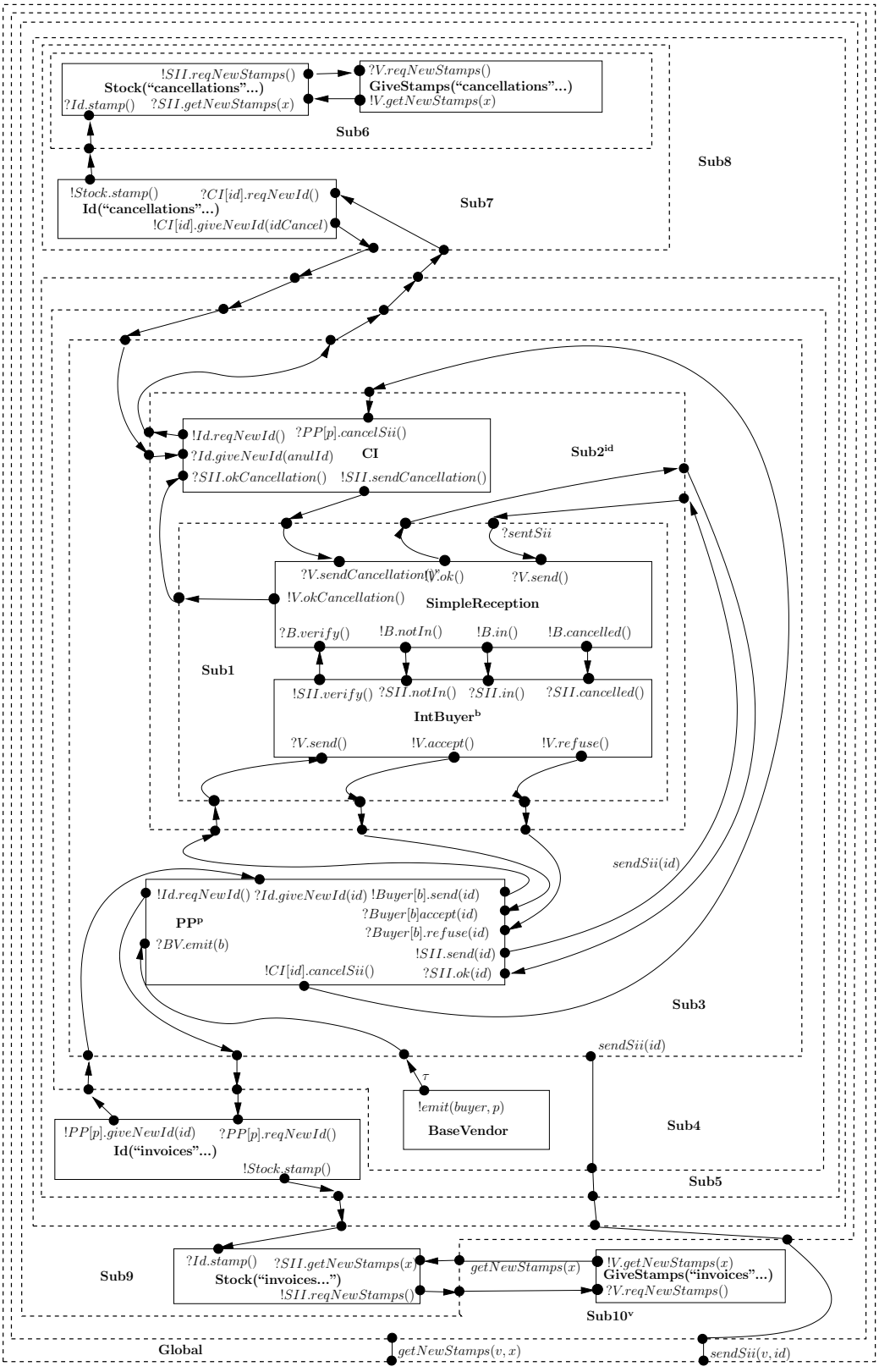


Figure 3.13: Global system results when grouping by variables and using structural hiding

3.3.6 Related Work

A similar case study is done by Tronel and all in [152] for the ScalAgent deployment protocol. ScalAgent is a platform for embedded systems, written in Java, to configure, deploy, and reconfigure distributed software. In [152] they make a fully automatic verification for a Ups (*Uninterruptible Power Supply*) management system for large scale sites, deployed in ScalAgent. Similar to us, they model the system as networks of communicating LTSs, which exchange messages by rendez-vous communications. In contrast with us, they start from a formal description of the system thanks to the infrastructure of ScalAgent, which describes its configuration in XML. We use a graphical approach to formalise the system from the informal description, and we translate the informal requirements to formal properties to be checked. In [152] they have chosen to make an automatic translator from the XML description to LOTOS [104]; the proofs are done by reachability analysis of *ERROR* states, which are defined into those XML descriptions.

The main advantages of [152] are: they use a fully automatic approach, and they directly use the tools from CADP which already includes hiding mechanisms and the use of interfaces constraints [50]. They also use parameters but included in the translation to LOTOS and not directly in the formal models as we do. This does not allow them to profit from the parameterized structure of the system to get better minimisations. They determine the variables domains by static analysis. Similar to us, they make finite instantiations for different parameters domains, and they use this instantiation capacity to do debugging and analysis. They find the minimal required instantiations to check the properties by empirical analysis.

Finally, even when we use the same theories and methods to check properties, our aims are different. In [152] they have developed a fully automatic verification methodology specific to ScalAgent. For us, our study case was analysed with the aim to address any kind of distributed application with asynchronous communications, and also to include the verification of implementations. As we said before, our models are suitable also as models generated from source code.

3.4 Conclusions

We have introduced a method and a formalism to formally describe distributed systems and verify their properties, and we have validated our approach through a case study of a real system, the Chilean electronic invoices. We argue that this method is suitable to a developer, not necessarily with expertise in formal methods, by following the methodology used on this case study.

We focus in the behaviour properties. Other analysis such as the data flow or data security require other specialised analysis and/or tools. The contributions of this chapter relies in the following points:

- We have defined a framework to describe in a natural manner the behaviour of distributed systems (with parameters) via network of processes. This language is a combination and an extension of works from [15] and [111]. We have introduced as well a graphical syntax to describe those networks.

- Our parameterized models achieve three different roles: they describe in a natural and finite manner infinite systems (when considering unbounded variable domains), they describe a family of systems (when considering various variable domains) and they describe in a compact way large systems (when considering large variable domains)
- We have developed a tool for obtaining finite non-parameterized systems from our language given the variable domains. This tool called FC2Instantiate is deeply introduced in chapter 6.
- Using the graphical syntax, we have shown how to model the Chilean electronic invoices system from its informal specifications. The system is fully described using 11 pLTS that synchronise in 7 pNets through 4 levels of hierarchy. In total, the model contains 27 parameterized synchronisations.
- Once instantiated, we generate the LTS describing the whole system behaviour by incremental synchronisation of processes. We group by parameters and we use hiding with minimisation at each level of synchronisation to limit as much as possible the *state explosion* problem. Before, we were limited to generate a global LTS with around $5,6 \times 10^5$ states (see Table 3.1), using this methodology, we were able to produce a global LTS equivalent to one having approx $1,2 \times 10^{12}$ states if generated by brute force with the variable values in Table 3.7.
- Finally, we have shown how to verify safety and liveness properties of the system, using our instantiation tool and classical finite-state model-checking tools.

Additionally, the instantiation tool is suitable (given small instantiations) for comparing different instantiations, instantiating based on per-formula criteria and searching for better minimisations. Especially this debugging capacity provides early detection of errors or backtrack analysis.

Chapter 4

Hierarchical Components Behaviour

As we state in the introduction (section 1.5), components have emerged as a new programming paradigm in software development. Beyond structuring concepts inherited from modules and objects, component frameworks provide means for architecture and deployment description. Some frameworks define a number of non-functional features for controlling the life-cycle of the components and the application, or allow the construction of distributed components. In general words, a component is a self contained entity that interacts with its environment through well-defined interfaces: provided services and required functionalities (to be provided by other components). Besides these interactions, a component does not reveal its internal structure.

Note that we use the term "non-functional" in the sense of "component management or control", and not in the sense of "quality of service" which can be used in other contexts, for example in time-sensitive models

In hierarchical component frameworks like Fractal [43], different components can be assembled together creating a new self contained component, which can be itself assembled to other components in a higher hierarchical level. Hierarchical components make visible the hierarchy of the system and hide, at each level, the complexity of the subentities. The compositional aspect together with the separation between functional and non-functional aspects helps the implementation and maintenance of complex software systems.

The challenge that we want to address is to build a formal framework which ensures that applications built from existing components are safe, in the sense that their parts fit together appropriately and behave together smoothly. Each component must be adequate for its assigned role within the system, and the update or replacement of a component should not cause the rest of the system deadlock or fail.

Standard components systems have typed interfaces, ensuring some level of static compatibility between the components: interfaces are bounded only if their operations have compatible types in the classical sense (OO method typing). This does not prevent assembled components from having non compatible behaviours, that could lead to deadlocks, live-locks, or other kinds of safety problems.

Several recent works try to address better dynamic guaranties, e.g. research on behavioural typing or contracts [45], as well as frameworks like Tracta [86], Wright [13] or Sofa [146].

Those approaches were developed to cover the correct component's composition,

considering their functional aspects. However, components programing allows, through the non-functional capabilities, to control the execution of a component and its dynamic evolution: plugging and unplugging components dynamically provides adaptability and maintenance. Therefore this inter-play between functional and non-functional aspects influences the behaviour of the system, even if we look at the components from a pure functional point of view.

Moreover, those approaches start from a strong assumption: the system has been correctly deployed in its initial construction, in particular all the necessary paths between functional interfaces have been established and the system has been started in a coherent state.

We aim to provide the final user with tools for verifying the behaviour at the design phase (definition), the assembly phase (implementation), as well as the dynamic reconfiguration (maintenance) of the component system. Therefore, the intended user of our framework is the application developer in charge of those tasks.

All those phases, usually defined by our target user, through some architectural definition language or graphical interface, are error-prone. Hence the need to provide tools not only for verifying a given composition's correctness, but also that the steps for building such composition (or a further reconfiguration) have been correctly applied.

Wrigth does not support nested components, necessary for the hierarchical component model we target. On the contrary, Tracta (Darwin) does support nested components but does not support dynamic changes on the software architecture. An interesting approach for modelling main description's mechanism of dynamic structures in Darwin (*lazy instantiation* and *direct dynamic instantiation*) is given by J. Magee and J. Kramer in [121]. They show how to model the system's behaviour using π-calculus [133], but neither property checking nor tools are provided.

SOFA supports both nested components and some level of dynamism by checking whether a particular component can dynamically replace (or not) an existing one. In particular, its *consent* operator allows checking the absence of errors in the interaction of the new component with its environment. However, it is not clear to us how a dynamic change on the paths between interfaces can be done without replacing the full architecture where it occurs. In addition, whether a component can be replaced at a given moment or not should be explicitly set in its behaviour through the update token π of SOFA.

We propose an approach orthogonal and probably complementary to SOFA (we analyse this point in section 7.1, future work). Our approach is to give the components behavioural specifications in the form of hierarchical synchronised transition systems. Similar to the other approaches we have reviewed, we assume that the models for the functional behaviour of basic (primitive) components is known. They may be derived from automatic analysis of source code or expressed by the developer in a dedicated specification language, e.g. the graphical language for synchronised automata introduced in chapter 3, section 3.2.2. Then, we automatically incorporate the non-functional behaviour within a controller built from the component description. The composite's semantic is computed as a product of the its sub-components LTSs with the controller of the composite. This system can be checked against requirements expressed as a set of temporal logic formulas, or again as an LTS (defining an abstract

specification).

In this chapter we start with synchronous components allowing us to introduce the main ideas of our approach in a simpler form (which we extend in chapter 5 to asynchronous components). In particular in this chapter we give:

- a methodology for building behavioural models of hierarchical components, including non-structural reconfiguration operations,
- the full behaviour modelling of the application as a hierarchy of parameterized LTSs,
- a structural reconfiguration description as transformations of the LTS expressing the component behaviour,
- a correctness properties classification for a component system, together with tools for their verification.

In section 4.1 we shortly overview our target component model: Fractal, and introduce a small example that will serve as an illustration for the rest of the chapter. Section 4.2 discusses the notion of correct behaviour and shows how our formalism (chapter 3) fits nicely with the components scenario. Section 4.3 develops, step by step, the formalisation and the behaviour computation of the example, starting with the specification of base components, then building the composite controllers, specifying errors, computing the composite behaviour and building specific abstract models useful for representing deployment and reconfiguration phases. In section 4.4.2 we give examples of proofs for some system properties and in section 4.6 we conclude.

4.1 The Fractal Component Model

The Fractal component model [43] provides an homogeneous vision of software systems architecture using few well defined concepts such as: component, controller, content, interface and binding. It is recursive in the sense that components structure is auto-similar at any arbitrary level, hence the name "Fractal".

4.1.1 Guidelines to Fractal Components

A Fractal component is formed out of two parts: a *controller* and a *content*. The content of a component is composed of (a finite number of) other components, called *sub-components*, which are under the control of the controller. A component that exposes its content is called a *composite* component. A component that does not expose its content, but at least one control interface, is called a *primitive* component.

The controller of a component can have *external* and *internal* interfaces. A component can interact with its environment through *operations* on its external interfaces, while internal interfaces are accessible only from the component's sub-components.

Interfaces can be of two sorts: *client* and *server*. A server interface can receive method invocations while a client interface emits methods calls. A *functional* interface provides or requires functionalities of a component, while a *control* interface is a

server interface that corresponds to a "non functional aspect", such as introspection, configuration or reconfiguration.

A *binding* is a connection path between a component client interfaces. A binding between a client interface c and a server interface s of two components C and S must verify one of the following constrains:

- c and s are external interfaces, and C and S have a direct common enclosing component. Such bindings are called *normal bindings*.
- c is an internal interface, s is an external interface, and S is a sub component of C. Such bindings are called *export bindings*.
- c is an external interface, s is an internal interface, and C is a sub component of S. Such bindings are called *import bindings*

Additionally, a primitive binding can be established only if the server interface accepts at least all the operations invocations that the client interface can emit, and a client interface can be bound to at most one server interface, while several client interfaces can be bound to the same server interface.

A component controller encodes the control behaviour associated with a particular component. In particular, a component controller can intercept oncoming and outgoing operation invocations and operation returns targeting or originating from the component's subcomponents; and it can superimpose a control behaviour to the behaviour of the components in its content, including suspending and resuming activities of these components. Each controller can thus be seen as implementing a particular composition operator for the components in its content. The Fractal model does not place a priori restrictions on the forms of control and composition a component controller can realise: it can be mainly interception-based as in industrial component frameworks containers for instance; it can be limited to provide a common execution context for the components in its content; or it can realise intrusive forms of superimposition.

Fractal defines three basic (optional) levels of control capabilities for a component: no control at all, introspection, and configuration. Only the latter is of interest to us. At the configuration control level, Fractal proposes four control interfaces:

- Attribute control: provides operations to get and set attribute values of the component.
- Binding control: provides operations to bind and unbind the component client interfaces to other component server interfaces.
- Content control: provides operations to add and remove sub-components into/from the component.
- Life cycle control: provides operations for starting and stopping the component, as well as to get its current status (started/stopped).

The Fractal specification defines a number of constraints on the interplay between functional and non-functional operations. In particular :

- Content and binding control operations are only possible when the component is stopped.
- When started, a component can emit or accept invocations. Note that this does not prevent control operations to throw an error (exception) because of an unstable state.
- When stopped, a component does not emit invocations and must accept invocations through control interfaces; whether or not an invocation to a functional interface is possible is undefined.

Other features are left unspecified in the Fractal definition, and may be set by a particular Fractal implementation, or left to be specified at user level. We assume the following choices:

1. the start/stop operations are recursive, i.e. they affect the component and each one of its sub-components simultaneously;
2. functional operations cannot fire control operations.
3. the controller (membrane) of composites is only a forwarder between external and internal functional interfaces without any other control capability;

The last feature (3) implies that there is exactly one internal interface for each external interface of a composite.

4.1.2 Component System Example

In this section we introduce a simple component system as an example, which we will use later to better explain our work. Figure 4.1 is a graphical view of it.

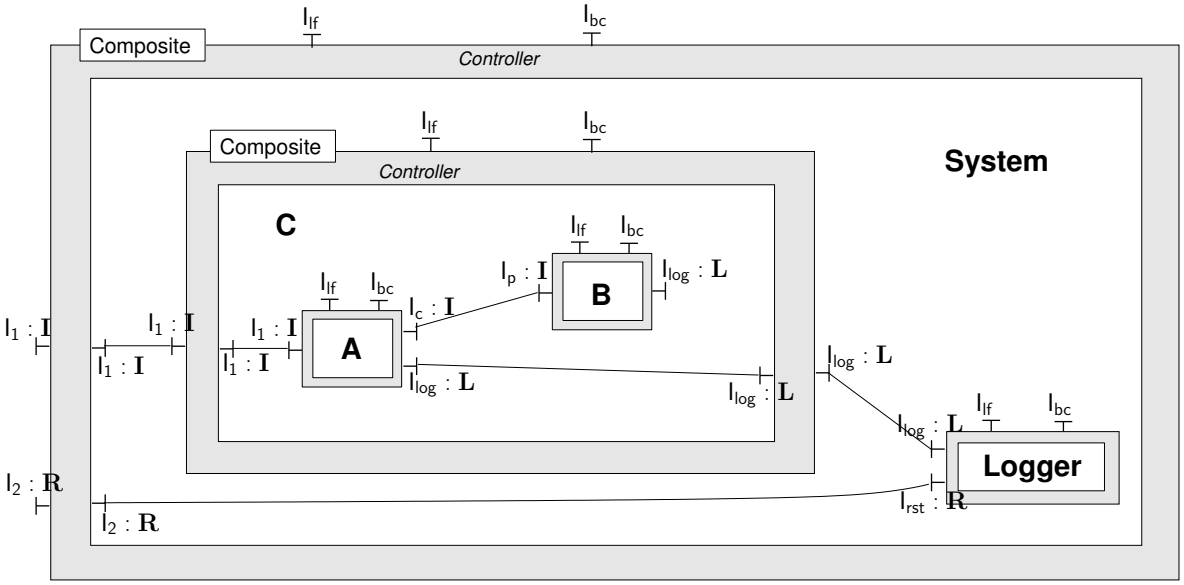


Figure 4.1: A simple component system

The example is built from three *primitive* components (**A**, **B** and **Logger**), which are composed in two levels of hierarchy defined by two *composite* components (**C** and

System). Each component exposes the interfaces for the control operations they support (in our example all the components support life-cycle control operation through the interface l_{lf} and binding control operations through the interface l_{bc}).

All the functional interfaces in the example are typed either by the type **I**, the type **L**, or the type **R**. We define the type **I** having the operation foo(), the type **L** having the operation log() and the type **R** having the operation reset().

The system is deployed in a bottom-up fashion from the innermost components to the outer component (**System** in our example). At each level of hierarchy a specific deployment is applied. For instance, at the **C** level of hierarchy in Figure 4.1 the deployment includes, among others, the binding between the interface l_{c} of **A** and the interface l_{p} of **B**.

4.2 Defining Correct Behaviour

Control (i.e., non-functional) operations can introduce changes on the component behaviour. For instance, adding or replacing a sub-component may add features (new actions) to the system. A sequence of control operations is called a *transformation phase*.

We make the assumption (this is a restriction with respect to the Fractal specification) that no functional operation can fire control operations but those are fired by our target user during deployment or in a further reconfiguration (transformation phase).

Then we are interested in three phases in the components behaviour:

1. Deployment: this is the building phase of a component. In this phase the component's content (its sub-components) is defined as well as the initial transformation phase (sequence of control operations), as usually described in the application ADL. The application deployment typically ends with a recursive start operation.

2. Running phase: only functional operations occur here.

3. Reconfiguration: we distinguish between non-structural reconfigurations (life cycle and binding controls) and structural transformations (adding, removing or updating components).

From these definitions, we discuss the *correctness* of the component system:

1. *Deployment*: "Is the deployment possible, does it finish?", if so, "has it been correctly applied?" in the sense that unexpected behaviours have not been raised during this phase.

2. *Running phase*: "Is the deployed system behaving correctly?". The concept of "correct behaviour" covers the absence of dead-locks and in general safety and liveness properties (common sense properties like not using an unbound required interface, or any user-requirement expressed as a temporal logic property). Ultimately, "Does this implementation respect a pre-defined specification? (with respect to some implementation pre-order)".

3. *Reconfiguration*: "After a transformation phase, does the system behave correctly?". This covers both preservation of some properties valid before the transformation, and the satisfaction of a new set of properties, corresponding to features added by the transformation. These proofs must take into account the intricate interplay between functional and non-functional actions during transformation, like the management of the internal state of subcomponents. For example, one can expect to be able to prove the safety and transparency (from the user point of view) of the replacement of a component by another one.

We want to provide the user with tools that help answering those questions **before** deploying the application or applying a transformation, so he can be confident about the reconfigurations he will apply and therefore, be sure he has a reliable system.

4.2.1 Components behaviour specification

Our formalism fits nicely with the components model. The behaviour of a primitive component is a pLTS, that can be specified by the developer, or derived from code analysis. For a given composite, its content is the arguments of the pNet and its initial bindings are encoded in the initial state of the transducer. The global pLTS of a composite encodes the functional behaviour of the component but also the control operations that do not change the geometry of the composite, namely start/stop, and bind/unbind operations. On this model, we can check all properties during and after the "initial composition", and involving reconfigurations only relying on start, stop, bind, and unbind.

We deal with reconfigurations that change the dimension of the pNet or the structure of the application (add/remove/update of components) as transformers of the model: starting with a hierarchical model in a given state, we build a new model after a sequence of basic reconfigurations, in which we maintain the state of the components that were unchanged. We can then check for the properties (preserved or new) of the reconfigured system.

4.3 Building the component's model behaviour

In this section we introduce our way to build the behaviour of a component system for the example introduced in 4.1.2. In section 4.3.4 we generalise this method for any component system.

Since reconfiguration phases change the behaviour of a component, we need to build the set of all the behaviours after applying those transformation phases. For non-structural transformations, this means in our formalism to build the component's transducer, where the transition between different states are fired by control operations. For instance, in the composite **C** the communication between **A** and **B** is not possible until the interface I_c in **A** is bound to the interface I_p in **B**. Then the control operation that binds those interfaces corresponds to a transition to a state where the communication becomes possible.

We build the transducers of our models using several small controller pieces that are composed with the sub-components using a synchronous parallel operator (no event is

possible if it is not possible in the current state of the controller parts). The result of this building is an automaton which we named as *Controller*.

4.3.1 Primitive components

We suppose the functional behaviours of the primitive components are known, as we stated before they can be obtained by source analysis or given by the user. The functional behaviour of a primitive component is expressed as an automaton with labels encoding methods calls and receptions in its interfaces as well as internal actions.

The primitive components can be implemented as basic runtime entities (such as Java objects) to which control operation capabilities are added (for instance by the developer in the source code, by reflection, using a Fractal Factory or such as in FracTalk [1] by a variable names convention).

The functional behaviours of the primitive components **A**, **B** and **Logger** are shown in Figure 4.2 in the form of pLTS.

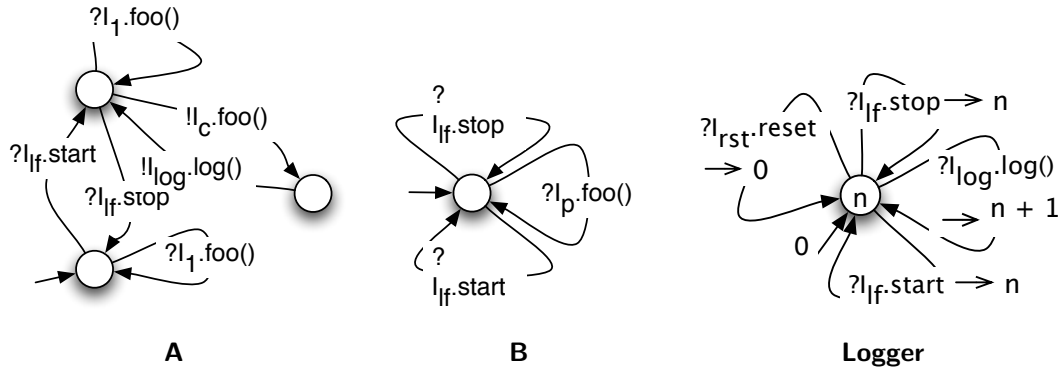


Figure 4.2: Behaviour of the base components of **A**, **B** and **Logger**

Logger provides a logging functionality through its provided interface $\mathsf{l}_{\mathsf{log}}$ up to n calls. If a call is done to the method $\mathsf{reset}()$ of its interface $\mathsf{l}_{\mathsf{rst}}$, **Logger** restart the count of logging calls to 0. We intentionally do not include value passing in the communications to keep the notations simple.

