Design, vérification et implémentation de systèmes à composants

Sophie Quinton

To cite this version:

Sophie Quinton. Design, vérification et implémentation de systèmes à composants. Mathématiques générales [math.GM]. Université de Grenoble, 2011. Français. NNT : 2011GRENM002 . tel-00685854

HAL Id: tel-00685854
https://theses.hal.science/tel-00685854v1

Submitted on 6 Apr 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Sophie Quinton

Thèse dirigée par **Susanne Graf**

préparée au sein du laboratoire **VERIMAG**
et de l'école doctorale **MSTII**

Design, vérification et implémentation de systèmes à composants

Thèse soutenue publiquement le **21 janvier 2011**,
devant le jury composé de :

Mr Jean-Bernard Stefani
Directeur de recherches, Président
Mr Albert Benveniste
Directeur de recherches, Rapporteur
Mr Roberto Passerone
Assistant professor, Rapporteur
Mr Kim Larsen
Professor, Examinateur
Mr Stavros Tripakis
Research scientist, Examinateur
Mme Susanne Graf
Directrice de recherches, Directrice de thèse

Table of contents

Table of contents **3**

Introduction (en français) **7**

Introduction (in English) **19**

I Contract-Based Design and Verification of Component-Based Systems **41**

1 Preliminaries and related work **43**

1.1 Preliminaries 43
1.1.1 Labeled transition systems 43
1.1.2 Modal transition systems 48
1.2 Related work 49
1.2.1 The BIP framework 49
1.2.2 Interface theories 57

2 Defining contract frameworks **61**

2.1 Methodology 61
2.2 Definitions 63
2.2.1 Component framework 63
2.2.2 Contract framework 64
2.2.3 Dominance 67
2.3 Reasoning within a contract framework 68
2.3.1 Compositionality 69
2.3.2 Circular reasoning 70
2.3.3 A sufficient condition for dominance 73
2.4 Verifying systems of arbitrary size 73
2.4.1 Formal methodology 73
2.4.2 An application to resource sharing in a network 75
2.5 Proofs 77

3 Beyond the definitions **79**
3.1 Possible extensions 79
3.1.1 Structuring 80
3.1.2 Projection 82
3.1.3 Equivalence of glues 83
3.1.4 Defining glues on a partition 84
3.1.5 Well-formedness 85
3.2 Additional notions 86
3.2.1 Consistency 86
3.2.2 Compatibility 87
3.2.3 Composition of contracts 91
3.2.4 Multiple contracts for components 92
3.3 Combining two refinement relations 93
3.3.1 Relaxing assume-guarantee reasoning 94
3.3.2 Relaxing circular reasoning 94
3.4 Proofs 95

4 A contract framework for the BIP semantic level **97**
4.1 Necessary ingredients for a successful encoding 98
4.2 The BIP semantic contract framework 99
4.3 Coherence conditions 101
4.3.1 Composition of glues and equivalence of components 101
4.3.2 Structuring systems 102
4.3.3 Consistency between $\sqsubseteq$ and $\preccurlyeq$ 102
4.3.4 Preservation of refinement by composition 103
4.3.5 Soundness of circular reasoning 103

5 Two component frameworks for BIP **105**
5.1 A first variant: BIP with maximal progress 107
5.2 A second variant: multi-shot BIP 113
5.3 Projection 117

6 Application to I/O automata and to the SPEEDS project **119**
6.1 Encoding of interface I/O automata 119
6.1.1 The I/O contract framework 120
6.1.2 Coherence conditions 121
6.1.3 Using the I/O contract framework 124
6.2 The SPEEDS project 127
6.2.1 The L0 contract framework 128
6.2.2 The L1 contract framework 132
6.2.3 Consistency between L0 and L1 133
6.2.4 Implementation issues 134

6.2.5 Proofs 136

7 Contract frameworks for transition systems 141

7.1 Labeled transition systems 141
7.1.1 Definitions 141
7.1.2 Refinement in any context 144
7.1.3 Structural consistency 144
7.2 Modal transition systems 145
7.2.1 Definitions 145
7.2.2 Refinement in any context 146
7.2.3 Structural consistency 146
7.3 Labeled transition systems with priorities 148
7.3.1 Definitions 148
7.4 Modal transition systems with priorities 149
7.4.1 Definitions 149
7.5 Proofs 152
7.6 Conclusion 159
7.6.1 Summary 159
7.6.2 Perspectives 159

II Implementation of Distributed Systems with Complex Interaction 161

8 Achieving distributed control through model checking 163

8.1 Preliminaries 163
8.1.1 Petri nets 163
8.1.2 Constraints 164
8.1.3 Distributed setting 166
8.1.4 Defining properties 167
8.1.5 Knowledge 168
8.2 The support policy 169
8.2.1 Building the support table 169
8.2.2 Distributed control based on the support table 170
8.2.3 Deadlock-freedom 172