We start by adding the control capabilities (life-cycle and binding) to **A**, i.e. building its controller. **A** provides one interface (l_1) and requires two interfaces (l_{c} and $\mathsf{l}_{\mathsf{log}}$). Since **A** is a primitive component, only the external views of its interfaces are needed (there is no internal interface since there is no internal binding in **A**). Then we build the controller for **A** as the synchronisation product of the 5 LTSs composing the Net in Figure 4.3. In the figure, we have drawn in the upper right hand the Fractal specification for an easy reference. The orientation of the links in the figure are used to help the visual view and understanding of the system, but they do not have any semantical meaning.

In Figure 4.3 we can see the automata encoding the control operations for the external requires interfaces l_{c} ($\mathbf{E\_RI_c}$) and $\mathsf{l}_{\mathsf{log}}$ ($\mathbf{E\_RI_{log}}$), and for the external provides interface l_1 ($\mathbf{E\_PI_1}$). We also see the automaton encoding the life-cycle control opera-

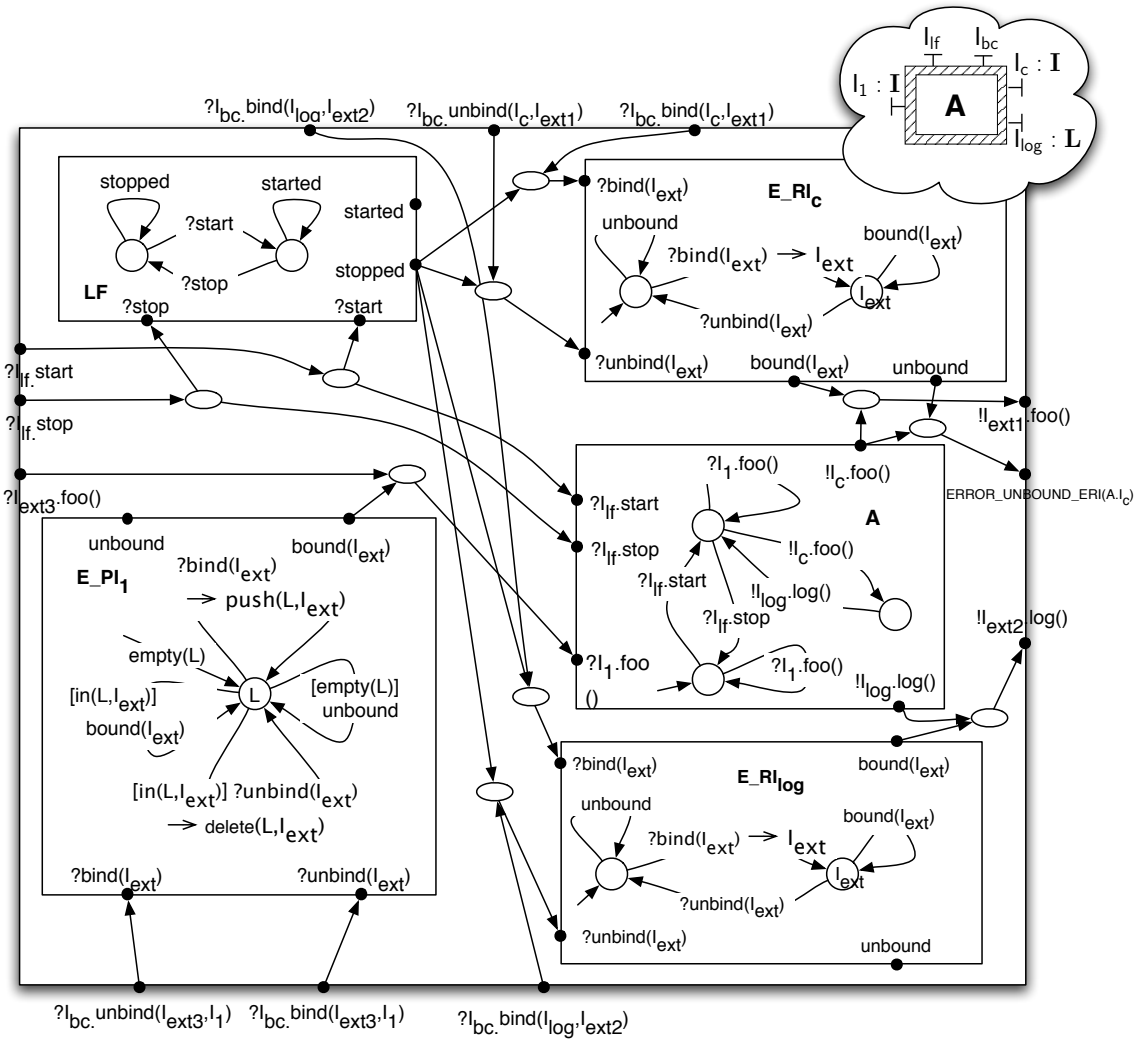


Figure 4.3: Controller for **A**

tion (**LF**) and the functional behaviour of **A**. Synchronised actions are encoded by links between processes.

Figure 4.3 includes some constraints (of common sense or from the Fractal SPEC) e.g. that the bindings of requires interfaces are only possible when the component is stopped or that calls to requires interfaces are only possible when these interfaces are bound.

In the graphics we use an ellipse when more than two actions are synchronised. E.g. a reception of a start action on the non-functional interface $\mathsf{l_{lf}}$ is propagated synchronously to the **LF** automaton and to the **A** behaviour automaton; on the reception of a unbind action on the $\mathsf{l_{bc}}$ interface will be transmitted to the corresponding **E_RI** control automaton only if the **LF** is in stopped state.

In the functional behaviour of **A** (Figure 4.2) we observe the presence of the non-functional actions $?\mathsf{l_{lf}}.\mathsf{start}$ and $?\mathsf{l_{lf}}.\mathsf{stop}$. We do not want to break the separation of concerns reached by Fractal but only to keep both simplicity and some generality during this chapter. Our main target is distributed component systems communicating

asynchronously (which we develop in chapter 5). Specifically we target the Fractive implementation of Fractal [29]. Momentarily we assume for our example that the functional part of the primitives are conscious of the life-cycle control interface.

The Fractal specification does not define all details of the controller constraints and semantics; some features will only be defined by specific implementations. For instance, in the Fractal implementation Julia [2], the primitive components are enriched with *interceptors* whose role is to suspend the incoming method calls while the component is stopped. If we were modelling for the Julia implementation, we could express this constraint in our approach by linking the started port of the **LF** controller part to the reception of methods calls.

In Figure 4.3, $\mathsf{I}_{\mathsf{ext}*}$ are variables encoding the set of external interfaces to which the interfaces of **A** can potentially be bound. This set is instantiated at the next level of hierarchy by type matching analysis (i.e. once its environment is defined). For instance when building the controller of **C**, the variable $\mathsf{I}_{\mathsf{ext1}}$ in the figure becomes the set $\{\mathsf{B}.\mathsf{I}_{\mathsf{P}}\}$.

The *Controller* of **A** is the automaton resulting from the synchronisation product of Figure 4.3. This controller encodes both, the functional and non-functional behaviours of **A**. We use the controller to compute the behaviour of the component after deployment as shown later.

Using the same methodology, we build the controllers for the other primitive components **B** and **Logger**.

4.3.2 Composites

This section describes the method we use to build the *Controller* of a given composite. The model of the composite is a parameterized Network, which arguments are the models of its subcomponents. As for primitives, this model includes both the functional aspects of the behaviour (coming eventually from the user specifications of basic components), and the non-functional management aspects (that we automatically generate from the ADL specification of the composite).

The global behaviour of the application will be computed later, building the synchronous product for each composite component in a bottom-up fashion, after instantiation of the parameters. At each level, only a selected set of actions (functional or non-functional) will be observable. This allows for a *grey-box* construct, in which an reduced model can be constructed for proving a given formula or set of formulas.

The controller for the composite **C** is the synchronisation product of the 7 LTSs composing the Net in Figure 4.4. We use a syntax $\mathsf{C}.\mathsf{I}$ to designate an interface I belonging to a component C. When C is absent, the interface belongs to the component itself, i.e. to the component of the controller. The arguments for the bind and unbind operations are always a *client* interface in the first argument and a *server* interface in the second.

We distinguish in the figure the internal control operations, which are labelled inside the controller Net (e.g. $?\mathsf{bind}(\mathsf{A}.\mathsf{I}_{\mathsf{log}}, \mathsf{I}_{\mathsf{log}})$), from the external control operations, which are in the edge of the Net (e.g. $?\mathsf{I}_{\mathsf{bc}}.\mathsf{bind}(\mathsf{I}_{\mathsf{ext2}}, \mathsf{I}_1)$). The internal control operations are those used during the component's deployment, while the external control operation are used during the deployment of the next level of hierarchy. Since the

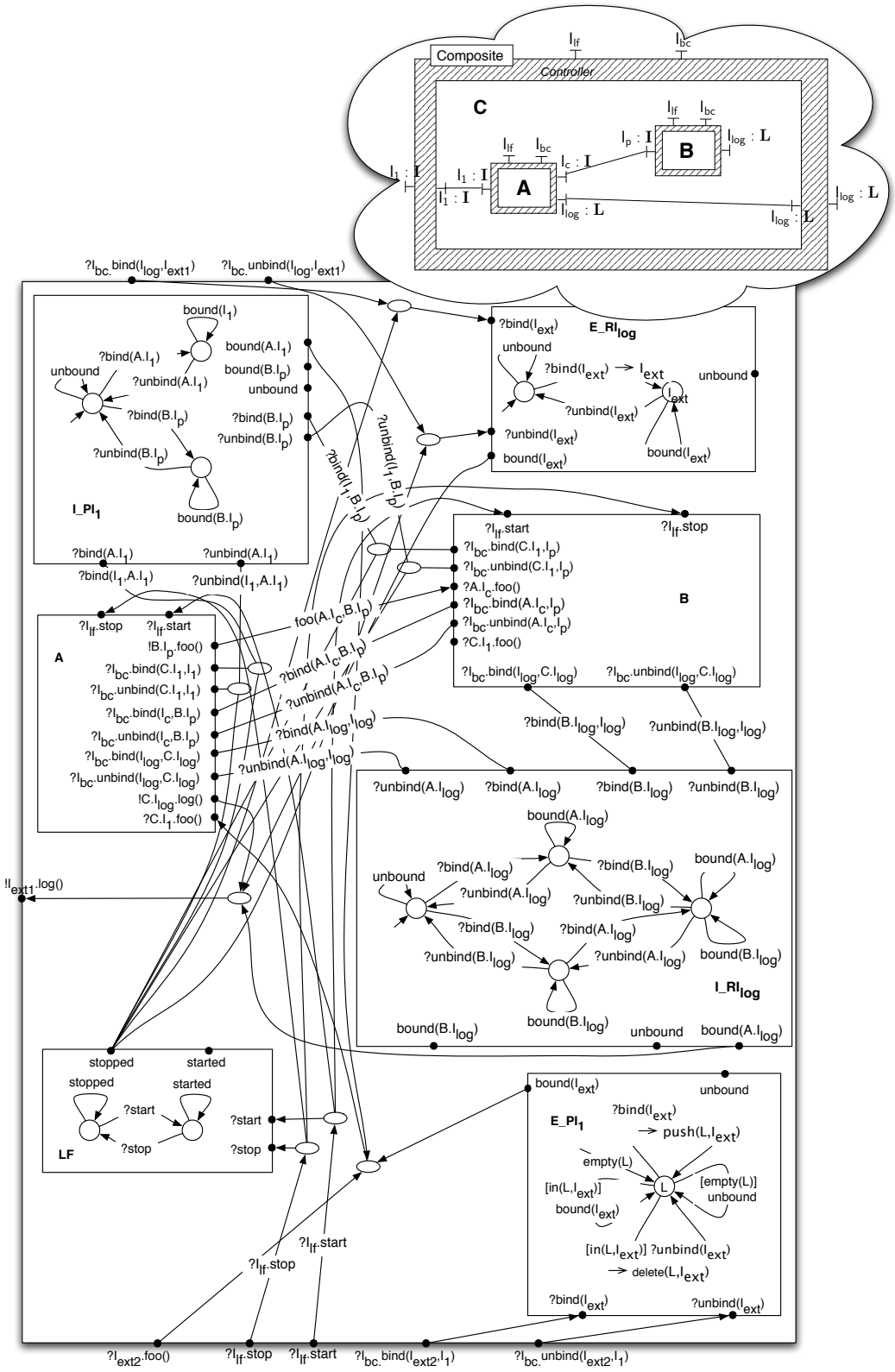


Figure 4.4: Controller of **C**

start/stop operations are hierarchical, they appear twice, both as internal and as ex-

ternal control operations. In Figure 4.4 the boxes **A** and **B** correspond to the *grey box* behaviour of **A** and **B** respectively.

Similarly to the primitive components, we can see in the figure some constraints in the control operations, such as that the binding between the internal interface I_1 of **C** and the external provides interface I_1 of **A**, encoded by $?\mathsf{bind}(\mathsf{I}_1, \mathsf{A}.\mathsf{I}_1)$ is possible only when the composite **C** is stopped. We also see in the figure an edge for the functional calls between the sub-components **A** and **B** named as $\mathsf{foo}(\mathsf{A}.\mathsf{I}_\mathsf{c}, \mathsf{B}.\mathsf{I}_\mathsf{p})$; by default this call is hidden to the upper levels of hierarchy since it is an internal action of **C**, but we chose to keep it visible. Remember that the final user can specify the internal actions he wants to observe, which will remain visible to the upper levels of hierarchy. Thus allowing the user to prove temporal properties involving those actions.

4.3.3 Detecting Errors

We can introduce in our model the detection of common sense errors (undesired behaviours) introduced in section 4.2. For instance, by triggering an $\mathsf{ERROR\_UNBOUND\_ERI}(\mathsf{A}.\mathsf{I}_\mathsf{c})$ message upon a call to the operations of the interface I_c when it is unbound, we can detect the erroneous uses of the I_c interface. This is shown in Figure 4.5.

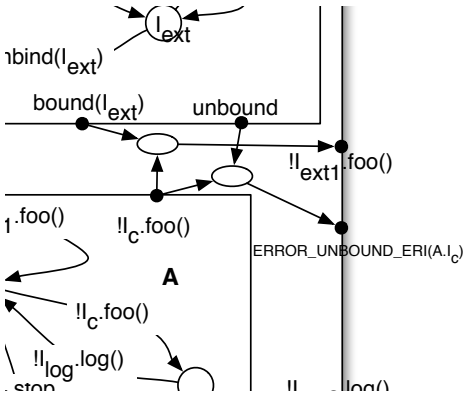


Figure 4.5: Zoom into the **A** controller detecting errors

In addition to common sense errors, others undesired behaviours are directly or intrinsically defined in the Fractal specification. In order to keep simplicity and clarity during our guided example, we will consider only the error consisting in calling an operation on an unbound interface.

4.3.4 General purpose Controller

The principles exposed for the example in the previous section are applied here in a systematic way: we have defined a *general purpose Controller*, that will be instantiated for each component in each level of hierarchy in the system, using the information available in the components ADL specification. Then the LTSs will be computed in a bottom-up fashion. The general purpose controller is shown in Figure 4.6.

To benefit from the compositional properties of our models, we define this construction in the context of a given temporal logic formula, or more generally for a given

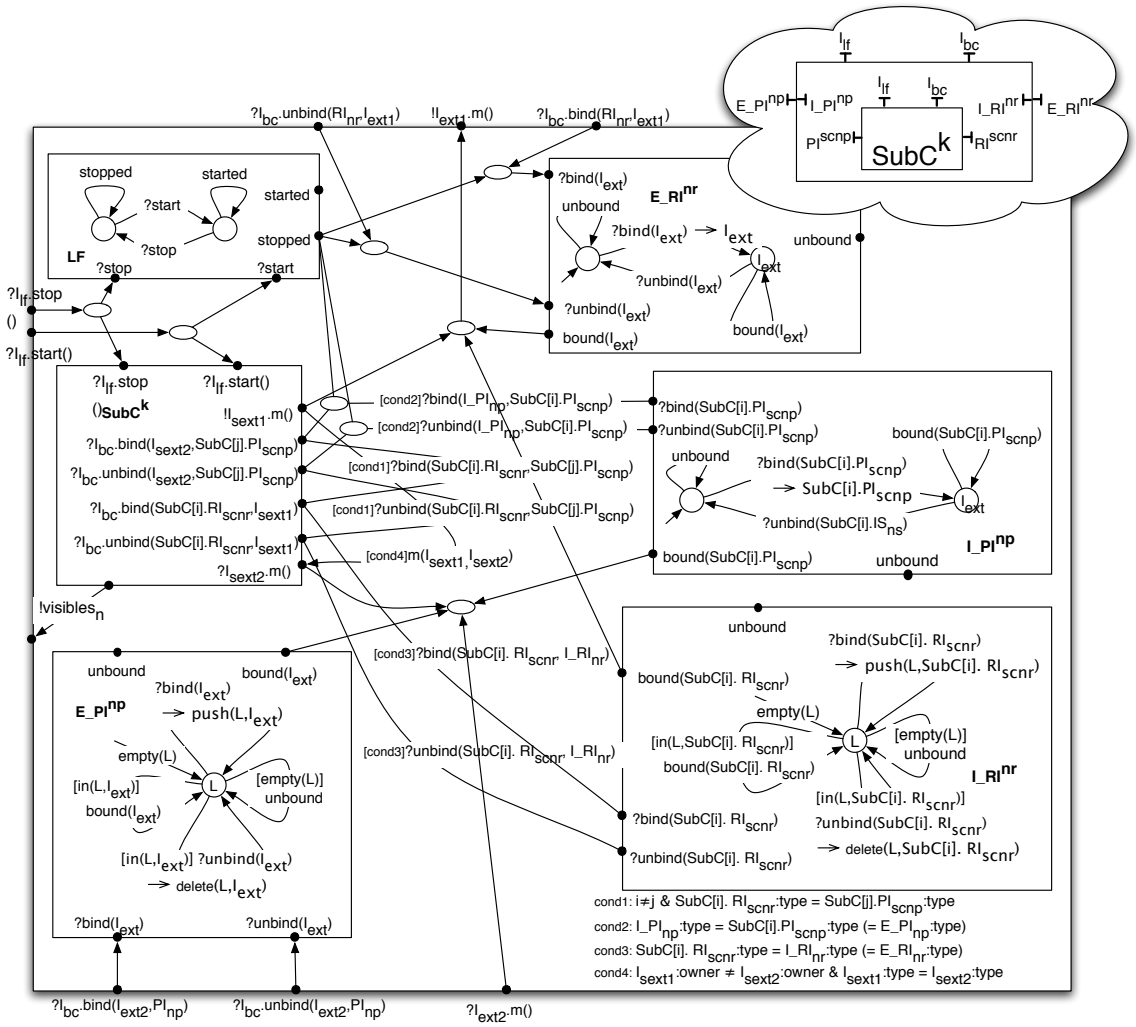


Figure 4.6: General purpose Controller

set of actions that the user wants to observe. Then we shall consider automata for a given family of *hidden actions* (renamed as τ actions), or conversely for a given family of *visible actions* (all others are hidden), minimised by weak bisimulation at each step of the construction. Note that the size of the system at a given level only depends on the complexity of this level of hierarchy (and of the actions the user wants to observe), not on the complexity of the lower levels.

In particular, specific models can be constructed to focus on the detection of some classes of errors (common sense or user defined).

In the general purpose Controller shown in Figure 4.6, we have a finite number k of sub-component automata (**SubC$^{\mathbf{k}}$**), a life-cycle automaton (**LF**), a finite number np of external (**E_PI$^{\mathbf{np}}$**) and internal (**I_PI$^{\mathbf{np}}$**) provides interface automata, and a finite number nr of external (**E_RI$^{\mathbf{nr}}$**) and internal (**I_RI$^{\mathbf{nr}}$**) requires interface automata.

To obtain the *Controller* for a component (primitive or composite), we specialise the general controller, using the sub-components and interfaces that the component ADL defines. For instance, for the composite component **C**, the set {**SubC$^{\mathbf{k}}$**} will be replaced by the networks representing the sub-components **A** and **B**. Please remark that this

specialisation only fixes the set of sub-components and internal/external interfaces, so the resulting pNet is still parameterized, and its actions contain variables for value-passing and for reference-passing.

For a primitive component, the set $\{\mathbf{SubC^k}\}$ is reduced to a single automaton which encodes its functional behaviour; its set of internal interfaces ($\{\mathbf{I\_PI^{np}}\}$ and $\{\mathbf{I\_RI^{nr}}\}$) is empty. The functional behaviour automaton encodes calls and receptions of methods on the component interfaces (in addition to internal actions).

4.3.5 Deployment and Static Automaton

Now we want to compute the full behavioural model at each level, but taking into account the *deployment phase* defined earlier.

The deployment is defined by the user; e.g. in Fractal, the bindings for the sub-components of a composite, can be given using its ADL. The deployment automaton is build from a sequence of internal control operations of the composite, possibly interleaved with functional, visible and errors actions, and terminated with a distinguished successful action $\surd$ (where in addition we interleave the external control operations).

For a component $\mathbf{C}$ (including the full application itself), let us call $\mathbf{O}_I$ the set of its internal control operations and $\mathbf{O}_E$ the set of its external control operations. Then we define the set $\mathbf{O_D} = \mathbf{O}_I \cup \mathbf{O}_E$. The deployment of $\mathbf{C}$ in our example (Figure 4.1) is shown in Figure 4.7.

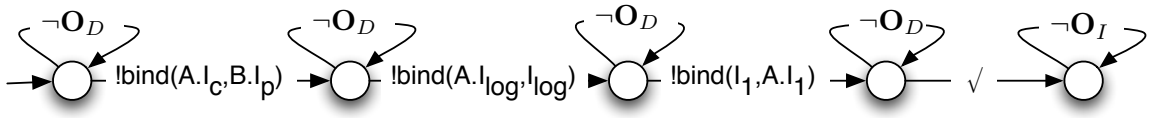


Figure 4.7: Deployment automaton for $\mathbf{C}$

We compute the synchronisation product of the deployment automaton with the component's controller (once instantiated). We name the resulting automaton of this product (modulo minimisation by weak bisimulation) as the *static automaton* of the component. The static automaton of a component intuitively corresponds to its black-box behaviour after successful deployment, hiding the internal operations (except those chosen to be visible) and forbidding any further internal reconfiguration actions. The static automaton encodes as well the behaviour of this sub-component when computing the behaviour at the next level of hierarchy.

Since the deployment automata of the primitive components are reduced to a single ($\surd$) action, their static automata are equivalent to their controller automata.

As an example, Figure 4.8 is the Static automaton of component $\mathbf{C}$. It includes only external binding operations (between $\mathbf{C}$, **System**, and **Logger**) such as $?\mathsf{I_{bc}}.\mathsf{bind}(\mathsf{I_{log}}, \mathsf{Logger}.\mathsf{I_{log}})$, functional actions of $\mathbf{C}$ such as $!\mathsf{Logger}.\mathsf{I_{log}}.\mathsf{log}()$, and errors relative to the external bindings of $\mathbf{C}$ ($\mathsf{ERROR\_UNBOUND\_ERI}(\mathsf{C}.\mathsf{I_{log}})$).

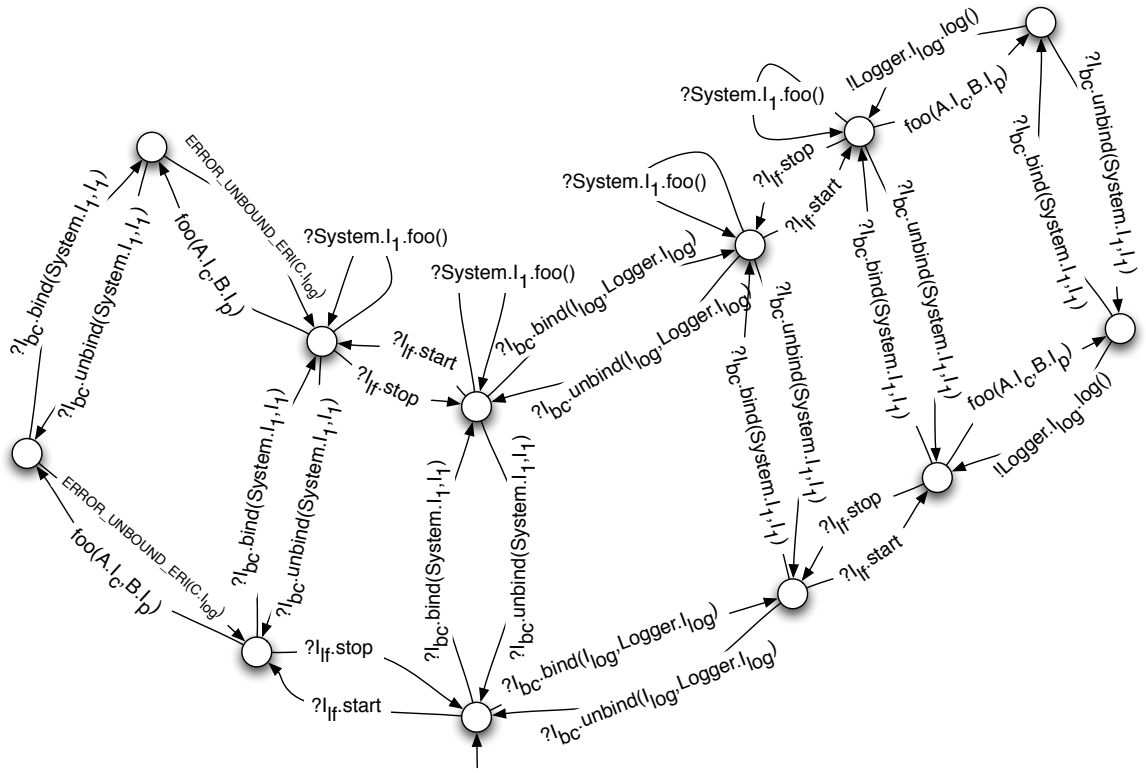


Figure 4.8: Static automaton for **C**

4.4 Properties

Having a behavioural model for a component system enables to prove temporal properties about its behaviour. Since we introduce both functional and non-functional behaviours in our model, those temporal properties can be applied to the different phases of the component's life time. Bellow we introduce a temporal classification of the properties as well as some verifications involving our example.

4.4.1 Species of Temporal Properties

All the temporal properties (that do not involve a structural reconfiguration) can be expressed and verified directly on the controller automaton of a component, or from the whole application. Yet, it is possible to define classes of properties that can be checked on smaller systems, avoiding to build the global state-space. Here we identify abstractions and tools allowing to verify some specific categories of properties.

Deployment

The interplay between the building of all components of the application, and their start operations (that are usually applied recursively after building) may be quite complex and error-prone. So it may be useful for the developer to check, independently, that deployment (possibly without start such as in Figure 4.7) of a component succeeds, and that the global deployment, including start operations, is also successful. This

will be checked on the synchronisation of the component controllers with their respective deployment automata, but leaving the successful synchronisations of the control operations visible.

Functional behaviour

A functional property is a property concerning only functional actions, or more precisely properties of a system after correct deployment, on a system in which we forbid any subsequent control action. This kind of formulas can be model-checked on a controller automaton for which we already have proved correct deployment, and in which we build only the relevant part of the behaviour, either by an ad-hoc construction algorithm (this is the static automaton with the external control operations pruned), or using on-the-fly techniques.

Functional behaviour properties are useful for component systems that do not perform any reconfiguration or for which non-functional actions have a transparent behaviour regarding functional aspects, i.e. non-functional actions commute with functional ones.

Non-structural Reconfiguration

Non-structural reconfiguration, i.e. involving only bind, unbind, start and stop operations, can be dealt directly on the controller automaton. However, the interleaving between functional and non-functional actions may have consequences on the state of the system ; we cannot provide any general abstraction fitting with this case that could reduce the complexity of the model construction for this class of properties.

Formulas involving non-structural reconfiguration are verified on the synchronisation product of the controller automaton with the deployment, but where we allow after the $\surd$ action, the interleaving of both internal and external further control operations.

Structural Transformations

Remove, *add* and *update* are the main control operations that modify the content of a composite. The first remark is that there is no hope to encode all possible future transformations in the model. Thus the method we propose address the problem of checking the safety of a structural transformation "before" applying it; typically before insertion, in an already running application, of a new component whose behavioural specification is known.

Technically, *add* and *remove* operations change the dimension of the enclosing Net, so they cannot be modelled as transducer transitions. Instead we model the structural reconfiguration operations as functions transforming the whole hierarchical model of the application ; each elementary structural change affects a single Net or LTS in the model.

Update could be expressed as a sequence *unbind*;remove;add;bind**, but this would lead both to less efficient implementations and to more complex model constructions and proofs: we are interested in expressing full sequences of reconfigurations, that preserve properties of the system, while elementary reconfigurations usually don't.

The main difficulty with structural reconfigurations is that one wants to keep the rest of the system in the same state. A large application should not be stopped when updating or adding a specific sub-component, and the state of a replaced component itself should be preserved whenever possible. The framework ensures minimum conditions before replacements (in terms of stopped/unbound state), but we have to assume that the developer will specify which data from the replaced components are to be saved, and how this data will be mapped in the new component.

A way to deal with this tree transformation and state transfer using our formalism can be the following sequence of steps :

- build a new controller with the replacement of the transformed part in the component; call $\mathcal{S}'$ this new controller;
- define a mapping between actions in the original and the new controller, based on a user-defined mapping between the action names and parameters in the replaced component;
- identify the set $\mathcal{T}$ of states on the initial controller where the transformation is possible;
- build the synchronised product of both old and new controller, using the mapping of old to new actions, and adding in each state of $\mathcal{T}$ a transition $\xrightarrow{t}$ encoding the transformation with target states in $\mathcal{S}'$.
- The automaton resulting from this synchronisation product (containing the transformation actions) becomes the component's controller which can be composed with the automaton, encoding the structural reconfiguration, to generate the static automaton and prove properties.

This approach, at the time of this thesis, is a work in progress. It needs more analysis and testing to better know it limits, features and drawbacks. Nonetheless, we have used it in our example for replacing the component **B** as described in the proving properties section (section 4.4.2).

4.4.2 Proving Properties

In this section we introduce how temporal properties can be verified using our approach. In this chapter we chose to use the Action-based Computation Tree Logics (ACTL, see e.g. [60]), while in the next chapter we will use a more expressive temporal logics named regular alternation free μ-calculus [129].

The choice to use abstract automaton in the previous chapter, ACTL in this chapter and regular μ-calculus in the next chapter to encode temporal properties, is for showing empirically that different temporal logics may be used to verify properties within our models (in some cases, as in chapter 3, because expressiveness needs too)

Deployment

The properties concerning deployment are verified in the automaton defined for deployment in section 4.4.1, i.e. on the synchronisation of the component controller with

its deployment automata, but leaving the successful synchronisations of the control operations (deployment operations) visible.

We want at first to verify that the deployment for a component is always successful. This is done by proving the ACTL formula (all paths lead to success):

$$\mathbf{A}(true_{true}\mathbf{U}_{\surd}\, true) \tag{4.1}$$

This formula is true for the deployment of **C** (Figure 4.7).

A second property we would like to verify is the absence of errors during the deployment. This is done by proving the formula (for a given set of errors $\mathbf{O}_E$):

$$\mathbf{AG}_{true}[\mathbf{O}_E]\ false \tag{4.2}$$

This property is also true for the **C**'s deployment. However, in a very reasonable scenario, let's suppose the user starts the component **C** at the end of the deployment. Under this scenario the property is not true any more (even though the deployment is possible), and the model-checking tool gives us the counter-example shown in Figure 4.9 (Note that we label successful synchronisations between the actions $?\alpha$ and $!\alpha$ as α)

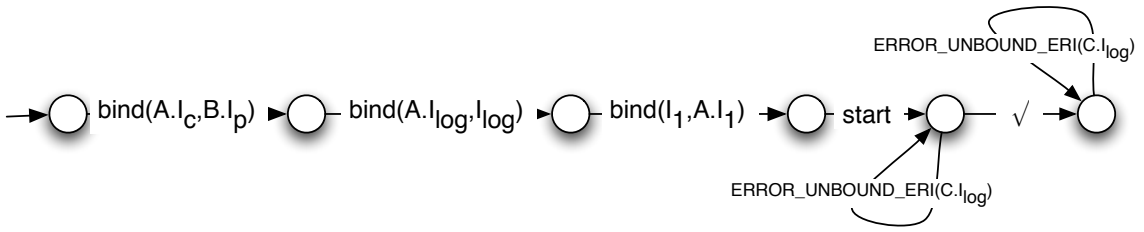


Figure 4.9: Diagnostic path

The error is because the required interface $\mathsf{C.I_{log}}$ may be used before it is bound, which in fact is true since the interface $\mathsf{I_{log}}$ of **C** will be bound at the next level of hierarchy (when deploying **System**). This example also shows us the importance of the hierarchical behaviour of start and stop in the right order.

Functional behaviour

The properties concerning pure functional behaviour are verified in the automaton defined for functional behaviour in section 4.4.1, i.e. in the static automaton with the external control operations pruned.

We would like to verify the absence of errors during a running phase, i.e. the absence of errors between the deployment and a new reconfiguration phase. We can verify the property by proving the ACTL formula for a given set of error actions $\mathbf{O}_E$:

$$\mathbf{AG}_{true}[\mathbf{O}_E]\ false \tag{4.3}$$

For instance, the proof is successful for **System**.

Another property we would like to prove (extracted for example from the user requirements) can be that every call to the function foo() in the interface $\mathsf{I_c}$ of **A** to the interface $\mathsf{I_p}$ of **B** is eventually logged in **Logger**.

This inevitability property is checked by verifying the ACTL formula:

$$\mathbf{AG}_{\neg \mathsf{foo(A.I_c,B.I_p)}}[\mathsf{foo(A.I_c, B.I_p)}]\ \mathbf{A}(true_{true}\ \mathbf{U}_{\mathsf{log(C.I_{log},Logger.I_{log})}}\ true) \tag{4.4}$$

In Figure 4.10 we show the static automaton for **System** when forbidding external control operations. We consider an instantiation where **Logger** has a logging capacity of 2 ($n = 2$ in Figure 4.2).

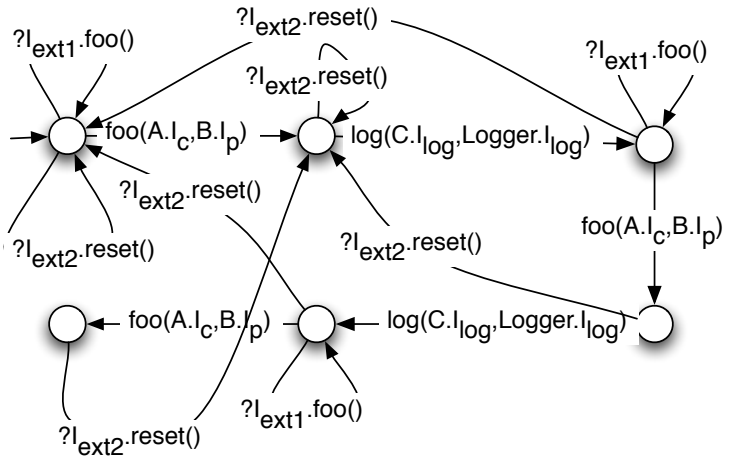


Figure 4.10: $\mathbf{S}\backslash\mathbf{O}_C$ for **System**

Formula (4.4) is false in **System** and the model-checking tool gives us the diagnostic shown in Figure 4.11.

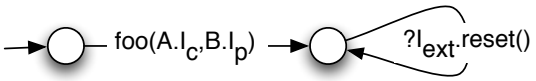


Figure 4.11: Property (4) diagnostic for **System**

This diagnostic is showing us one case where the Formula (4.4) is false because the behaviour of **System** can fall in an infinite loop of the action $?\mathsf{I_{ext}.reset}()$ after the action $\mathsf{foo(A.I_c, B.I_p)}$. Then the computation tree contains traces where the action $\mathsf{log(C.I_{log}, Logger.I_{log})}$ never happen, this is known as an unfair path.

We need to add a constraint to avoid the infinitely loop shown in Figure 4.11 by considering only fair paths (i.e where the constraint happen infinitely often) when proving the property. In practise, this is expressed in the logical formula itself. Then Formula (4.4) becomes the formula:

$$\begin{aligned}\mathbf{AG}_{\neg \mathsf{foo(A.I_c,B.I_p)}}[\mathsf{foo(A.I_c, B.I_p)}]\ \mathbf{AG}_{\neg \mathsf{log(C.I_{log},Logger.I_{log})}}\mathbf{EF}_{true}\\ < \mathsf{log(C.I_{log}, Logger.I_{log})} > true\end{aligned} \tag{4.5}$$

This formula express that after a $\mathsf{foo(A.I_c, B.I_p)}$ and while $\mathsf{log(C.I_{log}, Logger.I_{log})}$ is not reached, $\mathsf{log(C.I_{log}, Logger.I_{log})}$ is reachable in a finite number of transitions (which avoid the unfair paths).

However, we are considering the assumption that **System** is in an environment where the method reset() of its provided interface I_1 is infinitely often invoked. We should be careful about those assumption, for instance suppose that this interface is binding to a button of a graphical interface for the user: if we want to prove the correct autonomous behaviour of our system, i.e. without user intervention, then we should consider that the reset button is never pressed. We express that by adding the restriction of non-reset action ($\neg ?\mathsf{I}_{\mathsf{ext}}.\mathsf{reset}()$), then Formula (4.5) becomes:

$$\mathbf{AG}_{\neg \mathsf{foo}(\mathsf{A.I_c},\mathsf{B.I_p})}[\mathsf{foo}(\mathsf{A.I_c},\mathsf{B.I_p})]\ \mathbf{AG}_{\neg \mathsf{log}(\mathsf{C.I_{log}},\mathsf{Logger.I_{log}})}\mathbf{EF}_{\neg ?\mathsf{I_{ext}}.\mathsf{reset}()} < \mathsf{log}(\mathsf{C.I_{log}},\mathsf{Logger.I_{log}}) > true \tag{4.6}$$

Formula (4.6) is false for **System** and the tool gives us the diagnostic shown in Figure 4.12. The figure shows us the presence of a deadlock because **Logger** is full. To empty **Logger** it should be reset through the method call reset() in its I_1 interface, and in conclusion, **System** can not autonomously behave correctly, it needs the user to take part.

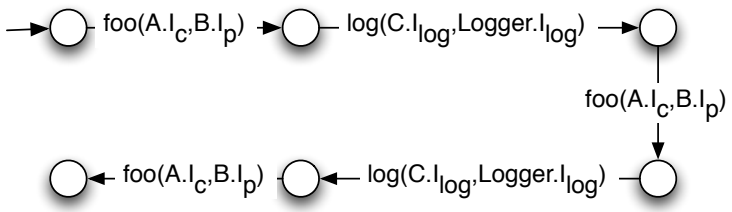


Figure 4.12: Property (4.6) diagnostic for **System**

We have shown in this section how we prove behavioural properties of a component or an application, where properties range from reachability of an error action to intricate temporal ordering of actions, including fairness properties. More research is needed for giving to final users a more accessible language for expressing those properties, for example an extension of so-called "specification patterns" [67] with specific constructs for component management.

Non-structural Reconfiguration

As we state in section 4.4.1, properties concerning non-structural reconfiguration are verified on the synchronisation product of the controller automaton and the deployment, in which we allow the interleaving of both internal and external control operations after the successful deployment.

We would like to prove some preservation of properties when doing non-structural reconfigurations. For instance, we can prove that Property (4.5) is preserved for any non-structural reconfiguration that does not involve the interface $\mathsf{I}_{\mathsf{log}}$ in **System**. Let $\mathbf{O}_{log}$ be the binding operations involving $\mathsf{I}_{\mathsf{log}}$, i.e. $\mathbf{O}_{log} = \{?\mathsf{bind}(\mathsf{C.I_{log}},\mathsf{Logger.I_{log}}), ?\mathsf{unbind}(\mathsf{C.I_{log}},\mathsf{Logger.I_{log}})\}$. Then this property of preservation is successful verified on **System** by checking the ACTL formula:

$$[\surd]\mathbf{AG}_{\neg \mathbf{O}_C}[\mathsf{foo}(\mathsf{A.I_c}, \mathsf{B.I_p})]\ \mathbf{AG}_{\neg \mathsf{log}(\mathsf{C.I_{log}}, \mathsf{Logger.I_{log}})}\mathbf{EF}_{\neg \mathbf{O}_{log}} < \mathsf{log}(\mathsf{C.I_{log}}, \mathsf{Logger.I_{log}}) > true \quad (4.7)$$

Structural Transformation

Suppose we do, during the application running-phase, an update of the sub-component **B** in **C** by a component **B2**. **B2** has a similar behaviour than **B**, but in addition it logs the calls to its $\mathsf{I_p}$ interface using its $\mathsf{I_{log}}$ interface; its automaton is shown in Figure 4.13. If we build this new system (using the method described in section 4.4.1), and try to prove again Formula (4.3), it appears to be false. Tool gives us a path, shown in Figure 4.14, containing the action ERROR_UNBOUND_ERI(B2.log).

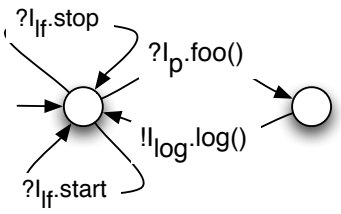


Figure 4.13: **B2** behaviour

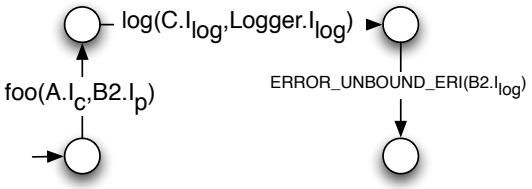


Figure 4.14: Formula (4.3) diagnostic when using **B2**

This is because in the initial deployment of the system, we did not bind the interface $\mathsf{I_{log}}$ of **B**. Since **B** did not use its interface $\mathsf{I_{log}}$, the composition did not produce an undesired behaviour. However, the new **B2** uses its $\mathsf{I_{log}}$ interface, and so it produces the error. Therefore the update of **B** by **B2** should be followed by a binding of its $\mathsf{I_{log}}$ interface. This example, likely to happen in real systems, shows the necessity of formal verification tools for checking reconfiguration requirements.

If we bind the interface $\mathsf{I_{log}}$ of **B2** to the internal interface $\mathsf{I_{log}}$ of **C**, after the update and before starting the system, then the property is preserved.

4.5 Tools

The user may describe the system topology using the Fractal Architecture Definition Language (ADL). The Fractal ADL is an open and extensible language to define component architectures for the Fractal component model, which is itself open and extensible. Fractive uses the default concrete syntax, based on XML, provided by Fractal. The XML file describing the example's **System** is shown in Figure 4.15.

The XML description shown in the figure specifies that the system is composed of the composite **C** (line 6), itself described in a separate file (`components/C.fractal`), and the primitive **Logger**, which implementation is the Java class `components.LoggerImpl` (line 17). **C** requires an interface named `log` of type `components.LogInterface` (lines 8,9) and provides an interface named `I1` of type `components.I1Interface` (lines 10,11). **Logger** provides an interface named `log` of type `components.LogInterface` (lines 15,16)

System.fractal

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE .... >
3
4  <definition name="components.System">
5
6   <component name="C"
7       definition="components.C">
8    <interface name="log" role="client"
9        signature="components.LogInterface"/>
10   <interface name="l1" role="server"
11       signature="components.I1Interface"/>
12  </component>
13
14  <component name="Logger">
15   <interface name="log" role="server"
16       signature="components.LogInterface"/>
17   <content class="components.LoggerImpl">
18    <behaviour file="LoggerBehav"
19         format="Aldebaran"/>
20   </content>
21  </component>
22
23  <binding client="C.log"
24       server="Logger.log"/>
25 </definition>
```

Figure 4.15: System ADL

Component	Controller	Static Aut.
A	24/99	24/91
B	16/98	16/90
Logger	4/16	4/14
C	432/2168	12/39
System	36/151	6/19
B2	24/107	24/99
C {update(B,B2)}	1786/7082	20/58

Figure 4.16: Example of automata sizes (states/transitions) of the example

We propose to extend the content tag of primitives to include their functional behavioural specification. For instance in Figure 4.15 we set that the behaviour of the class `components.LoggerImpl` is defined in a file named `LoggerBehav` with format FC2 Parameterized (lines 18, 19).

Finally, at lines 23, 24 in Figure 4.15, the ADL defines that at the initial deployment, the interface `log` of **C** should be bound to the interface `log` of **Logger**.

We developed a prototype tool in Java which takes the system ADL as input, and the functional behaviour of primitives to automatically generate the models described in this chapter. We use the CADP [81] tool-set to do the synchronisation product and the model-checking of formulas. From the ADL description, our prototype tool generates, once instanced, the synchronisation product in Exp-V2 format (and ASCII list of synchronisation vectors). The automaton describing the functional behaviour of primitives is taken directly from its file (line 18 in Figure 4.15). Our tool also generates a script to build the system (SVL [108, 79] script from CADP). Finally, the proofs are verified using evaluator, an on-the-fly model checking tool included in CADP. Table 4.16 shows some results for the generated automata in our example; the CADP tool-set allows us to handle systems with as much as 100 millions states at each construction level.

4.6 Conclusions

In this chapter we have provided methods and tools allowing the user to prove the correctness of the behaviour of synchronous hierarchical components. One of our main contributions is the specification of non-functional aspects behaviour, and the hierarchical building of LTSs modelling the system of components behaviour. Our approach

relies on the definition of a generic controller allowing (once instantiated) to encode the whole behaviour of any component except non-structural reconfiguration. Then a component behaviour is obtained by synchronisation product of the LTSs expressing the behaviour of its content and the control behaviour associated to its interfaces. Structural (dynamic) reconfiguration is handled by a LTS transformation. The tools provided to the user include:

- a controller automaton allowing to prove general properties on the behaviour of a component provided no structural reconfiguration is considered;
- error detection: firing of error messages upon common sense errors can automatically be added; then, for example, the user may prove the absence of such messages in order to assert the correctness of the application;
- modelling of structural reconfigurations as transformations of the application model.

We have developed a tool in Java that automatically and incrementally generates the synchronisation files for a component system from its description, and we use the CADP[81] tool set to calculate the synchronisation product, minimise the systems, and finally model-check the formulas.

Chapter 5

Distributed Hierarchical Components Behaviour

5.1 Introduction

In the preceding chapter we have introduced our approach for modelling the behaviour of (synchronous) hierarchical component systems built using the Fractal component model, and we have shown how several properties may be verified on such models.

The approach we proposed is to give the components behavioural specifications in the form of hierarchical synchronised transition systems. We assumed that the models for the functional behaviour of basic (primitive) components is known. Then, we automatically incorporate the non-functional behaviour within a controller built from the component description. The composite's semantic is computed as a product of the its sub-components LTSs with the composite's controller. This system can be checked against requirements expressed as a set of temporal logic formulas, or again as an LTS (defining an abstract specification).

In this chapter we extend our approach to support distributed (asynchronous) components. Dynamic aspects, e.g. reconfiguration, are particularly important when considering distributed components. In here, we propose to provide a framework for the behavioural specification and verification of *distributed*, *hierarchical*, *asynchronous*, and *dynamically reconfigurable* components.

Specifically, we focus on Fractive [29], a Fractal component model [43] implementation using the middle-ware ProActive [47, 28, 46]. In Fractive, components become active in the same way than ProActive's active objects: their membrane has a single non-preemptive control thread which serves, based on different serving policies, method requests from its unique pending queue. The requests to other components are done via a *rendez-vous* phase so there is a delivery guarantee and a order conservation of incoming calls. The responses (when relevant) are always asynchronous with replies by means of future references; their synchronisation is done by a *wait-by-necessity* mechanism.

Similar to synchronous components, we automatically incorporate the non-functional behaviour of the components, based on the component's definition, as automata being part of the synchronisation network defining the component's controller. In addition to chapter 4, we incorporate the Fractive component features by automatically adding automata encoding the queues, future responses and serving policies (in particular default

policies of serving functional versus non-functional requests based on the component life cycle status).

Once more our aim is to provide the final user with a framework for verifying the behaviour at different component phases (now distributed components) and our target user is always the application developer in charge of the component's life-time maintenance. Again, this framework should hide as much as possible the verification process complexity, and be automatic if possible.

Sections 5.2 and 5.3 present ProActive and Fractive respectively. In section 5.4 we introduce the behaviour model for Fractive components. Section 5.5 shows the automatic construction of the model based on an example. In section 5.6 we prove some properties of our example involving asynchronous features, and finally section 5.7 concludes about this chapter.

5.2 ProActive

ProActive [47, 28, 46] is a pure Java implementation of distributed active objects with asynchronous remote method calls and replies by means of future references. A distributed application built using ProActive is composed of several activities, each one having a distinguished entry point, the active object, accessible from anywhere. All the other objects of an activity (called *passive objects*) can not be referenced directly from outside. Each activity owns its own and unique service thread and the programmer decides the order in which requests are served (or not). Each activity has a pending queue where the incoming requests are dropped. Requests are asynchronous method calls addressed to the active object, and should be served by the service thread. The requests are sent using a *rendez-vous* phase so there is a guaranty of delivery and of order between incoming calls. During the rendez-vous a future (reference to the future result) is created on the sender side thus allowing asynchrony. The responses are always asynchronous; their synchronisation is done by a mechanism called *wait-by-necessity*.

Figure 5.1 shows an example consisting of two activities, each one having a single active object (entry point) and a set of passive objects.

The method calls to active objects behave as follow:

1. When an object makes a method call to an active object ($y = \mathbf{O}_B.m(\vec{x})$), the call is stored in the request queue of the called object (Q_B) and a future reference is created and returned (y references f). A future reference encodes the future return value, i.e. not yet available, of a method call to an active object.

2. At some point, the called activity (its unique thread) decides to serve the method call (`serve(`$m(\vec{x})$`)`). The request is taken from the queue and the method executed.

3. Once the method finishes, its result is updated, i.e. the future reference (f) is replaced with the concrete method result (`value of` y).

When a thread tries to access a future reference before it has been updated to the concrete real value, it is blocked until the update takes place (*wait-by-necessity* mechanism).

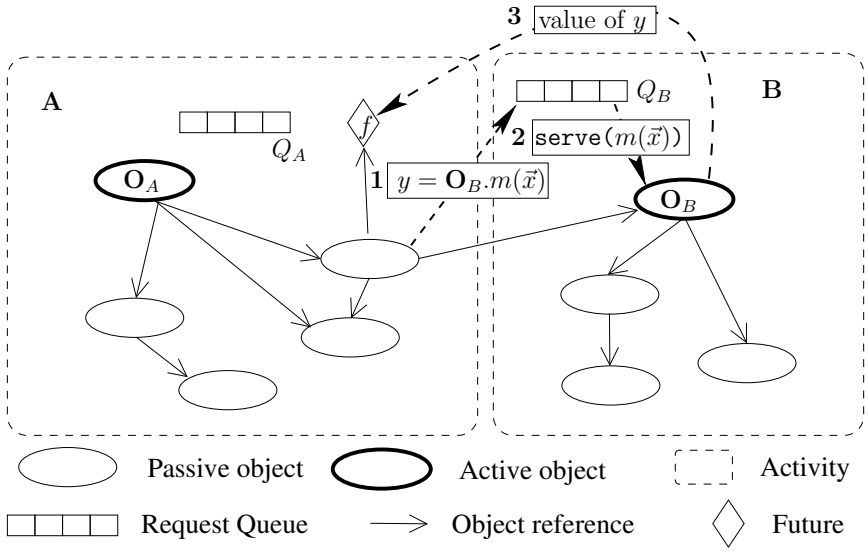


Figure 5.1: An example consisting of two activities

A developer can specify a policy for serving the requests from the active object's queue. In practise this is done by implementing the `runActive` method which is executed as soon as the activity starts. The ProActive API provides several versions of `serve` methods (such as blocking/unblocking serve, FIFO/LIFO order or based on queue filters among others). When `runActive` is not provided, the ProActive middleware implements a default FIFO policy.

The ASP calculus [46] has been defined to provide an operational semantics for the ProActive library and the aspects presented above.

5.3 Fractive

Fractive [29] is a Fractal implementation using the ProActive middle-ware. Fractive provides a component implementation having the same features than ProActive, including asynchronous calls, absence of shared memory, distribution transparency and serving request control.

As proposed in the Fractal specification, a Fractive application is made of a set of primitive components that are composed at different levels of hierarchy, each one defined by a specific composite.

5.3.1 Primitive Components

A primitive component in Fractive is made from one active object (and a set of passive ones) whose methods implement the provided interfaces operations. The client bindings mechanism are specified within the code (usually they affect the reference of a local field of the objects), i.e. the client bindings are defined by the developer.

Note that this approach allows primitive components made of several activities but only for simplicity, we can consider here that primitive components are formed of a

single active object and thus can be associated a single request queue. This directly generalises to primitives having several activities, with only one exporting interfaces.

Both, functional and non-functional requests are dropped to the request queue of the component. Fractive transmits the treatment of non-functional request to the active object. This is for separation of concerns: we do not expect the developer to take care of non-functional aspects (even if he could) but to focus on the functionality implementation of the component. In practise, a developer codes a primitive component as if were just a normal active object. This provides a re-usability feature of already coded active objects as primitive components.

The life-cycle of a primitive in Fractive behaves as follows:

1. When stopped, only non-functional requests are served.
2. Starting a primitive component means running the `RunActive` method of its active object
3. Stopping a primitive component means exiting from the `RunActive` method of its active object.

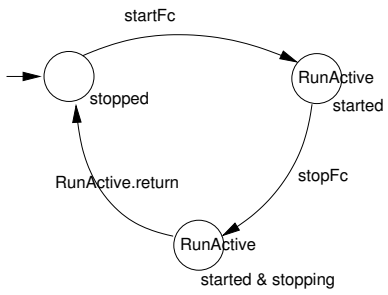


Figure 5.2: Primitive life-cycle

Since active objects are non-preemptive, the exit from the `RunActive` method can not be forced. The approach to implement 3 is to signal the component that it has been requested to stop. Then the component eventually terminates the execution of `RunActive`. This is shown in Figure 5.2.

In practise, stop requests are signalled by setting the local variable `isActive` to false. Then, the `RunActive` method should eventually end its execution.

5.3.2 Composites

A membrane and a finite set of sub-components form a composite. The membrane contains the set of external interfaces which can be bound to other components. It also contains internal interfaces which can be bound to its immediately inner sub-components external interfaces.

In Fractive the membrane of a composite is implemented as an active object, therefore a membrane contains an unique request queue and a single service thread. Each interface of the composite defines an internal and an external interface in the membrane. A Fractive composite is shown in Figure 5.3.

Method calls to external and internal interfaces are both dropped to the request queue of the composite (including functional and non-functional request). The service thread (`RunActive`) serves the methods from the queue in FIFO order but bypass functional requests whenever the composite is stopped (i.e. it serves only non-functional ones). As a consequence, a composite will not emit functional calls on its required interfaces while stopped, even when its sub-components may be active and sending requests to its internal interfaces.

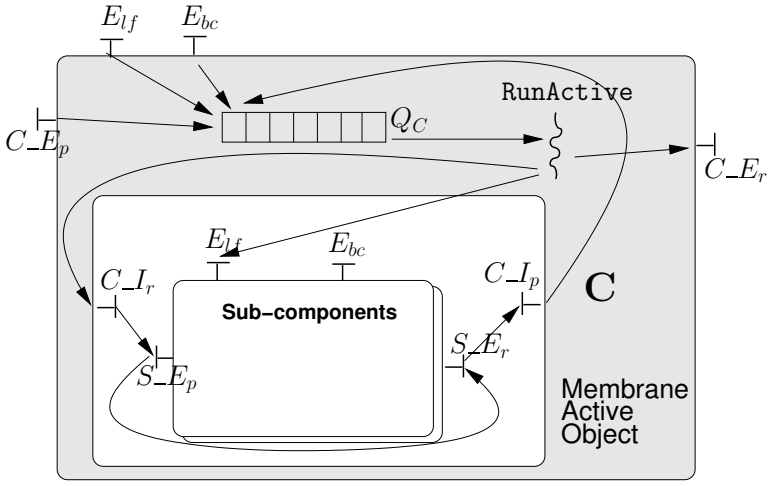


Figure 5.3: Fractive composite component

Note that we have named "required" the internal interface corresponding the an external "provided" and vice versa. This is the membrane point of view. Serving a functional request (when the composite is started) on an internal provided interface means forwarding the call to the corresponding external required interface of the composite. Serving a functional request on an external provided interface consists in forwarding the call to the corresponding internal required interface of the composite.

5.3.3 Choices Made With Respect to Fractal

Some features are left unspecified in the Fractal definition, and may be set by a particular Fractal implementation, or left to be specified at user level. Similar to the previous chapter, Fractive makes the following choices:

1. the start/stop operations are recursive, i.e. they affect the component and each one of its sub-components, in a top-down order, and
2. functional operations cannot fire control operations.

5.4 Building the component's model behaviour

In chapter 4, we have described our approach to construct behavioural models for Fractal hierarchical components, in a synchronous context. We suppose that the functional behaviour of primitive components can be derived from automatic analysis of source code or expressed by the developer. Then we automatically incorporate the non-functional behaviour: for each internal and external interface of the component, we define a LTS encoding the interface's behaviour (its binding operations) plus a LTS encoding the life-cycle behaviour of the component (start/stop operations). The semantics of a component was then computed as the synchronisation product of the non-functional LTSs and the LTSs of its sub-components (or the functional LTS in case of a primitive component). We named this synchronisation product the component's *controller*

The construction was done automatically in a bottom-up hierarchical fashion from the component system description (given through an ADL). At each level of hierarchy, i.e. for each composite, its deployment phase was applied. The deployment is a sequence of non-functional operations, expressed by an automaton, ending with a distinguished successful action $\surd$. Then, a successful deployment may be verified by the reachability analysis of the $\surd$ action on the synchronisation product of the component's controller and its deployment.

We also defined the *static* automaton of a component as being the synchronisation product of the *controller* automaton with the deployment automaton, hiding internal actions, forbidding any further reconfiguration (internal control actions), and minimised modulo weak bisimulation. This static automaton encodes the behaviour of this sub-component when computing the controller at the next level in the hierarchy.

We use a similar approach for Fractive components, but extended to include its features introduced by ProActive. In the following we detail the models for Fractive components.

5.4.1 Building models for Fractive components

A graphical view of the model for Fractive at a given level of the hierarchy is shown is Figure 5.4. The doted box in the figure visually separates what belongs to the content of the component (inside the box) and what belongs to the controller of the component.

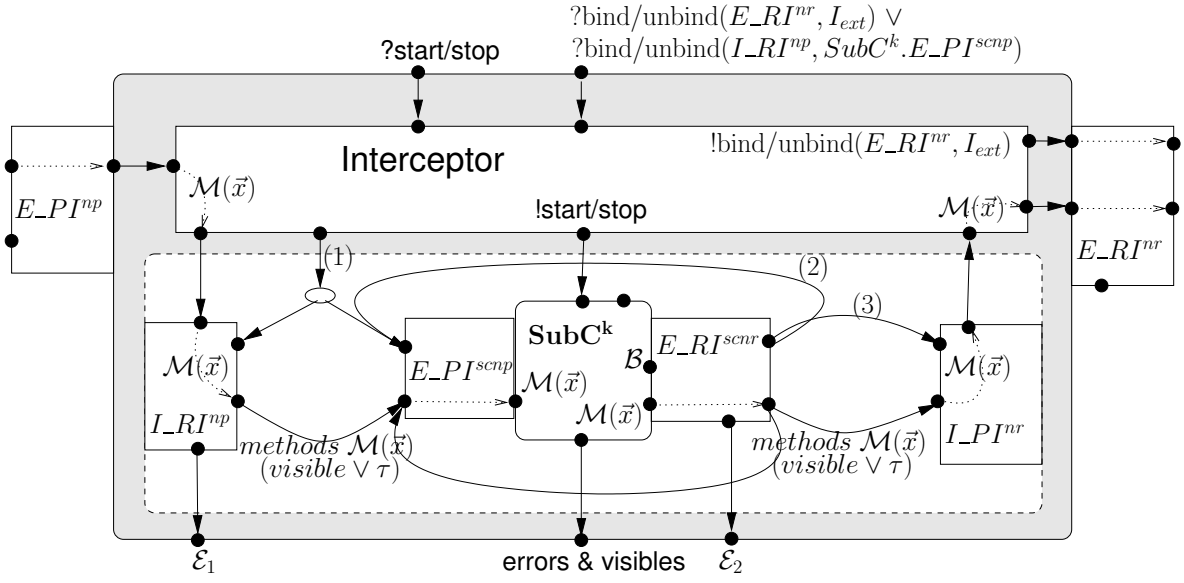


Figure 5.4: Component behaviour model

In the same spirit as in chapter 4, we observe in Figure 5.4 a box encoding the behaviour of each internal (I_PI and I_RI) and external (E_PI and E_RI) interface. The behaviour of the interfaces includes functional actions (method calls $\mathcal{M}(\overrightarrow{x})$), non-functional actions and the detection of errors ($\mathcal{E}_1$ and $\mathcal{E}_2$) such as the use of an un-

bounded interface. These errors are made visible at the higher level of hierarchy. The doted edges inside the boxes indicate a causality relation induced by the data flow through the box.

Figure 5.5 shows the detail for I_RI^{np} and LF. We observe in I_RI^{np} that method calls to this required interface are only possible when it is bound; if not, an error is signalled. We do not go deeper in the detail of the interfaces and life-cycle boxes, which have similar structure as in chapter 4.

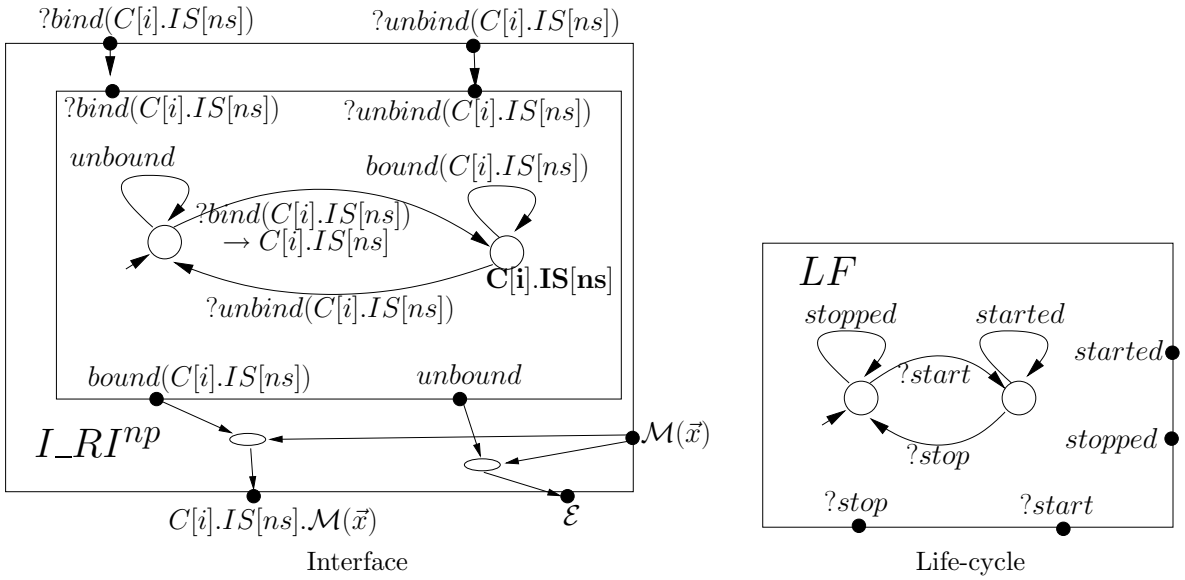


Figure 5.5: Examples of Internal Required Interface and Life-Cycle automata

One technical choice we have made with respect to chapter 4, is to bring the behaviour of external interfaces outside the *controller* definition of a component. In Figure 5.4 they are attached at the next level of hierarchy as we can see in both sides of $\mathbf{SubC^k}$. Indeed we only know the possible bindings of external interfaces of a component at the next level of the hierarchy: for a given interface, the set of interfaces which can be bound to it depends on the environment of the component to which it belongs. In chapter 4, we made the opposite choice, including the external interfaces behaviour inside the composite controller; as a consequence, we had to reach one level up in the hierarchy to be able to instantiate the *controller* automaton, and therefore to calculate the synchronisation product.

By adding the external interface automaton of a component in the next level of hierarchy, we can now calculate the *controller* automaton of a component before knowing its environment (the *controller* does not have actions of the external interfaces). Thus, all the properties not involving external interfaces can be verified in a fully compositional manner. Additionally, the internal control actions, which are used when doing the deployment, can now easily be characterised as the set of control actions in the alphabet of the *controller* automaton.

5.4.2 Adding Interceptors

In chapter 4 the controller of a component is limited to reconfiguration. In order to introduce more control capabilities, (e.g., interception and treatment of functional requests) we introduce an interceptor in the membrane of the composites that can intercept method calls between the composite environment and its subcomponents. This is shown graphically in Figure 5.4.

5.4.3 Modelling the Primitives

In [21] we have shown how to build the behavioural model of an active object by analysis of its source code. We also modelled the ProActive mechanisms for asynchronous requests management. Figure 5.6 shows the topology of the part of this synchronisation network corresponding to a request addressed to a remote activity, its processing and the return of its result.

As a Fractive primitive component contains a single active object, we shall use the same structure for their modelling, adding only the features necessary for managing non-functional aspects.

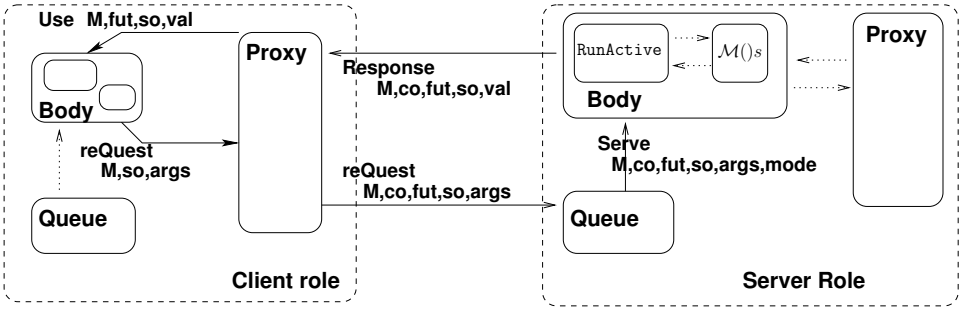


Figure 5.6: Communication Between two Activities

In the model (Figure 5.6), a method call to a remote activity goes through a proxy, that locally creates a "future" object (it creates a future for each call), while the request goes to the remote request queue. The request arguments include a reference to the future. It also contains a deep copy of the method's arguments, because there is no sharing between remote activities. Later, the request may eventually be served, and its result value will be sent back to the future reference.

The **Body** box in the figure is itself a synchronisation network made from the synchronisation product of the `RunActive` method's LTS with the behaviour of each method as described in [21]. The **Queue** box, additionally to methods reception, encodes the different primitives to serve the methods in the queue provided in the ProActive API, and used in its interactions with the body.

The case of Fractive primitive components is similar to the model shown in Figure 5.6. Each method call on each interface is encoded by the action of making the call (**reQuest** in the figure) and its response by the update of the future value (**Response** in the figure). The use of a future, which blocks the execution until the future value is available, is encoded by the action **Use** in the figure.

Our approach is to enrich the behaviour of the active object by adding two extra boxes, **LF** and **NewServe** as shown in Figure 5.7.

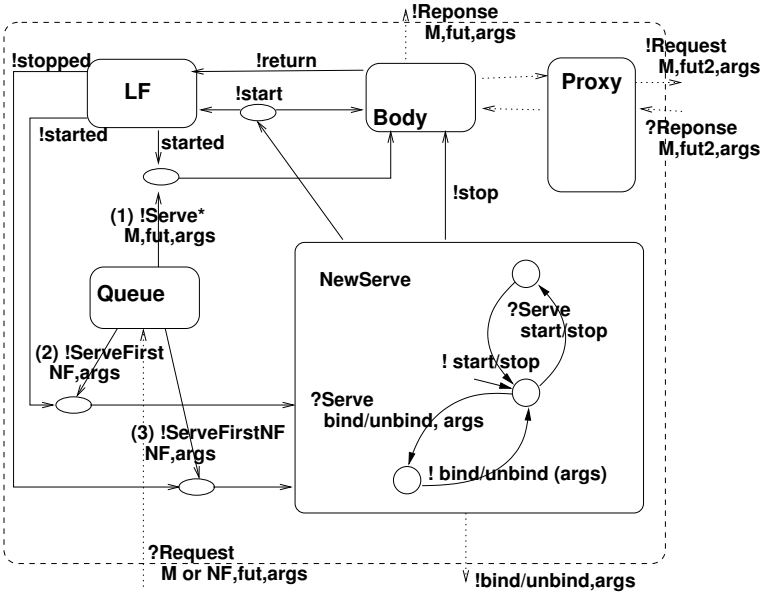


Figure 5.7: Behaviour model for a Fractive primitive

Similar to chapter 4, we assume that the functional behaviour of the component, i.e. the **body** in the figure, is known. Everything else, including the behaviour of the composite, is generated automatically based on the component's definition.

NewServe implements the treatment of non-functional requests. "start" fires the `RunActive` method (transition) in **body**. "stop" triggers the `!stop` synchronisation with **body** (Figure5.7). This synchronisation should eventually lead to the termination of the `RunActive` method (`!return` synchronisation). In the Fractive implementation, this is done through setting the state variable `isActive` to false, which should eventually cause the `RunActive` method to finish. The component is assumed to be stopped only once the `RunActive` method has finished.

In Figure 5.7 the action (1) will serve the first functional method that agree with the `Serve` API primitive utilised (for instance FIFO, LIFO, FIFO with a method name match condition, etc.), (2) will serve a non-functional method only if it is in the head of the queue and (3) will serve only non-functional methods in FIFO order bypassing the functional methods present in the queue.

When a functional request is ready to be taken from the queue and the component is started, it is served by the **body** of the active object. A primitive component does not contain internal interfaces, thus the binding operations are possible only on its external interfaces (which now are not included in the *controller*). In Figure 5.7 the binding operation signals are exported ($\mathcal{B}$ in the subcomponent of Figure 5.4) to the next level of hierarchy.

5.4.4 Modelling the Composites

As we mention in section 5.3, a composite membrane in Fractive is an active object. When started, it forwards method calls between internal and external functional interfaces. When stopped it serves only non-functional requests.

The behaviour of a composite membrane is the synchronisation network shown in Figure 5.8.

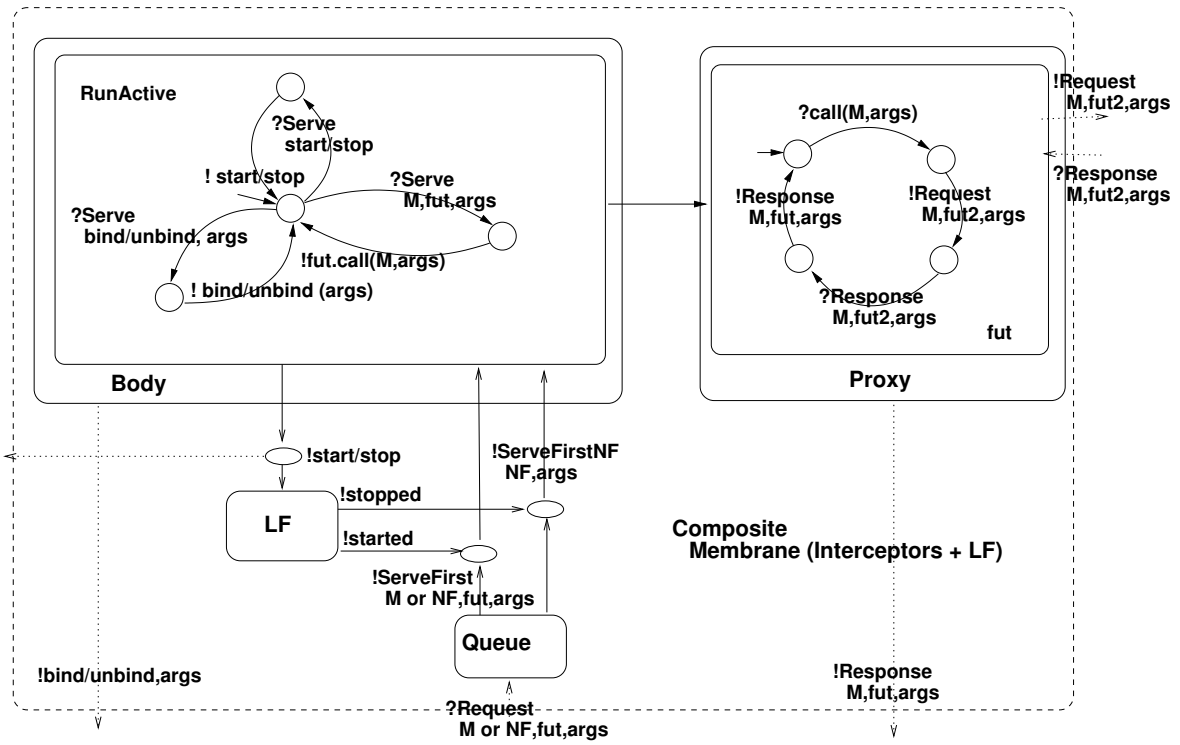


Figure 5.8: Behaviour of a composite membrane

The membrane active object is created based on the composite description (usually given by the ADL). This membrane takes the role of the interceptor introduced in the general model (Figure 5.4). Both functional and non-functional calls are drop in the queue of the composite (5). When started, the membrane indifferently serves functional or non-functional methods in FIFO order (1). When stopped, similar to primitives, it serves non-functional methods in FIFO order bypassing the functional methods present in the queue (2).

When serving a functional call (3), a method call is done to the corresponding interface and a future object is created (4). Note that the future references (**proxy** box in Fig. 5.8) are updated in a chain following the membranes from the primitive serving the method to the caller primitive. Since the method calls include the reference of the future in the arguments, future updates can be addressed directly to the caller immediately before in the chain. Consequently, like in the implementation, the future update is not affected because of rebinding or the life-cycle status of the components.

5.4.5 Building the Global Behaviour

The next step is to build a global model for the component, for the various pieces presented in the previous sections. This "global" behaviour construction is compositional in the sense that we only need study one level of hierarchy at a time, relying on some abstraction of the subcomponents behaviours in this process. In practise, the abstract model of a subcomponent can be defined by its formal specification, or computed recursively from analysis of its ADL and its code.

As in chapters 3 and 4, before computing any synchronous product, we build finite abstraction of our models using finite instantiations of the data values of parameters. Whenever the checking tools allow it, this instantiation and the corresponding state space generation is done on-the-fly during the proof. This data instantiation is interpreted as a partition of the data domains and induces an abstract interpretation of the parameterized LTS. The instantiation also will be chosen with respect to the values occurring in the properties we interested in.

5.5 The User View

The models for the non-functional aspects described in this chapter are built automatically. The user only has to provide the architecture through the Fractal ADL and the functional behaviour of the primitive components (corresponding to the **Body** part in Figure 5.7).

5.5.1 Looking at one Example

Figure 5.9 shows a simple example of a hierarchical component system in Fractive. This is the classical problem of a bound buffer with one consumer and one producer. Both the consumer and the producer consumes/produces one element at a time. Additionally the buffer emits an alarm through its interface I_{alarm}, when the buffer is full.

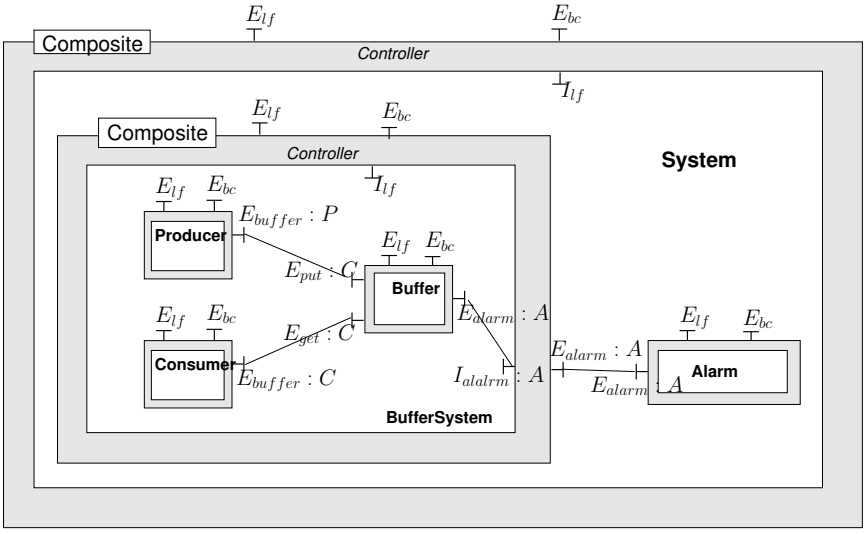


Figure 5.9: Consumer-Producer sample

As we show in chapter 4, the user describes the system's topology using the Fractal Architecture Definition Language (ADL). Fractive uses as well the ADL's XML syntax provided by Fractal. The ADL file describing **System** for the example on this chapter, is shown in Figure 5.10.

System.fractal

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE .... >
3
4  <definition name="components.System">
5
6   <component name="BufferSystem"
7       definition="components.BufferSystem(3)">
8    <interface name="alarm" role="client"
9       signature="components.AlarmInterface"/>
10  </component>
11
12  <component name="Alarm">
13   <interface name="alarm" role="server"
14      signature="components.AlarmInterface"/>
15  <content class="components.Alarm">
16   <behaviour file=''AlarmBehav''
17      format=''FC2Param''/>
18  </content>
19 </component>
20
21  <binding client="BufferSystem.alarm"
22      server="Alarm.alarm"/>
23 </definition>
```

Figure 5.10: System ADL

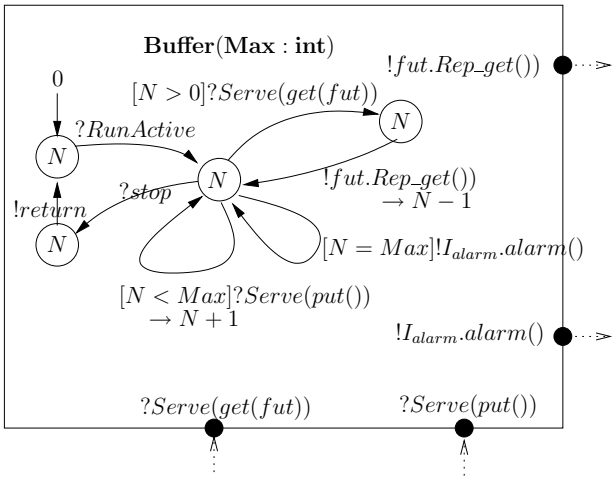


Figure 5.11: Buffer behaviour (provided by user)

The XML description shown in Figure 5.10 specifies that the system is composed of the composite **BufferSystem** (line 6), itself described in a separate file (`components/BufferSystem.fractal`), and the primitive **Alarm**, which implementation is the Java class `components.Alarm` (line 15). **BufferSystem** receives a construction parameter the maximal size of the buffer (3 in our example, line 7) and requires an interface named `alarm` of type `components.AlarmInterface` (lines 8,9). **Alarm** provides an interface named `alarm` of type `components.AlarmInterface` (lines 13,14).

Again, we extends the content tag of primitives to include their functional behavioural specification. For instance in Figure 5.10 we set that the behaviour of the class `components.Alarm` is defined in a file named `AlarmBehav` with format FC2 Parameterized (lines 16, 17).

As already mentioned, we assume that the functional behaviour of primitive components, i.e the body part in Figure 5.7, is known. It can be provided by the user or extracted from source by static analysis. For instance the behaviour of `Buffer` is given through an FC2 Parameterized file format, which is shown graphically in Figure 5.11. This parameterized automaton is instantiated with $Max = 3$ as set in line 7, Figure 5.10.

Finally, at lines 21, 22 in Figure 5.11, the ADL defines that at the initial deployment, the interface `alarm` of **BufferSystem** should be bound to the interface `alarm` of **Alarm**.

5.5.2 Automatic Construction

In chapter 4 we introduced a tool that hierarchically builds the behaviour of a component system. This tool automatically generates at each hierarchy level (each component) the binding-related behaviour for each interface as well as the life-cycle automaton of the component.

In the framework of Fractive the tool-set includes:

- A tool that hierarchically builds the behaviour model of a component system by analysing its ADL description. At each level of the component hierarchy, it builds the automata describing life-cycle, binding behaviour and the Fractive new elements, namely the automata encoding the request queue, the proxies for future responses, the NewServe policy for primitives and the RunActive policy for composites. This tool produces networks of parameterized automata in Parameterized FC2 format.
- A tool named FC2Instantiate, described in chapter 6.5, producing a finite instantiation of the system from a finite abstract domain for each parameter. These values may be in some cases taken from the system description, as the buffer capacity set to 3 in the ADL, or deduced from the significant values occurring in the properties. For parameters which types are simple (see [21]) these abstractions are abstract interpretations in the sense of [53, 148].
- Interface tools with the CADP tool-set [81], at the level of LTSs and of synchronisation networks. We then make a heavy use of the CADP tools (distributed state space generator, bisimulation minimiser, on-the-fly model-checker).

The length of Fractive request queues are unbound, and their abstraction must be chosen carefully. To generate a finite model of the system, we need to define a concrete value for the queue depths as an abstraction in the sense of [53, 148]. We do not go deeper on analysing the concrete value that best abstracts our example, although some basic facts can be taken in consideration:

- The consumer in our example never requests a element from the buffer simultaneously (it waits for the response before emitting a new request). Therefore the queue depth of the buffer should be at least 2. If is not the case, a request from the consumer, when the buffer is empty, would fill the buffer's request queue leaving no space for a call from the producer filling the buffer. As consequence the consumer's request will never be served (since the buffer is empty) leading to a deadlock situation. We can generalise this reasoning to n consumers, where the depth of the request queue of the buffer should be at least $n + 1$.
- More complex is the analysis for the queue depths when considering life-cycle aspects of the components. As we introduced in this work functional calls are not served meanwhile the component is stopped. Then a scenario to avoid would be a stopped component with a queue full of functional requests. As the queue is full, the reception of a `start` request will never success, the component will never be started and the functional requests in the queue will never be served, leading again to a deadlock situation.

Moreover, the choice of the queue depth is critical w.r.t. size of the generated state space: considering request queues of size 3, we were only able to generate the state space of **BufferSystem** (approx. 191M states and 1,498M transitions) on a cluster composed of 24 bi-processor nodes using the distributed model generation tool *distributor* from the CADP tool-set. For the complete system we did not even try to generate the complete automaton.

Staying in the context of explicit-state tools, we use a better approach : we define the set of non functional actions (whether in deployment or reconfigurations) involved in each specific property we want to prove. Then we forbid any other non-functional actions for the model and we use this set to determine an approximation for the queue's length. Given those parameters, we build all the basic automata, hiding any action not involved in the properties, and reducing the basic automata w.r.t. weak bisimulation. Last, we compute the products of the reduced automata, using the on-the-fly verification feature of CADP. This approach has enabled us to verify all properties listed in the next section using a simple desktop machine (CPU Pentium 3GHz, RAM 1.5 GB).

5.6 Properties

The preceding sections focused on building the correct models. This section presents some properties to illustrate the verification power of our approach. On this chapter, we choose to use regular alternation-free μ-calculus [129].

As we have introduced in chapter 2, regular alternation-free μ-calculus is an extension of the alternation-free fragment of the modal μ-calculus with action predicates and regular expressions over action sequences. It allows direct encodings of "pure" branching-time logics like CTL or ACTL among others. Moreover, it has an efficient model checking algorithm, linear in the size of the formula and the size of the LTS model.

5.6.1 Deployment

For synchronous components (chapter 4), the *static* automaton represents the normal behaviour of the component after deployment. In Fractive, method calls are asynchronous, and there may be delays between the request for a non-functional method and its treatment. So checking the execution of a control operation must be based on the observation of its application on the component, rather than the arrival of the request. In the syntax of our tools:

- The actions `Sig(bind(intf1,intf2))` and `Sig(unbind(intf1,intf2))` encodes when a binding between the interfaces `intf1` and `intf2` is effective. It corresponds for instance to the synchronisations `!bind/unbind(`E_RI^{nr}, I_{ext}`)` or `!bind/unbind(`$I\_RI^{np}, SubC^{k}.E\_PI^{scnp}$`)` in Fig. 5.4.

- The actions `Sig(start(name))` and `Sig(stop(name))` encodes when the component `name` is effectively started/stopped. It corresponds to the synchronisations `!start/stop` in Fig. 5.4.

One of the interesting properties is that the hierarchical start operation effectively occurs during the deployment; i.e. that the component and all its sub-components are at some point started. This property can be expressed as the (inevitable) reachability of `Sig(start(name))` in the static automaton of `System`, for all the possibles executions, where `name` = {`System, BufferSystem, Alarm, Buffer, Consumer, Producer`}. We leave the actions `Sig(start(name))` observable in the static automaton and we express this reachability property as the following regular μ-calculus formula, verified in our example:

$$\begin{aligned} &\mu X.(< true > true \wedge [\ \neg\ \texttt{Sig(start(System))}\ ]\ X) \wedge \\ &\mu X.(< true > true \wedge [\ \neg\ \texttt{Sig(start(BufferSystem))}\ ]\ X) \wedge \\ &\mu X.(< true > true \wedge [\ \neg\ \texttt{Sig(start(Alarm))}\ ]\ X) \wedge \\ &\mu X.(< true > true \wedge [\ \neg\ \texttt{Sig(start(Buffer))}\ ]\ X) \wedge \\ &\mu X.(< true > true \wedge [\ \neg\ \texttt{Sig(start(Consumer))}\ ]\ X) \wedge \\ &\mu X.(< true > true \wedge [\ \neg\ \texttt{Sig(start(Producer))}\ ]\ X) \end{aligned} \tag{5.1}$$

5.6.2 Pure-Functional Properties

Most of the interesting properties concern the behaviour of the system after its deployment, at least while there are no reconfigurations. For instance, in the example, we would like to prove that a request for an element from the queue is eventually served, i.e. that the element is eventually obtained. If the action of requesting an element is labelled as `get_req()` and the answer to this request as `get_rep()`, then this inevitability property is expressed as the following μ-calculus formula, as well verified by the static automaton of the example:

$$[true * .\texttt{get\_req}()]\mu X.(< true > true \wedge [\neg \texttt{get\_rep}()]X) \tag{5.2}$$

5.6.3 Functional Properties Under Reconfigurations

Our approach enables to verify properties not only after a correct deployment, but also after and during reconfigurations. For instance, property (5.2) becomes false if we stop the producer since at some point the buffer will be empty, and the consumer will be blocked waiting for an element. However, if the producer is restarted, the consumer will receive eventually an element and the property should become true again. In other words, we can check that, if the consumer requests an element, and then the producer is stopped, if eventually the producer is started again, the consumer will get the element requested.

For proving this kind of properties the static automaton is not sufficient, we need a behavioural model containing the required reconfiguration operations. We add to the component network a *reconfiguration controller* (Figure 5.12): its start state corresponds to the deployment phase, and the next state corresponds to the rest of the life of the component, where reconfigurations operations are enabled but are no more synchronised with the deployment. This state change is fired by the successful termination of the deployment ($\surd$).

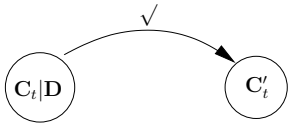


Figure 5.12: Synchronisation product supporting further reconfigurations

For the property stated above, the reconfigurations `?stop(Producer)` and `?start(Producer)` are left visible. This property is expressed by the μ-calculus formula, which is also insured in our example :

$$\begin{aligned}
&\texttt{(* If a request from the consumer is done before reconfiguration *)}\\
&[(\neg(?\texttt{stop(Producer)} \vee ?\texttt{start(Producer)}) * .\texttt{get_req}()](\\
&\texttt{(* a response is given before stopping the producer *)}\\
&\mu X.(< \neg ?\texttt{stop(Producer)} > true \wedge [\neg(\texttt{get_rep}() \vee ?\texttt{stop(Producer)})]X)\\
&\texttt{(* or given after restart the producer and without stopping it again *)}\\
&\vee [true * .?\texttt{start(Producer)}]\\
&\mu X.(< \neg ?\texttt{stop(Producer)} > true \wedge [\neg(\texttt{get_rep}() \vee ?\texttt{stop(Producer)})]X))
\end{aligned} \tag{5.3}$$

5.6.4 Asynchronous Behaviour Properties

Let us now focus on a property specific to the asynchronous aspect of the component model. The communication mechanism in Fractive allows any future, once obtained, to be updated with the associated value, provided that the corresponding method is served and terminates correctly; binds, unbinds or stops operation cannot prevent this. For example, if the consumer is unbound after a request, it gets anyway the response, even if the link is then unbound or the component stopped. Using the approach for reconfigurations described above: enabling `?unbind(buffer,Buffer.get)` and `?stop(Consumer)`, the property can be expressed as follows. This property is verified in the example:

$$[true * .\texttt{get\_req}()]\mu X.(< true > true \wedge [\neg\texttt{get\_rep}()]X) \tag{5.4}$$

Note that since we enabled the unbind and stop operations for the consumer, those actions occur in the automaton on which we prove the property.

5.7 Conclusion

On this chapter we have provided methods and tools for building the specification of distributed hierarchical components, in a hierarchical bottom-up fashion.

As in chapter 4, our approach relies on defining a synchronisation network of LTSs, each LTS expressing a different aspect of the component behaviour. The functional behaviour of primitive components are given by the user either with an specification language or obtained by data source analysis. The non-functional behaviours are automatically incorporated based on the component description. Then a set of properties

is described and proved on a component system example. Some of those properties, e.g. dealing with deployment, concern any component system and can be verified in a systematic way.

The main contributions of this chapter are:

- We define a general synchronisation network modelling the functional and non-functional behaviour at any hierarchical level of a distributed component system.
- We adapt the work done in [21] for modelling the behaviour of active (primitive and composite) components.
- In both primitives and composites we focus on the Fractive implementation of distributed components. We have incorporated the Fractive component features by automatically adding automata encoding the queues, future responses and serving policies depending on the life-cycle status
- Finally we prove a set of properties classified upon the component life phase, and considering the asynchronous aspects of Fractive components.

The model is automatically built from the functional behaviour of primitives and the component system description (as a XML file). We have illustrated our approach with a guided example: we described step by step the automatic construction of the model, and we discussed techniques to avoid the state explosion problem.

Again, as in chapter 4, the main originality of this chapter is to encode the deployment and reconfigurations as part of the behaviour, and thus verify the behaviour of the whole system of components.

Chapter 6

Tools

The work done during this thesis has involved the development of some automatic tools for the analysis and verification of distributed systems. Those tools are part of a integrated platform for verifying distributed system behaviours, named VERCORS [154].

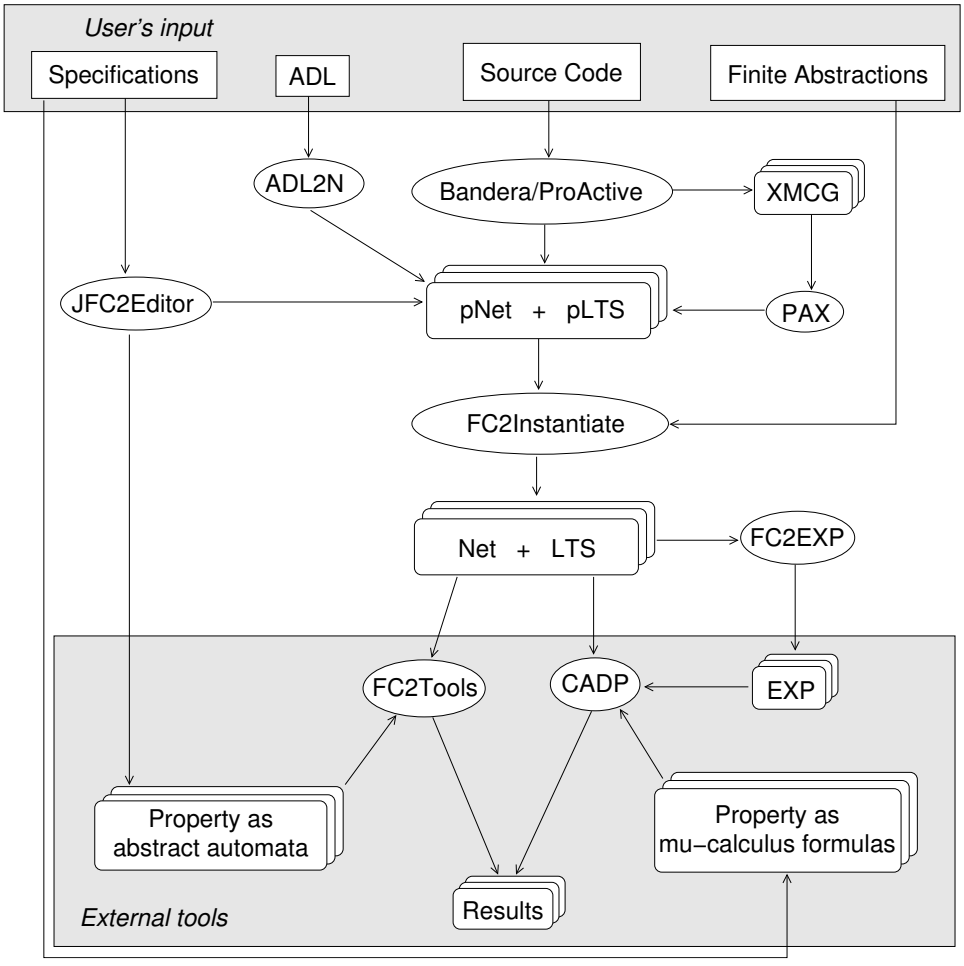


Figure 6.1: The VERCORS toolset

VERCORS is a set of integrated tools (being developed in this thesis author's team) for the formal verification of distributed systems. Figure 6.1 shows the current components being part of VERCORS and how they are related. In the upper part are the

input data sets given by the user to the toolset (specification (informal), ADL, source code and finite domain's abstractions). The tools are drawn in ellipses, CADP [81] and FC2Tools [36] are external tools (i.e. developed for another teams) which we have review in section 2.3.

The author of this thesis has developed the input syntax (FC2Parameterized) and software implementation of the tool FC2Instantiate, He also did the implementation of FC2EXP and set-up the basis for the tool ADL2N currently being developed by the team engineers.

6.1 Bandera/ProActive

The Bandera Project [54] aims to develop techniques and tools for automated reasoning about software system behaviour, and to apply these tools to construct high-confidence mission-critical software. Automated reasoning is achieved by (1) mechanically creating high-level models of software systems using abstract interpretation and partial evaluation technologies, and then (2) employing model-checking techniques to automatically verify that software specifications are satisfied by the model.

The Bandera tools includes a model generator from Java source code. We have adapted this tool to generate models for distributed Java programs built using the ProActive [28] middleware. The main function of our adapted tool is the identification, using complex static analysis techniques, of the various distributed entities in the Java source code (active objects references, futures, remote calls, parameters, etc.). The output of this tool is an Extended Method Call Graph (EMCG) [21] for each Java object and a set of pNets describing the topology of those objects. This work, done by Christophe Massol, is described in [124].

6.2 PAX

PAX [21, 40, 39] instead for Parametrized Automatic eXtractor. Given a Extended Method Call Graph (for an object), PAX construct a pLTS, using Structured Operational Semantic (SOS) rules, describing its behaviour. PAX is not yet complete implemented. Its formal basis where proposed by Rabea Boulifa during her thesis work [37].

6.3 JFC2Editor

JFC2Editor is a reimplementation in Java of Autograph [10] extended to support the FC2Parameterized format. As Autograph, JFC2Editor aims to become a tool for the modelling of both process behaviours and properties as abstract automaton.

The development of JFC2Editor is currently stopped and it is not clear if it would be retaken soon due to priorities and available resources.

6.4 ADL2N

The ADL to Network translator is a tool being developed. It takes a component system described in one or several ADL files and generates the pNets describing its topology. It also generates the additional pLTSs (such as queues and interceptors) described in section 5.

A prototype of this tool (with minimal functionalities) was developed by the thesis author to validate his work. In the current stage, the tool is being implemented by software engineers and running against several medium size example. In addition, supporting other output formats (such as LOTOS [104]) is being analysed.

6.5 FC2Instantiate

In order to work with our parameterized formalism, we have developed a tool, named FC2Instantiate, to get instantiations from the parameterized descriptions given the finite abstract domain of their unbounded parameters. Both description and domains inputs are in FC2Parameterized format, the output of the tool is standard FC2 format.

In this section we describe the syntax of the FC2Parameterized format and the use of the tool FC2Instantiate through a guided example.

Given a system of communicating automata with parameters and the domain of its unbounded parameters, FC2Instantiate is a tool, 100% written in Java, that generates a finite system of communicating automata by translating each of the parameters to all the values in its domain.

We start by reviewing the FC2 format, necessary to a better understanding of its extension, FC2Parameterized.

6.5.1 FC2 Format

As we mention in Section 2.2.1, the FC2 format allows for description of labelled transition systems and networks of such. Automata are tables of states, states being each in turn a table of outgoing transitions with target indexes; networks are vectors of references to subcomponents (i.e., to other tables), together with synchronisation vectors (legible combinations of subcomponent behaviours acting in synchronised fashion). Subcomponents can be networks themselves, allowing hierarchical description.

It is important to note that this format was created as a mean of communication between several software tools in the process algebra area [117]. The format specification defines only the syntax, and it is up to the tools to define a compatible semantics for the different format's entities. Indeed, it has been used with success in tools dealing with quite different flavours of process algebras, included timed ones.

In the following we introduce the syntax of FC2 format.

Structure of an FC2 file

An FC2 object (usually found in a file) consists of:

[optional] Declarations of expression operators,

A table of **nets**, containing:
the number of nets,
global semantic tables,
a global label,
the nets.
Each **net** (or **graph**) containing:
[optional] its number,
local semantic tables,
the net label,
The **vertice** table [optional], composed of:
the number of vertices,
the vertices.
Each **vertex** contains:
[optional] its number,
the vertex label,
The **edges** table [optional], composed of:
the number of edges,
the edges.
Each **edge** contains:
[optional] its number,
the edge label,
the index(es) of the resulting state(s).

FC2 simple example

```
nets 1
hook "main">0
net 0
struct "t1" logic "initial">0
hook"automaton"
vertice 2
vertex0 struct"v0" edges2
behav"a" result 0
behav"b" result 1
v1s"v1"E2
b"a"r0
b"b"r1
```

For example, this FC2 object contains a single net (indeed an automaton, as indicated in the net hook). The automaton has two vertices with two edges each. All information attached appears here directly in place, though it could have been tabulated. The text for the first vertex appears in long form (more readable), while the second vertex is in short, compact, form.

FC2 Objects

Expressions

```
exp : CONSTANT
    | UNARY exp
    | exp INFIX exp
    | PREFIX OP exp CP
    | OP exp CP
    | STRING
    | STAR
    | ref
    ;
```

```
ref :  INT
       |  GLOBAL INT
       |  BQUOTE INT
    ;
```

An expression represents an algebraic term. Some of the operators are predefined, some are user-defined, and declared in the file header. The basic bricks are the constant operators, the strings, and integers used as references in the semantic tables.

The "*" character has a special syntax (token STAR), used for specifying idle arguments in synchronisation vectors:

* is the constant expression "*idle*"

`*i` with `i` a positive integer, is equivalent to the sequence of `i` idle constants, "`*,*,...,*`".

Types Operators are typed, and the types are used to determine in which table a reference is to be searched. Possible types are: `behav, struct, logic, hook, int, any`. The type `any` is used as a type variable.

The following table lists the predefined operators. There is no predefined PREFIX.

```
Constant:  "t" : behav    (long form "tau")        Binary:    "^" : any * int -> any
           "q" : behav    (long form "quit")                  "+" : any * any -> any
           "_" : any                                          "<" : any * any -> any
Unary:     "!" : any -> any                                   ">" : any * any -> any
           "?" : any -> any                                   "," : any * any -> any
           "~" : any -> any                                   ";" : any * any -> any
           "#" : any -> any                                   "." : any * any -> any
```

All infix operators have priorities and they are left associative (e.g. `x.y.z` means `(x.y).z`). User-defined operators will be assigned a priority in the declarations as shown later.

In the following we list the priorities of predefined operators. The priorities range from 0 to 50, the lower the number, the strongest the priority.

Operator	Priority
+	50
> <	40
,	30
;	20
. ^	10

Example of expressions

```
tau     % the constant "tau", usually interpreted as the internal action
q       % the special action "quit" (or the "delta" of Lotos)
3       % a reference to the entry number 3 in the corresponding semantic table
!"s"    % the application of the unary operator "!" to the string "s",
        % e.g. the emission of signal "s"
1+2     % the sum of the expressions referenced 1 and 2

(t<1,!(3,?(11,1)))+(t<2,!(10,?(0)))
        % a big expression
```

Labels

Labels are used to store all kind of information attached to FC2 objects (nets, vertice, edges). They are records with 3 specific fields: **struct, behav, logic**, plus an additional

field, **hook**, gathering all kind of extra information. All fields are optional. When a field is present, it contains a single expression.

Hooks will usually contain information private to a tool, or to a small set of tools. In addition, some of the conventions defined in this document also make use of hooks. Hook information need not be defined by the format, so it will not be parsed: it appears as mere strings in the hook value. Each tool will decide whether or not to decode this value, depending on the name of the hook. On the other hand, some tools may agglomerate hooks when building new objects, so hook values (with same name) will be composed from string with the usual expression operators.

Example of labels

```
struct "name0"
behav "act1"+"act2"
                        % value can be explicite expressions
logic "initial"
logic (~0 + 1).2        % or use references

                        % To give a hook a name, one usually use the ">" operator
hook "colour">"red"
hook "coord">"x=134,y=24"+"x=35,y=124"
```

Convention : In a given label, the Struct, Behav, and Logic fields should appear at most once. There can be several Hook fields, though, and multiple hooks will be interpreted as conjunctions (as if connected by a "." operator).

Semantic Tables

Expressions may appear at many places in the file, but usually their values are tabulated, both for compact and semantic reasons. Thus, expressions are gathered in tables, sorted by type; there are four semantic table types, corresponding to the four label fields:

behavs, structs, logics, hooks

A *semantic table* has a type, a size, and a number of entries. Each entry has an optional index, and contains an expression. The size is a positive integer, bigger than all entry indexes. An entry without an index gets its index from the preceding entry, plus one.

Example of Semantic Table

```
behavs 3
:0 tau
:1 "a1"
:2 !1

logics 2
:0 "initial">0
:1 may (1)      % refers to behav :1, if "may" is behav -> logic
:2 diverge . must (2)
```

Tables can be found either inside a net (*local* table), or attached to the top level FC2 object (*global* table, shared by all nets)

Semantic interpretation: The size indication is intended to ease the creation/filling of tables at parse time: the "size" specified is the effective size of the table, and not the number of entries. Indexes start at 0. Indexes appearing explicitly in entries are –real– indexes; this implies that there can be holes in a table.

Nets

The nets table is the top level structure of an FC2 file.

```
net_table :  /* EMPTY */
          |  NETS INT tables labels net_list
     ;
```

The integer following the **nets** (or **N**) keyword is the number of nets in the file (mandatory). The "table" and "labels" non-terminals contain the global tables, and the global label (information concerning the whole network).

Convention: When needed, the root of the nets tree is indicated by a conventional hook of the form: `hook "main">0,` in which the right hand side argument of `>` is of type `net`

Each individual net contains local tables, a label, and a vertice table:

```
net :   NET opt_index tables labels vertice_table
```

The index of the net (integer immediately following the *net* keyword) is optional.

Convention: The hook of the net usually contains an indication of its *kind*. The kinds currently foreseen are: `"transducer", "synch_vector", "partition", "automaton"` The kind of a net carries information needed to resolve some ambiguities. The default kind is "automaton".

Convention: The initial vertex (or vertice) of an automaton or of a transducer may be indicated either in the logic field of the net label, or in the logic field of the vertex (resp. vertice) itself. Some tools (e.g. FC2Tools) require a single initial vertex to be defined.

Initial indication

```
...
net 0 hook "automaton" struct "foo"
      logic "initial">1
      vertice 3
v s"st_0" edges 1
b"a" r1
v s"st_1" edges 1
b"b" l"initial" r2
v s"stop"
```

The **struct** field of the net label is used to encode the structure of the network (which net contains which other nets), using a "`<`" operator, with the name of the node as first argument, and a comma-separated list of net references as second argument:

Example of Structured Net

```
nets 2
hook "main" "0"

net 0 hook "transducer" struct "system"<1,1,1
vertices ...
                    % the main net is called "system", and has
                    % three identical sons, copies of net 1.

net 1 hook "automaton" struct "cell"
                    % this one is a simple automaton named "cell"
                    % (a net with no argument)
```

Vertice and Edges

A vertice table is the main component of a net. The number of vertice in the table is mandatory.

```
vertice_table : /* Empty */
              | VERTICE INT vertex_list
              ;
...
vertex : VERTEX opt_index label edge_table
```

A Vertex contains an optional index, a label, and an edges table. The number of edges in the edges table is mandatory.

```
edge_table : /* Empty */
           | EDGES INT edges
           ;
...
edge : EDGE opt_index label target_vertice
           | label target_vertice
           ;
target_vertice : result
           | result exp
               ;
result : ARROW | RESULT /* -> or r */
       ;
```

In each **edge** entry:

- The **edge** (or **e**) keyword is optional.
- The edge index is optional.
- The result keyword is mandatory (either **r**, **result**, or `->`), and is followed by an expression of type vertex, usually a single vertex index, or a "+"-separated list of vertex indexes (e.g. "`1+2+5+6`").
- When a net has only one vertex (usual for synchronisation vectors) this index (but not the preceding **result** keyword) can be omitted.

Example of Edges

```
v s"state1" E2       % vertex with 2 edges
e0 b?0 r1            % edge 0 has behaviour ?0 and one target vertex (number 1)
e1 b?1 r 1+2+3       % edge 1 has 3 target vertice, number 1, 2, and 3
                     %
vertex 2 struct "state2" edges 2    % another vertex, in expanded syntax
edge 0 behav ?0 -> 1
edge 1 behav !1 result 1+2+3
```

Declarations

The declaration section allows a specific tool to use more operators than those predefined in FC2. The syntactic class, the lexical representation, and the type of the operator must be specified.

The syntax class is one of **constant, unary, prefix, infix**. The difference between **unary** and **prefix** is that the latter requires parenthesis surrounding its arguments. Locally-declared **infix** operators should be assigned to a priority between 0 and 50.

The lexical representation must be a legal token for the FC2 scanner, that is either a SYMBOL (single character symbol) or an IDENT (fully alphabetic identifier).

The type must be coherent with the syntactic class.

```
declaration : decl_class decl_token decl_type
            ;
decl_class : PREFIX_DECL | UNARY_DECL | INFIX_DECL priority INT | CONSTANT_DECL
           ;
decl_token : IDENT | SYMBOL
           ;
decl_type : OP types CP ARROW type
          ;
```

Example of Declarations

```
constant diverge () -> logic
infix $ (logic logic) -> logic priority 45
prefix & (struct behav behav) -> struct
```

6.5.2 Specification of Parameterized System

The FC2Parameterized format allows the systems specification using parameterized networks of communicating automata. It is an extension of the standard FC2 format with additional operators (introduced in detail in section 6.5.5) and user defined variables. This extension is done using the declarations section of FC2 format. We use the graphical syntax introduced in section 3.2.2 for a better understanding.

Example description

We use the Consumer-Producer example introduced in section 3.2.2 and section 3.2.3. The example is shown in Fig. 6.2, it is composed of a single bounded buffer, with a maximal capacity of Max elements, and a bounded quantity of consumers ($\#consumer$) and producers ($\#producer$) running in parallel.

Both producers and consumers have a single functionality. The producers keep feeding the buffer with an arbitrary quantity (x) of elements without waiting for a response, while the consumers request an element from the buffer and wait for its response.

On the contrary, the buffer has two functionalities. On one side it keeps a bounded stock where elements are added and taken, but it also manages a queue where the request from the consumers are enqueued until elements become available.

We model each functionality of the system as a pLTS, and the hierarchical composition of this functionalities (pLTS) as pNets. We chose to describe the example behaviour in two FC2 files: `SystemParameterized.fc2` describes the synchronisation network (i.e. the composition) between the producers, consumers and the buffer; both

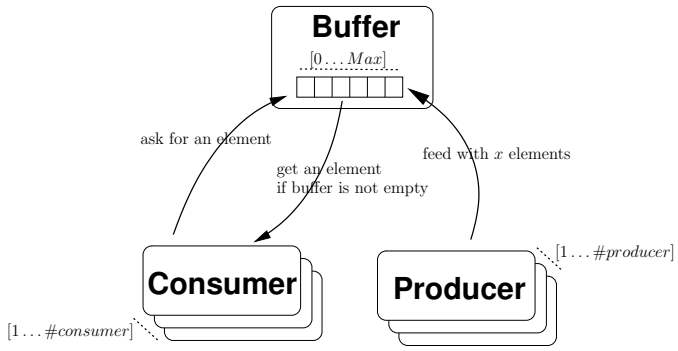


Figure 6.2: Consumer-Producer system interaction

the behaviour of a consumer and the behaviour of a producer is included in the same file, while the behaviour of the buffer is described in the file `BufferParameterized.fc2`.

The System

Let's take a look back to the system, it is shown in Figure 6.3. The arbitrary numbers of consumers and producers are respectively represented by the exponents c and p in the figure.

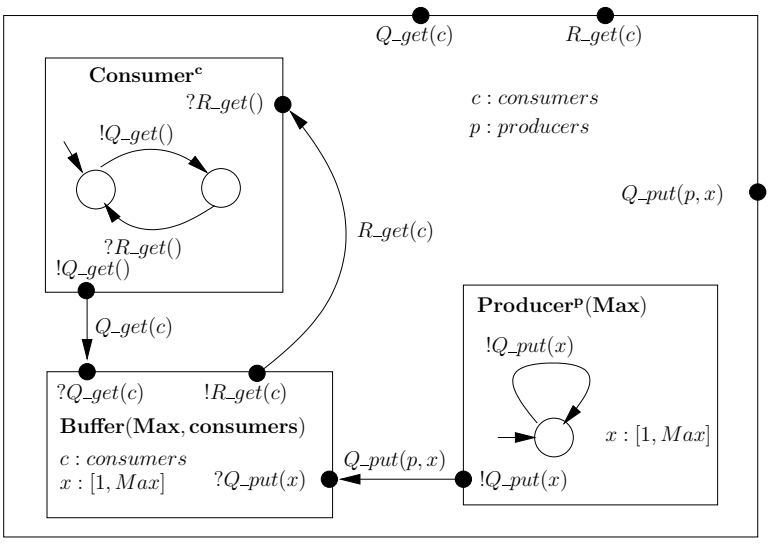


Figure 6.3: Parameterized consumer-producer system

In the FC2Parameterized format, the system is described by three nets (consumer, producer and buffer) and a fourth net defining the synchronisation between them (having the synchronisation vectors) as follows:

SystemParameterized.fc2

```
declarations
constant consumers() ->any
constant producers() ->any
infix & (any any ) -> any priority 8
... % other necessary declarations
nets 4
   hook"main" > 0
   struct"Consumer-Producer"
   net 1
      struct "Buffer"
... % the rest of the net definition (semantic table)
   net 2
      struct "Consumer"
... % the rest of the net definition (semantic table & vertices table)
   net 3
      struct "Producer"
... % the rest of the net definition (semantic table & vertices table)
   net 0
      hook "synch_vector"
      struct _< 1,2&consumers,3&producers
... % the rest of the net definition (including synchronisation vectors)
```

When writing a parameterized system, all the variables and operators not predefined in the standard FC2 format should be declared in the `declarations` section of the file. In our example, the file begins declaring the variables `consumers` and `producers`, encoding respectively the set of consumers and producers. The keyword to declare a variable in FC2Parameterized is `constant`. The type of the variables (after the arrow) can be any of the FC2 Types. However, the FC2Instantiate tool does not make type checking yet, meanwhile we mainly use the type "any".

The operator `&` declared as infix in the file, when is used within the `struct` label of a `net` instantiates the referenced net (its left argument) to the size of the given set (its right argument). Then `struct _< 1,2&consumers,3&producers` indicates that the network is composed by one single instance of the network 1, #`consumers` instances of the network 2 and #`producers` instances of the network 3. For instance if `consumers` = {`"cons1", "cons2"`} and `producers = [2, 4]`, then using the tool `struct _< 1,2&consumers,3&producers` is expanded to `struct _< 1,2,2,3,3,3`.

Before given the complete `SystemParameterized.fc2` file contents, we analyse each one of its networks.

Consumer

The parameterized automaton modelling the Consumer behaviour is shown in Figure 6.4.

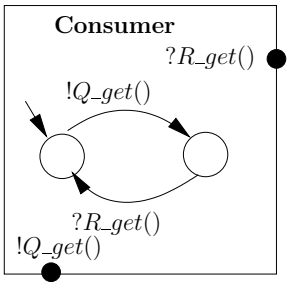


Figure 6.4: Parameterized Consumer

SystemParameterized.fc2

```
...
   net 2
      struct "Consumer"
      behavs 2
         :0 "Q_get()"
         :1 "R_get()"
      logic "initial">0
      behav !0+?1
      hook "automaton"
      vertice 2
         vertex 0
            edges 1
               edge0
                  behav !0 -> 1
         vertex 1
            edges 1
               edge0
                  behav ?1 -> 0
...
```

Since the consumer does not use parameters, its definition is done using the standard FC2 format as shown next to the figure.

Producer

The parameterized automaton modelling a Producer behaviour is shown in Figure 6.5. The transition's alphabet is specified in the behaviour's semantic table of the net. The semantic table for the Producer is shown next to the figure.

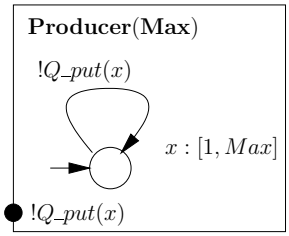


Figure 6.5: Parameterized Producer

SystemParameterized.fc2

```
declarations
...
  infix & (any any ) -> any  priority 8
  prefix in (any any) -> any
...
  net 3
    behavs 1
      :0 "Q_put"&in(1,Max)
...
```

The `&` operator, when inside a semantic table, takes a FC2 Expression in its left side and a list of sets (each set separate by "`,`") in its right side (in our example there is only one set). This operator generates, during instantiation, an entry in the semantic table for each member of the Cartesian product of the sets in its right side.

The `in` operator takes two integers and generates the set of all the integers between those integers inclusive ($in(1,3) = \{1,2,3\}$). For instance, when running the FC2Instantiate tool with $Max = 3$, the semantic table above is extended to:

SystemInstantiatied.fc2

```
...
  net 3
    behavs 3
      :0 "Q_put(1)"
      :1 "Q_put(2)"
      :2 "Q_put(3)"
...
```

Within states (vertices) and transitions (edges), their local variables should be assigned using a hook label (one per variable). The variables are assigned with the infix operator `=`, which takes a variable name in its left side and a set in its right side. When using the tool, the variable is assigned to each element of the set.

In the producer the state has no variables. On the contrary, the only transition contains the variable x which will range in the set $[1, Max]$. Then the vertices table of the Producer is:

SystemParameterized.fc2

```
declarations
...
  infix = (any any ) -> any priority 8
  infix & (any any ) -> any priority 8
...
    vertice 1
      vertex 0
        edges 1
          edge0
            hook x=in(1,Max)
            behav !(0&x) -> 0
```

The left parameter of the operator `&`, when used inside an automaton's edge (which is not a synchronisation vector), is a reference to an action in the corresponding semantic table of the `net`; and the right parameter is an expression. When using the tool, an expression of the form `0&x` (inside an automaton edge) will be replaced by a reference to the entry `0` of the semantic table (note that the corresponding entry in the table must be also parameterized by `x` or equivalent) for each instantiation of `x`.

The complete parameterized description of a producer and its instantiation when $Max = 3$ are:

SystemParameterized.fc2

```
declarations
  constant Max() -> int
  constant x() -> int
  infix & (any any ) -> any priority 8
  infix = (any any ) -> any priority 8
  prefix in (any int) -> any
...
  net 3
    behavs 1
      :0 "Q_put"&in(1,Max)
    logic "initial">0
    hook "automaton"
    vertice 1
      vertex 0
        edges 1
          edge0
            hook x=in(1,Max)
            behav !(0&x) -> 0
...
```

SystemInstantiated.fc2

```
...
  net 3
    behavs 3
      :0 "Q_put(1)"
      :1 "Q_put(2)"
      :2 "Q_put(3)"
    logic "initial">0
    hook "automaton"
    vertice 1
      vertex 0
        edges 3
          edge0
            behav !0 -> 0
          edge1
            behav !1 -> 0
          edge2
            behav !2 -> 0
...
```

Buffer

The buffer communicates with consumers and producers through three actions: *Q_get* to receive a request for element from a consumer, *R_get* to give the answer to the consumer and *Q_put* to receive feeds from producers. Since the behaviour of the buffer is given in a separate file, in the file `SystemParameterized.fc2` only is specified its behaviour semantic table (i.e. its alphabet of actions) as follows:

SystemParameterized.fc2

```
declarations
...
  constant Max() -> int
  infix & (any any ) -> any priority 8
...
   net 1
      struct "Buffer"
      behavs 3
         :0 "Q_get"&consumer
         :1 "R_get"&consumer
         :2 "Q_put"&in(1,Max)
      behav ?0+!1+?2
      hook "synch_vector"
```

Synchronisation vectors and complete `SystemParameterized.fc2` file

Finally in the `net 0` we give the synchronisation vectors that define the actions from the producers, consumers and the buffer which should be synchronised.

In the synchronisation vector we use the $ operator to indicate which of the instances of a network is being to be referenced.

In a synchronisation vector a reference such as `behav 0&c < ?(0&c)$1, !0$(2&c) -> 0` indicates that the action labelled with the entry 0 (for an instantiation of c) in the net 1, is synchronised with the entry 0 of the net 2 (where the net 2 is instantiated for an evaluation of c). This synchronisation is observable and produces a global action labelled with the entry 0 (for an instantiation of c) in the behaviour semantic table of the net having the synchronisation vector.

The complete `SystemParameterized.fc2` file is shown bellow:

SystemParameterized.fc2

```
declarations
constant Max() -> int
constant x() -> int
constant consumers() ->any
constant producers() ->any
constant c() ->any
constant p() ->any
constant queue() ->any
infix & (any any ) -> any priority 8
infix $ (any any ) -> any priority 8
infix = (any any ) -> any priority 8
prefix in (any int) -> any
nets 4
  hook"main" > 0
  struct"Consumer-Producer"
  net 1
    struct "Buffer"
    behavs 3
      :0 "Q_get"&consumers
      :1 "R_get"&consumers
      :2 "Q_put"&in(1,Max)
    behav ?0+!1+?2
    hook "synch_vector"
  net 2
    struct "Consumer"
    behavs 2
      :0 "Q_get()"
      :1 "R_get()"
    logic "initial">0
    behav !0+?1
    hook "automaton"
    vertice 2
      vertex 0
        edges 1
          edge0
            behav !0 -> 1 vertex 1
      vertex 1
        edges 1
          edge0
            behav ?1 -> 0
  net 3
    struct "Producer"
    behavs 1
       :0 "Q_put"&in(1,Max)
     logic "initial">0
    behav !0
    hook "automaton"
    vertice 1
      vertex 0
        edges 1
          edge0
            hook x=in(1,Max)
            behav !(0&x) -> 0
net 0
  struct _< 1,2&consumers,3&producers
  hook "synch_vector"
 behavs 3
   :0 "Q_get"&consumers
    :1 "R_get"&consumers
    :2 "Q_put"&(producers,in(1,Max))
  behav 0+1+2
   vertice 1
      vertex 0
       edges 3
         edge 0
           hook c=consumers
            behav 0&c < ?(0&c)$1, !0$(2&c) -> 0
          edge 1
           hook c=consumers
           behav 1&c < !(1&c)$1, ?1$(2&c) -> 0
         edge 2
           hook p=producers
           hook x=in(1,Max)
           behav 2&(p,x) < ?(2&x)$1, !(0&x)$(3&p) -> 0
```

As we mention before, if `consumers = {"cons1", "cons2"}` and `producers = [2, 4]`, then using the tool the expression `struct _< 1,2&consumers,3&producers` is expanded to `struct _< 1,2,2,3,3,3`. Then when `c="cons2"` and `p=3`, the vector `behav 0&c < ?(0&c)$1, !0$(2&c) -> 0` becomes `behav 1 < ?1,*, !0, *,*, * -> 0`.

Buffer's behaviour

The buffer's behaviour is described through a synchronisation network between two components: **stock** and **queue**. **stock** takes care of keeping the actual number of elements (up to Max) in the buffer's stock and receive the feeds from the producers. **queue** receives the requests from the consumers and put them in a queue. The answers to the consumers are given in FIFO order from the request queue until elements are available in the buffer's stock.

The network describing the buffer behaviour is shown in Figure 6.6.

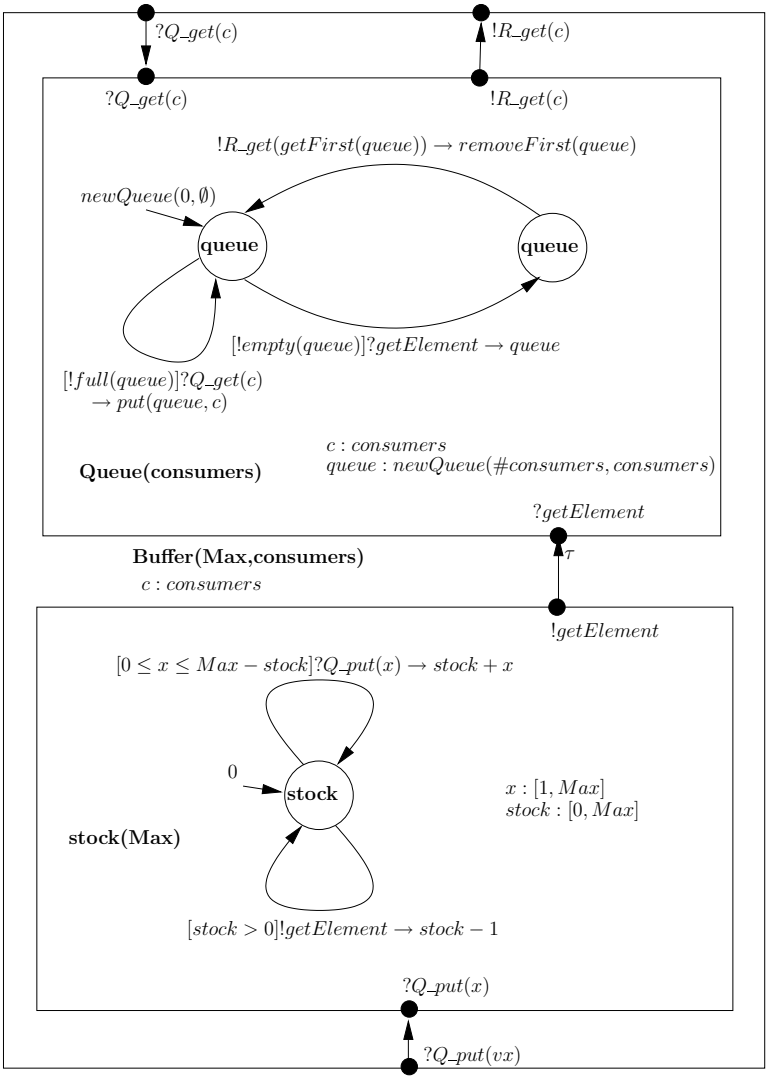


Figure 6.6: Parameterized Buffer

The behaviour of the buffer is specified in a separate file because FC2Instantiate does not support nested synchronisation networks in a single file.

In the stock, its only state is parameterized with the variable **stock**. The variable **stock** encodes the quantity of elements actually available in stock. When an element is taken, this variable is decreased by 1, which is expressed by a transition to the state encoding the stock having one element less, in the figure correspond to the transition labelled $[stock > 0]!getElement \rightarrow stock - 1$. When feeding the stock (action Q_put) this variable is incremented by the quantity of elements received, transition labelled as $?Q\_put(x)$ ($[0 \leq x \leq Max - stock]?Q\_put(x) \rightarrow stock + x$ in the figure. Note that both transitions are guarded to avoid taken an element from an empty stock or to overfill it.

The FC2 file describing the stock is:

BufferParameterized.fc2

```
declarations
constant Max() ->any
constant stock() ->any
constant x() ->any
infix & (any any ) -> any priority 8
infix = (any any ) -> any priority 8
infix - (any any ) -> any priority 8
prefix in (any int) -> any
prefix greaterThan (any any) -> any
prefix when (any) -> any
...
   net 2
      struct "stock"
      behavs 2
         :0 "Q_put"&in(0,Max)
         :1 "getElement"
      logic "initial">0
      behav ?0+!1
      hook "automaton"
      vertice 1
         vertex 0
            hook stock=in(0,Max)
            edges 2
               edge0
                  hook x=in(1,(Max-stock))
                  behav ?(0&x) -> 0&(stock+x)
               edge1
                  hook when(greaterThan(stock,0))
                  behav !1 -> 0&(stock-1)
```

As you can see, new operators supported by the tool are introduced: the conditional operator `when`, the comparison operator `greaterThan`, and the arithmetic operators `-` and `+`.

In the queue, the states are parameterized by the state variable **queue** which encodes the states of a queue. A queue is characterised by its contents.

When a request for one element is received ($?Q\_get(c)$), the caller id (c) is appended to the end of the queue $put(queue, c)$. As soon as an element is available in the stock ($?getElement$), the first caller from the queue is taken ($getFirst(queue)$) and a response to it is given ($!R\_get(c)$). At the same time the caller is removed from the queue ($removeFirst(queue)$).

Figure 6.6 introduces several operators supported by the FC2Instantiate tool to manipulate queues. Their complete descriptions are given in section 6.5.5.

The FC2 file describing the Queue is:

BufferParameterized.fc2

```
declarations
constant consumers() ->any
constant c() ->any
constant queue() ->any
infix & (any any ) -> any priority 8
infix = (any any ) -> any priority 8
prefix when (any) -> any
prefix instantiateQueue (any) -> any
prefix getFirst (any) -> any
prefix removeFirst (any) -> any
```

```
prefix fullQueue (any) -> any
prefix emptyQueue (any) -> any
prefix putQueue (any) -> any
prefix size (any) -> any
...
   net 1
      struct "queue"
      behavs 3
         :0 "Q_get"&consumers
         :1 "R_get"&consumers
         :2 "getElement"
      logic "initial">0
      behav ?0+!1+?2
      hook "automaton"
      vertice 2
         vertex 0
            hook queue=instantiateQueue(size(consumers),consumers)
            edges 2
               edge0
                  hook c=consumers
                  hook when(!fullQueue(queue))
                  behav ?(0&c) -> 0&putQueue(queue,c)
               edge1
                  hook when(!emptyQueue(queue))
                  behav ?2 -> 1&queue
         vertex 1
            hook queue=instantiateQueue(size(consumers),consumers)
            edges 1
               edge0
                  hook when(!emptyQueue(queue))
                  behav !(1&getFirst(queue)) -> 0&removeFirst(queue)
...
```

As specified in Figure 6.6, the complete Buffer behaviour is done by synchronising the *getElement* action of **stock** and of **queue** (the queue request elements from the stock). The full FC2 file describing this synchronisation is following:

BufferParameterized.fc2

```
declarations
constant Max() ->any
constant consumers() ->any
constant stock() ->any
constant x() ->any
constant c() ->any
constant queue() ->any
infix & (any any ) -> any priority 8
infix $ (any any ) -> any priority 8
infix = (any any ) -> any priority 8
infix - (any any ) -> any priority 8
prefix in (any int) -> any
prefix greaterThan (any any) -> any
prefix when (any) -> any
prefix instantiateQueue (any) -> any
prefix getFirst (any) -> any
prefix removeFirst (any) -> any
prefix fullQueue (any) -> any
prefix emptyQueue (any) -> any
prefix putQueue (any) -> any
prefix size (any) -> any
nets 3
   hook"main" > 0
```

```
struct"Buffer"
net 1
   struct "queue"
   behavs 3
      :0 "Q_get"&consumers
      :1 "R_get"&consumers
      :2 "getElement"
   logic "initial">0
   behav ?0+!1+?2
   hook "automaton"
   vertice 2
      vertex 0
         hook queue=instantiateQueue(size(consumers),consumers)
         edges 2
            edge0
               hook c=consumers
               hook when(!fullQueue(queue))
               behav ?(0&c) -> 0&putQueue(queue,c)
            edge1
               hook when(!emptyQueue(queue))
               behav ?2 -> 1&queue
      vertex 1
         hook queue=instantiateQueue(size(consumers),consumers)
         edges 1
            edge0
               hook when(!emptyQueue(queue))
               behav !(1&getFirst(queue)) -> 0&removeFirst(queue)
net 2
   struct "stock"
   behavs 2
      :0 "Q_put"&in(0,Max)
      :1 "getElement"
   logic "initial">0
   behav ?0+!1
   hook "automaton"
   vertice 1
      vertex 0
         hook stock=in(0,Max)
         edges 2
            edge0
               hook x=in(1,(Max-stock))
               behav ?(0&x) -> 0&(stock+x)
            edge1
               hook when(greaterThan(stock,0))
               behav !1 -> 0&(stock-1)
net 0
   behavs 3
      :0 "Q_get"&consumers
      :1 "R_get"&consumers
      :2 "Q_put"&in(1,Max)
   struct _< 1,2
   behav ?0+!1+?2
   hook "synch_vector"
   vertice 1
      vertex 0
      edges 4
        edge 0
          hook c=consumers
          behav ?(0&c) < ?(0&c)$1 -> 0
        edge 1
          hook c=consumers
```

```
    behav !(1&c) < !(1&c)$1 -> 0
  edge 2
    hook x=in(1,Max)
    behav ?(2&x) < ?(0&x)$2 -> 0
  edge 3
    behav tau < ?2$1, !1$2 -> 0
```

6.5.3 Instantiation File

The domain of the global variables, i.e. the variables visible all over the parameterized system definition, is defined in a instantiation file. The instantiation file is given in FC2 format having a single net. The variables are assigned using the operator `=` in the hooks of that net (they can be assigned equally to a set or a value). For our consumer-producer example, we instantiate the system with 2 producers, 2 consumers and a maximal buffer capacity of 3. The instantiation file is:

InstantiationDomains.fc2

```
declarations
constant Max() ->any
constant consumers() ->any
constant producers() ->any
infix = (any any ) -> any priority 8
prefix in (any int) -> any
prefix set (any) -> any
nets 1
   net 0
     hook Max=3
     hook consumers=set("cons1","cons2")
     hook producers=in(1,2)
```

6.5.4 Using the tool

The command to run FC2Instantiate is:

```
JAVA_CMD -cp FC2Instantiate.jar:FC2Parser.jar:jargs.jar\
fr.inria.oasis.fc2.FC2Instantiate [-o <net_file_output>]\
-d <definition.fc2> [-v] <instantiations.fc2>+
```

where `JAVA_CMD` is the Java runtime command and

- `<net_file_output>` is an optional file to print the result. If it is not given the result will be print to the standard output.
- `<definition.fc2>` is the file describing the parameterized system (in FC2 parameterized format)
- `<instantiations.fc2>+` is a list of files defining the domain of the global variables. If a variable is defined in more than one file, it takes the value defined in the last file where it is defined.
- `-v` for extra information (debugging)

6.5.5 FC2Parameterized reference manual

While the FC2Parameterized format is not "human-friendly", it is a powerful language to describe behaviour of systems. Even when humans may directly use this language to model distributed system, its main target is to be the intermediate format produced by our automatic tools in order to do verification.

This section contains a preliminary version of the FC2Parameterized reference manual. It is expected that the tool (FC2Instantiate) and the format will evolve quickly in the short term, and a separate reference manual document will follow.

Additionally to the predefined operators in the FC2 format, the FC2Parameterized defines the following (all of them supported in the FC2Instantiate):

General Operators

- & : $Exp \times Exp \rightarrow Exp$, where Exp is an FC2 expression. Its semantic depends on the context where is used:
  - *In a semantic table.* The & operator, when inside a semantic table, takes a FC2 Expression in its left side and a list of sets (each set separate by ,) in its right side. This operator generates an entry in the semantic table for each member of the Cartesian product of the sets in its right side.
  - *In an automaton's edge.* The left parameter of the operator &, when used inside an automaton's edge (which is not a synchronisation vector), is a reference to an action in the corresponding semantic table (for instance, the behaviour label in the edge/vertex, reference the behaviour table of the net). The right side contains an expression. When using the tool, an expression of the form `0&x` (inside an automaton edge) will be replaced by a reference to the entry `0` of the semantic table (note that the entry in the table should be also parameterized by `x` or equivalent) for each instantiation of `x`.
  - *In a synchronisation vector.* When it is present at the left side of the $ operator, it semantic is the same that when is inside an automaton's edge (defined above). When is in the right side of the $ operator is analogous to the edge case, but it references to an evaluation of a net instead of a semantic table.
  - *In the struct label of a net.* In this case the & operator indicates the number of instantiations to be done for a net. In its left side is the referenced net, and in its right side a set. The number of instantiations of the nets will be the same as the number of elements of the set. For instance `struct _< 1,2&consumers,3&producers` indicates that the network is composed by one single instance of the network 1, #consumers instances of the network 2 (consumers is a variable encoding a set) and #producers instances of the network 3.
- $: $Exp \times Exp \rightarrow Exp$, where Exp is an FC2 expression. In the synchronisation vectors, the $ operator is used to indicate which of the instances of a network is being referenced.

- *when* : *boolean* is used in a hook to indicate a condition. When the condition is not true, the instantiation stops for the vertex/edge holding the hook.
- $=: varName \times Set$ is for assignment of values to a variable. When instantiating, the variable on the left side will be assigned to each element in the set of the right side.

Set operators

- $set : List(any) \rightarrow Set$. This operator receives a list of elements and constructs an ordered set containing them. Ex: `var=set("a",3,"b","other")` ($var = \{"a", 3, "b", "other"\}$).
- $in : int \times int \rightarrow Set$. The operator receives two integers and generates the set of all the integers between them inclusive. Ex: `var=in(2,5)` ($var = \{2, 3, 4, 5\}$).
- $size : Set \rightarrow int$. The size operator receives a set and returns the number of elements in it. Ex: `size(var)=4` ($var = \{"a", 3, "b", "other"\}$).
- $merge : Set \times Set \rightarrow Set$. The merge operator receives two sets and returns an unique set with the disjoint union of elements in both sets. Ex: `merge(set("a","b"),merge(in(1,2),set("b")))` ($= \{"a", "b", 1, 2, "b"\}$).

Queue Operators

Often in distributed systems, because their asynchronous communication nature, it is needed to model *queues*, sometimes named *channels*. FC2Parameterized provides the following set of queue manipulation:

- $instantiateQueue : Set \times int \rightarrow queue$. This operator receives a set of the potentially elements that could be added to the queue, and a maximal capacity of the queue. It generates the queue (queue is an internal type for FC2Parameterized). Ex: `queue=instantiateQueue(var,6)` ($var = \{"a", 3, "b", "other"\}$).
- $putQueue : queue \times any \rightarrow queue$. This operator receive a queue and an element to be added at the end of the queue. It returns the queue with the element already added. Ex: `queue=putQueue(queue,"newElement")`.
- $getFirst : queue \rightarrow any$. This operator returns the first element of a queue. Ex: `var=getFirst(queue)`.
- $getFirstFilter : queue \times Set \rightarrow any$. This operator returns the first element of a queue that is in the set *Set*. Ex: `var=getFirstFilter(queue,set("a",2))`.
- $removeFirst : queue \rightarrow queue$. This operator returns the queue *queue* without it first element. Ex: `queue=removeFirst(queue)`.
- $removeFirstFilter : queue \times Set \rightarrow queue$. This operator returns the queue *queue* without the first element in the queue which is also in the set *Set*. Ex: `queue=removeFirstFilter(queue,set("a",2))`.

- $fullQueue : queue \rightarrow boolean$. This operator checks whether the queue has reach the maximal capacity. Ex: `fullQueue(queue)`.
- $emptyQueue : queue \rightarrow boolean$. This operator checks whether the queue is empty. Ex: `emptyQueue(queue)`.
- $emptyQueueFilter : queue \times Set \rightarrow boolean$. This operator returns true if the queue has no elements from the set Set. It returns false otherwise. Ex: `emptyQueueFilter(queue,set("a",1))`.
- $hasElement : queue \times any \rightarrow boolean$. This operator returns true if the element any is in the queue $queue$. It returns false otherwise.

Arithmetic operators

- $+ : int \times int \rightarrow int$. Addition between integers. Ex: `var=3+var`.
- `^`: $int \times int \rightarrow int$. Power. Ex: `var^3` ($= var^3$).
- $- : int \times int \rightarrow int$. Subtraction between integers.
- $\times : int \times int \rightarrow int$. Multiplication between integers.
- $/ : int \times int \rightarrow int$. Integer division between integers. The result is the integer part of the division.
- $mod : int \times int \rightarrow int$. It gives the common residue of the arguments.
- $greaterThan : int \times int \rightarrow boolean$. It returns true if the first argument is greater than the second one, false otherwise.
- $lessThan : int \times int \rightarrow boolean$. It returns true if the first argument is less than the second one, false otherwise.
- $equal : int \times int \rightarrow boolean$. It returns true if the first argument is equal than the second one, false otherwise.

6.6 FC2EXP

FC2EXP is a small tool that translate the set of synchronisation vectors in standard FC2 format (i.e. not parameterized), into a synchronisation vector list in the EXP format of the CADP [81] tool.

The FC2 file should not contain any automata, i.e. the nets in the file have only the behaviour semantic tables (alphabet) and the synchronisation vectors defined in the net 0. The behaviour of each net (excluding the net 0) is supposed to be in separate files.

The command to run FC2EXP is:

```
JAVA_CMD -cp FC2EXP.jar:FC2Parser.jar:jargs.jar:JCup.jar\
fr.inria.oasis.fc2exp.FC2EXP <file.fc2>
```

The result are given in the standard output.

For example, in the Consumer-Producer example analysed in section 6.5.2, the FC2 file describing the synchronisation between the Buffer and the producers/consumers (once instantiated[1]) is following:

SystemInstantiated.fc2

```
version "1.0"
nets 4
  hook "main" > 0
  struct "Consumer-Producer"
  net 3
    behavs 3
      :0 "Q_put(1)"
      :1 "Q_put(2)"
      :2 "Q_put(3)"
    struct  "Producer"
    behav !0+!1+!2
  net 2
    behavs 2
      :0 "Q_get()"
      :1 "R_get()"
    hook "synch_vector"
    struct  "Consumer"
    behav !0+?1
  net 1
    behavs 7
      :0 "Q_get(cons1)"
      :1 "Q_get(cons2)"
      :2 "R_get(cons1)"
      :3 "R_get(cons2)"
      :4 "Q_put(1)"
      :5 "Q_put(2)"
      :6 "Q_put(3)"
    hook "synch_vector"
    struct  "Buffer"
    behav ?0+?1+!2+!3+?4+?5+?6
net 0
  behavs 10
    :0 "Q_get(cons1)"
    :1 "Q_get(cons2)"
    :2 "R_get(cons1)"
    :3 "R_get(cons2)"
    :4 "Q_put(1,1)"
    :5 "Q_put(1,2)"
    :6 "Q_put(1,3)"
    :7 "Q_put(2,1)"
    :8 "Q_put(2,2)"
    :9 "Q_put(2,3)"
  hook "synch_vector"
  struct  _<1,2,2,3,3
  behav 0+1+2+3+4+5+6+7+8+9
  vertice 1
    vertex0
      edges 10
        edge0 behav  0<?0,!0,*,*,* -> 0
        edge1 behav  1<?1,*,!0,*,* -> 0
        edge2 behav  2<!2,?1,*,*,* -> 0
        edge3 behav  3<!3,*,?1,*,* -> 0
        edge4 behav  4<?4,*,*,!0,* -> 0
        edge5 behav  5<?5,*,*,!1,* -> 0
        edge6 behav  6<?6,*,*,!2,* -> 0
        edge7 behav  7<?4,*,*,*,!0 -> 0
        edge8 behav  8<?5,*,*,*,!1 -> 0
        edge9 behav  9<?6,*,*,*,!2 -> 0
```

FC2EXP would translate this file to the following:

SystemInstantiated.exp

```
label par
"?'Q_get(cons1)'" * " !'Q_get()'" * _ * _ * _ -> "Q_get(cons1)",
"?'Q_get(cons2)'" * _ * " !'Q_get()'" * _ * _ -> "Q_get(cons2)",
" !'R_get(cons1)'" * "?'R_get()'" * _ * _ * _ -> "R_get(cons1)",
" !'R_get(cons2)'" * _ * "?'R_get()'" * _ * _ -> "R_get(cons2)",
"?'Q_put(1)'" * _ * _ * " !'Q_put(1)'" * _ -> "Q_put(1,1)",
"?'Q_put(2)'" * _ * _ * " !'Q_put(2)'" * _ -> "Q_put(1,2)",
"?'Q_put(3)'" * _ * _ * " !'Q_put(3)'" * _ -> "Q_put(1,3)",
"?'Q_put(1)'" * _ * _ * _ * " !'Q_put(1)'" -> "Q_put(2,1)",
"?'Q_put(2)'" * _ * _ * _ * " !'Q_put(2)'" -> "Q_put(2,2)",
"?'Q_put(3)'" * _ * _ * _ * " !'Q_put(3)'" -> "Q_put(2,3)"
in
"Buffer" || "Consumer" || "Consumer" || "Producer" || "Producer"
end par
```

[1] instantiation values: producers=$[1,2]$, consumers=$\{cons1, cons2\}$

Chapter 7

Conclusions and future works

This thesis focused on behavioural properties verification of distributed component systems, that would be applicable in automatic tools, on a real system.

We began discussing the need for computer systems reliability and introduced the use of formal methods as a powerful technique to achieve this goal. Then we stated the particular difficulties for verifying distributed systems using such methods and, reviewed the current state of the art on the field.

We discussed the reasons why existing formalisms and description languages are not suitable for our target systems and goals. Based on this discussion, we proposed a new approach for modelling distributed systems. We validated our approach with a case study, and applied it for the automatic modelling and reasoning of distributed component systems.

The main contributions of this thesis include:

- A new approach for modelling the behaviour of distributed components systems. This approach takes the best from two relevant work: networks of communicating automata [16, 15] and symbolic graphs with assignments [111, 93]. We named the behavioural models in our approach "parameterized networks of communicating automata". Our parameterized models achieve three different roles, they describe: infinite systems in a natural and finite manner (when considering unbounded variable domains), a family of systems (when considering various variable domains), and in a compact way large systems (when considering large variable domains). In [21] we have shown that the models are suitable as the target language for static analysis tools.

- The definition of FC2Parameterized, a concrete syntax for writing system specifications using our parameterized models. The FC2Parameterized format was developed as an extension of the FC2 format [36, 118] capable of including parameters. We have also introduced a graphical notation for a sub-set of the FC2Parameterized format.

- A case study, using our approach, of a real distributed system: the Chilean electronic invoices system [65] (currently operational).

- The implementation of FC2Instantiate, a tool for obtaining (given the variable domains) finite non-parameterized systems from parameterized networks of com-

municating automata. This tool has proved itself useful for: comparing different instantiations, instantiating based on per-formula criteria and searching for better minimisations. Especially the tool's debugging capacity has provided early detection of errors or backtrack analysis.

- The use of parameterized networks of communicating automata for the behavioural specifications of hierarchical components. Having the behaviour of basic (primitive) components, we have developed a mechanism for automatically incorporating the non-functional behaviour within a controller built from the component description. The semantics of a composite is computed as a product of its sub-components LTSs and the controller of the composite.

- The use of the mechanism described above for the modelling of Fractive based systems. Fractive is a distributed components implementation using the middleware ProActive [28, 18, 47] of the Fractal [43] component model. Our mechanism includes the automatically incorporation of Fractive's features, such as request queues, future proxies and policies for serving method calls.

- The basis for a tool currently in development, named ADL2NET, for reading the system's descriptions, given as a set of Fractal ADL [43] files, and for generating the behavioural models using our mechanism.

- The implementation of FC2EXP, a tool and various small scripts for the incorporation of our formats into the verification toolsets FC2Tools [36] and CADP [81].

- A temporal classification of properties (section 4.4.1) to verify on component systems. Many of these properties, such as successful deployment and error detection, can be applied in a systematic way to any component system.

- The illustration of properties verification using three different formalisms to express properties: abstract automata [36, 10] (section 3.3.4), ACTL [60] formulas (section 4.4.2) and regular μ-calculus [129] formulas (section 5.6). The illustration includes properties on each temporal classification and considers asynchronous aspects of distributed components.

Additionally when it was pertinent during the development of this thesis, we proposed and analysed several mechanisms to avoid as much as possible the well-known *state explosion* in the construction of the system's behaviour.

Finally, many approaches have been developed to cover the right composition of components considering their functional aspects. One of the strongest advantage of using components is the separation of concerns from the user point of view. However, when applying behavioural verification, one still needs to take into account the interplay between functional and non-functional aspects, at least for existing component models. The main originality of our work is to encode the deployment and reconfigurations as part of the behaviour of the system, and thus verify the behaviour of the whole component system.

This thesis provides a step towards a concrete and strongly usable behavioural verification tool-set. This tool-set is capable of building the models automatically, and gives

feedback about generic properties and errors detection. The generated models allow defining and verifying further properties and to check them against a specification.

7.1 Future work

In the short term we are concentrating on finishing the implementation of the tools, and apply them on bigger case studies. This surely will provide feedback to improve our approach.

In the medium and the long term some axes of research that should be explored are:

7.1.1 Preorder relation

One of the question we left open on this thesis is whether a component is faithful with its specification. Answering this question would allow, without having to rebuild the complete behavioural model, to determinate if a specific component may safely replace another one.

There are a number of methods that can be used to address this issue; bisimulation equivalences (modulo renaming and hiding) would guaranty that all behavioural properties are preserved, but is too strong and many components that would fit safely in the system when replacing another one, will be refused.

One suitable approach could be to use the Sofa's *compliance* relation, but in Sofa the behaviours are expressed as regular expressions, and the *compliance* relation is based on trace language inclusion, though it is yet unclear how to compare with our bisimulation-based semantics.

Besides this issue, we believe the ideas which inspired the Sofa's *consent* relation, based on "obligations" that a process should accomplish, are well stated and we want to explore their applicability to our bisimulation-based semantics. This would lead to a refinement preorder, that allows the implementation to make some choices amongst the possibilities left by the specification, and compatible with the composition by synchronisation networks.

7.1.2 Properties specification

We have shown three different formalisms to express properties: abstract automata, ACTL and regular μ-calculus. Abstract automata can be viewed as more easy to use while regular μ-calculus is the most expressive formalism. ACTL can be located in between both.

The three of them require a qualified user and are far from being error-prone free. A very promising work for narrowing the gap between properties specification and the non-expert user has been done by Dwyer et al. [67], where they define, classified by temporal scope, various patterns (or macros) that allow to express many properties in a natural language-like syntax.

We want to propose extensions for Dwyer's patterns to cover component specific properties. For instance, we can define the macro `AfterDeployment` for meaning the temporal scope after a successful deployment. Others patterns could be `NoErrors`, `ControlActions` and `FutureUpdate`, all of the encoding specific set of actions.

Nevertheless, Before defining this patterns in a confident manner, further results from real case studies on distributed components are required.

7.1.3 New Fractive's features

Since Fractive is continuously being developed, with new features being added, like the latest: collective interfaces having broadcast, or selective multicast communications; or currently discussed issues such as: whether a component should preserve or not its request queue when updated, or even when it can be updated, are still in development within the team.

Therefore, our approach must evolve, as much as possible with Fractive's features.

Bibliography

[1] FracTalk: Fractal components in SmallTalk. http://csl.ensm-douai.fr/FracTalk.

[2] JULIA framework (fractal implementation). http://fractal.objectweb.org.

[3] OASIS project. http://www-sop.inria.fr/oasis.

[4] SOFA: Software appliances web site. http://nenya.ms.mff.cuni.cz/.

[5] *Wright web site*. http://www-2.cs.cmu.edu/ able/wright/.

[6] Application and theory of petri nets. In Claude Girault and Wolfgang Reisig, editors, *Selected Papers from the First and the Second European Workshop on Application and Theory of Petri Nets*, volume 52 of *Informatik-Fachberichte*. Springer, 1982.

[7] Statistical analysis of floating point flaw. Technical report, Intel Corporation, November 1994. available at http://support.intel.com/support/processors/pentium/fdiv/wp/.

[8] Distributed Component Object Model (DCOM). Technical report, Microsoft Corporation, November 1996.

[9] Building a better bug-trap. The Economist, June 2003.

[10] R. de Simone A. Ressouche, A. Bouali, and V. Roy. The FC2Tool user manuel. http://www-sop.inria.fr/meije/verification/, 1994.

[11] J. Adamek. Static analysis of component systems using behavior protocols. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 116–117. ACM Press, 2003.

[12] J. Adamek and F. Plasil. Component composition errors and update atomicity: Static analysis, 2004.

[13] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[14] J.C.M. Baeten andW. P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[15] A. Arnold. *Finite transition systems: semantics of communicating systems*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1994.

[16] A. Arnold. Nivat's processes and their synchronization. *Theor. Comput. Sci.*, 281(1-2):31–36, 2002.

[17] I. Attali, T. Barros, and E. Madelaine. Formalisation and proofs of the Chilean electronic invoices system. In *XXIV International Conference of the Chilean Computer Science Society (SCCC 2004)*, pages 14–25, Arica, Chili, November 2004. IEEE Computer Society.

[18] I. Attali, D. Caromel, and R. Guider. A step towards automatic distribution of java programs. In *FMOODS 2000, Stanford University.*

[19] Austry and Boudol. Algebre de processus et synchronisation. *Theoretical Computer Science*, 30, 1984.

[20] L. M. Barroca and J. A. McDermid. Formal methods: Use and relevance for the development of safety-critical systems. *Comput. J.*, 35(6):579–599, 1992.

[21] T. Barros, R. Boulifa, and E. Madelaine. Parameterized models for distributed java objects. In *FORTE'04 conference*, Madrid, 2004. LNCS 3235, Spinger Verlag.

[22] T. Barros, L. Henrio, and E. Madelaine. Behavioural models for hierarchical components. In Patrice Godefroid, editor, *Model Checking Software, 12th International SPIN Workshop*, volume LNCS 3639, pages 154–168, San Francisco, CA, USA, August 2005. Springer.

[23] T. Barros, L. Henrio, and E. Madelaine. Behavioural models for hierarchical components. Technical Report RR-5591, INRIA, June 2005.

[24] T. Barros, L. Henrio, and E. Madelaine. Verification of distributed hierarchical components. In *International Workshop on Formal Aspects of Component Software (FACS'05)*, Macao, October 2005. Electronic Notes in Theoretical Computer Science (ENTCS).

[25] T. Barros and E. Madelaine. Formal description and analysis for distributed systems. Technical Report 4-04, University of Kent, Computing Laboratory, April 2004. Doctoral Symposium at IFM'04, Canterbury, Kent, England.

[26] T. Barros and E. Madelaine. Formalisation and proofs of the Chilean electronic invoices system. Technical Report RR-5217, INRIA, June 2004.

[27] G. Batt, D. Bergamini, H. Jong, H. Garavel, and R. Mateescu. Model checking genetic regulatory networks using GNA and CADP. In *SPIN*, pages 158–163, 2004.

[28] F. Baude, D. Caromel, F. Huet, and J. Vayssière. Objets actifs mobiles et communicants. *Technique et science informatiques*, 21(6–2002):1–26, 2000.

[29] F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3-7 November*, Springer Verlag, 2003. LNCS.

[30] D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu. BISIMULATOR: A modular tool for on-the-fly equivalence checking. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 581–585. Springer, 2005.

[31] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1):109–137, 1984.

[32] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 1(37):77–121, 1985.

[33] J.A. Bergstra, A. Pose, and S.A. Smolka. *Handbook of Process Algebra*. North-Holland, 2001.

[34] M. Blair, S. Obenski, and P. Bridickas. Patriot missile defense: Software problem led to system failure at dhahran. Technical Report GAO/IMTEC-92-26, United States - General Accounting Office - Information Management and Technology Division, February 1992.

[35] R. E. Bloomfield and D. Craigen. Formal methods diffusion: Past lessons and future prospects. Technical Report D/167/6101/1, Bundesamt für Sicherheit in der Informationstechnik, Bonn, Germany, December 1999.

[36] A. Bouali, A. Ressouche, V. Roy, and R. de Simone. The FC2Tools set. In D. Dill, editor, *Computer Aided Verification (CAV'94)*, Standford, June 1994. Springer-Verlag, LNCS.

[37] R. Boulifa. *Génération de Modèles Comportementaux des Applications Réparties*. PhD thesis, Université de Nice - INRIA Sophia Antipolis, 2004.

[38] R. Boulifa and E. Madelaine. Preuve de propriétés de comportement de programmes proactive. Technical Report RR-4460, INRIA, May 2002. in french.

[39] R. Boulifa and E. Madelaine. Finite model generation for distributed Java programs. In *Workshop on Model-Checking for Dependable Software-Intensive Systems*, San-Francisco, June 2003. North-Holland.

[40] R. Boulifa and E. Madelaine. Model generation for distributed Java programs. In E. Astesiano N. Guelfi and G. Reggio, editors, *Workshop on Scientific Engineering of Distributed Java ApplIcations*, Luxembourg, November 2003. Springer-Verlag, LNCS 2952.

[41] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the Unified Modeling Language. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 – Object-Oriented Programming, 11th European Conference*, volume 1241 of *LNCS*, pages 344–366. Springer, 1997.

[42] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[43] E. Bruneton, T. Coupaye, and J. Stefani. Recursive and dynamic software composition with sharing. Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02), June 2002.

[44] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[45] A. Fantechi C. Carrez and E. Najm. Behavioural contracts for a sound assembly of components. In Springer-Verlag, editor, *in proceedings of FORTE'03*, volume LNCS 2767, November 2003.

[46] D. Caromel, L. Henrio, and B. Serpette. Asynchronous and deterministic objects. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–134. ACM Press, 2004.

[47] D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency Practice and Experience*, 10(11–13):1043–1061, November 1998.

[48] J. L. Lions (Chairman). Ariane 5 flight 105 inquiry board report. Technical report, European Space Agency, July 1996. available at http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf.

[49] S. C. Cheung and J. Kramer. Checking subsystem safety properties in compositional reachability analysis. In *ICSE '96: Proceedings of the 18th International Conference on Software Engineering*, pages 144–154, Washington, DC, USA, 1996. IEEE Computer Society.

[50] S. C. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 5:334–377, October 1996.

[51] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[52] E. M. Clarke, J. M. Wing, R. Alur, R. Cleaveland, D. Dill, A. Emerson, S. Garland, S. German, J. Guttag, A. Hall, T. Henzinger, G. Holzmann, C. Jones, R. Kurshan, N. Leveson, K. McMillan, J. Moore, D. Peled, A. Pnueli, J. Rushby, N. Shankar, J. Sifakis, P. Sistla, B. Steffen, P. Wolper, J. Woodcock, and P. Zave. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[53] R. Cleaveland and J. Riely. Testing-based abstractions for value-passing systems. In *International Conference on Concurrency Theory (CONCUR)*, volume 836 of *Lecture Notes in Computer Science*, pages 417–432. Springer, 1994.

[54] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from java source code. *Int. Conference on Software Engineering (ICSE)*, 2000.

[55] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Form. Methods Syst. Des.*, 1(2-3):275–288, 1992.

[56] P. Cousot. Automatic verification by abstract interpretation, invited tutorial. In L.D. Zuck, P.C. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Proceedings of the Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2003)*, pages 20–24, Courant Institute, NYU, New York, N.Y., USA, January 2003. LNCS 2575, Springer, Berlin.

[57] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.

[58] P. Cousot and R. Cousot. Software analysis and model checking. In E. Brinksma and K.G. Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification, CAV 2002*, Copenhagen, Denmark, LNCS 2404, pages 37–56. Springer-Verlag Berlin Heidelberg, July 2002.

[59] R. de Neufville. The baggage system at Denver: prospects and lessons. *Journal of Air Transport Management*, I(4):229–236, December 1994.

[60] R. De Nicola and F.W. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science*, volume 469 of *LNCS*, La Roche Posay, France, 1990. Springer.

[61] R. de Simone. High level devices in Meije-SCCS. *Theoretical Computer Science*, 40, 1985.

[62] R. de Simone. Higher-level synchronising devices in Meije-SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[63] C. Demartini, R. Losif, and R. Sisto. dSPIN: A dynamic extension of SPIN. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 261–276, London, UK, 1999. Springer-Verlag.

[64] L. G. DeMichiel. Enterprise JavaBeans Specification, version 2.1. Technical report, Sun Microsystems, November 2003.

[65] Gobierno de Chile, Servicio de Impuestos Internos, Factura Electrónica. https://palena.sii.cl/cvc/dte/menu.html.

[66] Y. Dumond, D. Girardet, and F. Oquendo. A relationship between sequence and statechart diagrams, 2000.

[67] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. 21st International Conference on Software Engineering*, pages 411–420. IEEE Computer Society Press, ACM Press, 1999.

[68] Jr. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[69] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1985. ISBN 0387137181.

[70] E. A. Emerson and C. L. Lei. Efficient model checking in fragments of the propositional μ–calculus. In *Symposion on Logic in Computer Science*, pages 267–278, Washington, D.C., USA, June 1986. IEEE Computer Society Press.

[71] V. Emery. The pentium chip story: A learning experience. Technical report, Vince Emery Productions, 1996. available at http://www.emery.com/1e/pentium.htm.

[72] U. Engberg and M. Nielsen. A calculus of communicating systems with label-passing. Technical Report DAIMI PB-208, University of Aarhus, 1986.

[73] A. Evans, R. B. France, K. Lano, and B. Rumpe. Developing the UML as a formal modelling notation. In *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France*, pages 297–307.

[74] J. C. Fernandez, A. Kerbrat, and L. Mounier. Symbolic equivalence checking. In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 85–96, London, UK, 1993. Springer-Verlag.

[75] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32, Providence, 1967. American Mathematical Society.

[76] Formal Systems (Europe) Ltd., Oxford, England. *Failures Divergence Refinement: User Manual and Tutorial*, 1.2β edition, October 1992.

[77] H. Garavel. OPEN/CAESAR: An open software architecture for verification, simulation, and testing. Technical Report RR-3352, INRIA, Institut National de Recherche en Informatique et en Automatique.

[78] H. Garavel. Compilation of LOTOS abstract data types. In Son T. Vuong, editor, *Proc. 2nd International Conference on Formal Description Techniques (FORTE'89)*, Amsterdam, December 1989. Elsevier (North-Holland).

[79] H. Garavel and F. Lang. SVL: a scripting language for compositional verification. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea)*, pages 377–392. IFIP, Kluwer Academic Publishers, August 2001. Full version available as INRIA Research Report RR-4223.

[80] H. Garavel and F. Lang. NTIF: A general symbolic model for communicating sequential processes with data. In *Proceedings of FORTE'02 (Houston)*. LNCS 2529, November 2002.

[81] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, August 2002.

[82] H. Garavel, R. Mateescu, and I. M. Smarandache. Parallel state space construction for model-checking. In Matthew B. Dwyer, editor, *SPIN*, volume 2057 of *Lecture Notes in Computer Science*, pages 217–234. Springer, 2001.

[83] H. Garavel and J. Sifakis. Compilation and verification of LOTOS specifications. In Logrippo, R. L. Probert, and H. Ural, editors, *Proc. 10th International Symposium on Protocol Specification, Testing and Verification*, Amsterdam, June 1990. IFIP, Elsevier (North-Holland).

[84] H. Garavel, C. Viho, and M. Zendri. System design of a CC-NUMA multiprocessor architecture using formal specification, model-checking, co-simulation, and test generation. *International Journal on Software Tools for Technology Transfer (STTT)*, 3(3):314–331, August 2001.

[85] R. Gerth. Concise PROMELA reference, 1997. available at http://spinroot.com/spin/Man/Quick.html.

[86] D. Giannakopoulou, J. Kramer, and S. Chi Cheung. Behaviour analysis of distributed systems using the tracta approach. *Automated Software Engg.*, 6(1):7–35, 1999.

[87] S. Gnesi, E. Madelaine, and G. Ristori. An exercise in protocol verification. In T. Bolognesi, E. Brinksma, and C. Vissers, editors, *Third Lotosphere Workshop and Seminar*, Pisa, September 1992.

[88] J.F. Groote and A. Ponse. Proof theory for μCRL: a language for processes with data. In Andrews et al., editors, *Proceedings of the International Workshop on Semantics of Specification Languages*, Workshops in Computing Series, pages 231–250. Springer Verlag, 1994.

[89] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, 1994.

[90] A. Hall. Seven myths of formal methods. *IEEE Softw.*, 7(5):11–19, 1990.

[91] M. Hennessy. Acceptance trees. *Journal of the ACM*, 32(4):896–928, October 1985.

[92] M. Hennessy. *Algebraic Theory of Processes*. The MIT Press, Cambridge, Mass., 1988.

[93] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138(2):353–389, February 1995.

[94] M. G. Hinchey and J. Bowen, editors. *Applications of Formal Methods*. Prentice Hall International, 1995. isbn 0-13-366949-1.

[95] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[96] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[97] G. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[98] G. Holzmann. *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, 2003. ISBN 0-321-22862-6.

[99] G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. 1996. American Mathematical Society, DIMACS/39, August 1996.

[100] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second edition, 2001. ISBN: 0-201-44124-1.

[101] T. Huckle. Collection of software bugs. Technical report, Institut für Informatik, 2004.

[102] ISO: Information Processing Systems. Basic reference model for open systems interconnection. ISO 7498, 1983.

[103] ISO: Information Processing Systems - Open Systems Interconection. Estelle - a formal description technique based on an extended state transition model. ISO 9074, 1987.

[104] ISO: Information Processing Systems - Open Systems Interconection. LOTOS - a formal description technique based on the temporal ordering of observational behaviour. ISO 8807, August 1989.

[105] ISO/IEC: Information Processing Systems - Open Systems Interconection. Enhancements to LOTOS (E-LOTOS). ISO 15437, 2001.

[106] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 40, 1985.

[107] A. Lakas. *Les Transformations Lotomaton : une contribution à la préimplémentation des systèmes LOTOS*. PhD thesis, Univ. Paris VI, June 1996.

[108] F. Lang. Compositional verification using SVL scripts. In J. Katoen and P. Stevens, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2002 (Grenoble, France)*, volume 2280, pages 465–469, April 2002.

[109] F. Lang. EXP.OPEN 2.0: A flexible tool integrating partial order, compositional, and on-the-fly verification methods. In J. van de Pol, J. Romijn, and G. Smith, editors, *Proceedings of the 5th International Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands)*, November 2005. Full version available as INRIA Research Report RR-5673.

[110] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[111] H. Lin. Symbolic transition graph with assignment. In U. Montanari and V. Sassone, editors, *CONCUR '96*, Pisa, Italy, August 1996. LNCS 1119.

[112] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Form. Methods Syst. Des.*, 6(1):11–44, 1995.

[113] N. Lynch and M. Tuttle. An introduction to Input/Output Automata. Technical Report MIT/LCS/TM-373, MIT, Cambridge, Massachusetts, November 1988.

[114] S. Graf M. Bozga and L. Mounier. IF2.0: A validation environment for component-based real-time systems. In *proceedings of CAV'02*, volume 2404 of *LNCS*, Copenhagen, July 2002.

[115] R. Mateescu M. Cornejo, H. Garavel and N. De Palma. Specification and verification of a dynamic reconfiguration protocol for agent-based applications. In Z. Mossurska A. Laurentowski, J. Kosinski and R. Ruchala, editors, *Proceedings of the 3rd IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems DAIS'2001 (Krakow, Poland)*, pages 229–242. IFIP, Kluwer Academic Publishers, September 2001. Full version available as INRIA Research Report RR-4222.

[116] M. Fisher and R. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.

[117] E. Madelaine. Verification tools from the CONCUR project. *EATCS Bull.*, 47, 1992.

[118] E. Madelaine and R. de Simone. The FC2 reference manual, 1993. available by ftp from `ftp-sop.inria.fr/meije/verif/fc2.userman.ps`.

[119] E. Madelaine and D. Vergamini. Specification and verification of a sliding window protocol. In *FORTE'91 conference*, Sydney, 1991. North-Holland.

[120] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.

[121] J. Magee and J. Kramer. Dynamic structure in software architectures. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14, New York, NY, USA, 1996. ACM Press.

[122] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. Wiley, 1999. ISBN: 0-471-98710-7.

[123] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[124] C. Massols. Outils d'analyse statique et de vérification pour les applications java distribuées. Technical Report rapport de stage MIAGE, UNSA, September 2003. in french.

[125] R. Mateescu. *Vérification des propriétés temporelles des programmes parallèles.* PhD thesis, Institut National Polytechnique de Grenoble - INPG, April 1998. Thèses d'informatique; Thèses de mathématiques.

[126] R. Mateescu. A generic on-the-fly solver for alternation-free boolean equation systems. In H. Garavel and J. Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 81–96. Springer, 2003.

[127] R. Mateescu. Logiques temporelles basées sur actions pour la vérification des systèmes asynchrones. Technical Report RR-5032, INRIA, December 2003. in french.

[128] R. Mateescu. On-the-fly verification using cadp. *Electr. Notes Theor. Comput. Sci.*, 80, 2003.

[129] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. In S. Gnesi, I. Schieferdecker, and A. Rennoch, editors, *Proceedings of the 5th Int. Workshop on Formal Methods for Industrial Critical Systems FMICS'2000 (Berlin, Germany)*, GMD Report 91, pages 65–86, Berlin, April 2000.

[130] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989. ISBN 0-13-114984-9.

[131] R. Milner. Operational and algebraic semantics of concurrent processes. pages 1201–1242, 1990.

[132] R. Milner. *Communications and mobile systems: the π-calculus.* Cambridge University Press, 1999. ISBN 0 521 64320 1 (hc.) 0 521 65869 1 (pbk.).

[133] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Inf. Comput.*, 100(1):1–77, 1992.

[134] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149–171, 1993.

[135] E. Najm, A. Lakas, A. Serouchni, E. Madelaine, and R. de Simone. ALTO: an interactive transformation tool for LOTOS and LOTOMATON. In T. Bolognesi, E. Brinksma, and C. Vissers, editors, *Third Lotosphere Workshop and Seminar*, Pisa, September 1992.

[136] K. Ng, J. Kramer, J. Magee, and N. Dulay. A visual approach to distributed programming. In *Tools and Environments for Parallel and Distributed Systems*, pages 7–31. Kluwer Academic Publishers, February 1996.

[137] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1–2):83–133, November 1984.

[138] M. Nivat. Sur la synchronisation des processus. *Rev. Techn. Thomson-CSF*, (11):899–919, 1979.

[139] M. Nivat and A. Arnold. Comportements de processus. In *AFCET Les Mathématiques de l'Informatique*, pages 35–68, 1982.

[140] Object Management Group. *UML 2.0 Object Constraint Language (OCL) Specification*, formal/03-10-14 edition, 2003. version 2.0.

[141] Object Management Group. *The Common Object Request Broker Architecture (CORBA): Core Specification*, March 2004. version 3.0.3.

[142] Object Management Group. *Unified Modeling Language: Superstructure*, formal/05-07-04 edition, August 2005. version 2.0.

[143] J. Parrow. *Handbook of Process Algebra*, chapter 8, pages 479–543. Elsevier, 2001.

[144] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer-Verlag.

[145] F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proc. Fourth International Conf. Configurable Distributed Systems (ICCDS'98)*, pages 42–52. IEEE CS Press, May 1998.

[146] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11), November 2002.

[147] A. Rausch. Towards a software architecture specification language based on UML and OCL. In *Workshop on Describing Software Architecture with UML, 23rd International Conference on Software Engineering*, Toronto, Canada, 2001.

[148] J. Riely. *Applications of Abstraction for Concurrent Programs*. PhD thesis, University of North Carolina at Chapel Hill, 1999.

[149] D. Sangiorgi and D. Walker. Π-*Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.

[150] Ri. Sharpe. Formal methods start to add up once again. Technical report, Computing, vnu bussiness publications, January 2004.

[151] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical Report 7007.011, National Institute of Standards and Technology (NIST), May 2002.

[152] F. Tronel, F. Lang, and H. Garavel. Compositional verification using CADP of the ScalAgent deployment protocol for software components. In *6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems FMOODS'2003*, Paris, France, November 2003.

[153] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, 1994.

[154] Verification of models for distributed communicating components, with safety and security (VERCORS). http://www-sop.inria.fr/oasis/Vercors.

[155] J. M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–23, 1990.

Formal specification and verification of distributed component systems

Abstract

A component is a self contained entity that interacts with its environment through well-defined interfaces. The component library Fractive provides high level primitives and semantics for programming Java applications with distributed, asynchronous and hierarchical components. It also provides a separation between functional and non-functional aspects, the latter allows the execution control of a component and its dynamic evolution.

In this thesis, we provided a formal framework to ensure that the applications built from Fractive components are safe. Safe, in the sense that each component must be adequate to its assigned role within the system, and the update or replacement of a component should not cause deadlocks or failures to the system. We introduced a new intermediate format extending the networks of communicating automata, by adding parameters to their communication events and processes.

Then, we used this intermediate format to give behavioural specifications of Fractive applications. We assumed the models of the primitive components as known (given by the user or via static analysis). Using the component description, we built a controller describing the component's non-functional behaviour. The semantics of a component is then generated as the synchronisation product of: its LTSs sub-components and the controller. The resulting system can be checked against requirements expressed in a set of temporal logic formulas, as illustrated in the thesis report.

Spécification et Vérification formelles des Systèmes de Composants Répartis

Résumé

Un composant est une entité autonome qui interagit avec son environnement par des interfaces correctement spécifiées. Fractive est une implantation du modèle de composants Fractal qui propose des primitives de haut niveau et une sémantique pour la programmation à base de composants Java distribués, asynchrones et hiérarchiques. Fractive propose également une séparation entre aspects fonctionnels et non-fonctionnels, ces derniers permettant un contrôle de l'exécution d'un composant et de son évolution dynamique.

Dans cette thèse, nous proposons un outillage formel pour la vérification d'applications construites avec Fractive. Cela permet de vérifier que chaque composant remplit correctement le rôle qui lui a été assigné au sein du système, et que la mise à jour ou le remplacement d'un composant n'engendre pas d'interblocage ou de panne du système. Nous avons défini un nouveau format intermédiaire qui étend les réseaux d'automates communicants, en paramétrisant leurs événements de communication et de traitement.

Nous avons ensuite utilisé ce format intermédiaire pour définir les spécifications comportementales d'applications Fractive. Nous considérons que les modèles des composants primitifs sont connus (donnés par l'utilisateur ou par analyse statique). En utilisant la description des composants, nous construisons un contrôleur décrivant le comportement non fonctionnel du composant. La sémantique d'un composant est ensuite générée comme le produit de synchronisation des LTSs de ses sous-composants et du contrôleur. Le système résultant peut être vérifié par rapport aux besoins exprimés dans un ensemble de formules de logique temporelle, comme illustré dans le manuscrit.