9 A synchronization-based approach 173

9.1 A synchronization-based approach 173
9.1.1 An example where the support policy fails 173
9.1.2 Existing solutions 174
9.1.3 Adding synchronizations to provide sufficient knowledge 175
9.1.4 A distributed controller imposing the global property 176
9.1.5 Minimizing the number of coordinators 177

9.2 Implementation and experimental results 179
9.2.1 The pragmatic dining philosophers 179
9.2.2 Of tracks and trains 182

10 Reducing the need for additional synchronizations 187
10.1 Alternative to adding synchronizations 188
10.1.1 Support policy based on the controlled system 188
10.1.2 Controllers based on an incomplete support table 191
10.2 Comparison with existing work 193
10.2.1 History-based controllers 193
10.2.2 A practical solution to the distributed control problem 194
10.3 Conclusion 196
10.3.1 Summary 196
10.3.2 Perspectives 198

Conclusion 199

Bibliography 201

Introduction (en français)

Contexte et motivation

L'informatique fait partie de notre quotidien depuis maintenant plusieurs décennies. Elle a bouleversé le fonctionnement de notre société. Les ordinateurs sont partout autour de nous, depuis les téléphones portables jusqu'aux avions, et ils font tout pour nous : ils conduisent, jouent de la musique, ils nous permettent de communiquer avec le reste du monde, nous donnent accès à un savoir infini... Récemment, les progrès considérables dans les capacités de nos ordinateurs ont été surpassés seulement par la demande de systèmes toujours plus sophistiqués. Développer de tels systèmes à la fois larges et complexes est un challenge. À cette fin, les *designers* appliquent une méthode qui a prouvé son utilité dans divers contextes à travers les âges : *diviser pour régner*. Son principe est simple mais efficace : un problème de grande taille est résolu plus facilement s'il est divisé en plusieurs sous-problèmes de plus petite taille, qui peuvent à leur tour être décomposés, et ainsi de suite jusqu'à obtenir des sous-problèmes suffisamment petits pour être résolus directement. Une fois ces sous-problèmes résolus, la solution du problème de départ est obtenue par composition des solutions des sous-problèmes. Dans l'univers du *design* de systèmes, cette approche est appelée *design à base de composants*. La solution du problème est un composant, qui est soit assez petit pour être construit directement — on l'appelle alors composant *atomique* — soit obtenu en composant des composants plus petits — on l'appelle alors composant *hiérarchique*. Ainsi, le problème de la construction de systèmes complexes de grande taille est résolu en écrivant des composants atomiques et en les assemblant pour former des composants hiérarchiques de plus en plus complexes jusqu'à obtenir le système voulu.

Un système construit de manière simple et claire est moins susceptible de contenir des erreurs. De plus, cela peut améliorer considérablement les résultats obtenus lors de la phase de vérification

ultérieure. Pour cette raison, il existe une grande variété de formalismes pour développer des systèmes à base de composants. Les langages synchrones comme LUSTRE [CPHP87] ou Signal [BGJ91] ont été transférés avec succès du monde académique à l'industrie. Les langages de modélisation de systèmes tels que SystemC [Sys] et Simulink [Sim] sont également largement utilisés dans l'industrie. Des *frameworks* comme Ptolemy [EJL$^+$03] et 42 [MB07] s'attachent à gérer l'hétérogénéité en permettant de combiner plusieurs modèles de calcul et communication.

Récemment, de nombreux langages sont apparus dans lesquels l'interaction entre composants est complexe : les connecteurs ne servent pas seulement à transférer des données mais jouent également un rôle dans la synchronisation des composants. Parmi ces formalismes on trouve le Kell calculus [BS03] et le calcul de connectors Reo [Arb04]. Nous nous intéressons particulièrement au *framework* BIP [GS05, BBS06, BS08a] developpé à Verimag. BIP est un langage dans lequel la représentation de l'interaction est assez expressive pour décrire de nombreux modèles, depuis le rendez-vous jusqu'au broadcast. De plus, à l'image des composants, il est possible de définir des connecteurs hiérarchiques, c'est-à-dire des connecteurs définis comme la composition de plusieurs connectors. Dans de tels formalismes, il est essentiel de raisonner sur la structure du système. Notre travail est motivé par le *framework* de composants HRC (*Heterogenous Rich Component*, composant riche et hétérogène), divisé en deux parties L0 et L1, défini dans le projet SPEEDS [SPE] et utilisé dans le projet COMBEST [COM]. HRC L0 a été inspiré par les langages synchrones tandis que HRC L1 a été inspiré par BIP.

Dans cette thèse, nous avons étudié comment les systèmes complexes sont *designés*, vérifiés puis implémentés. En particulier, à cause de la grande variété de formalismes existants, nous nous sommes attaché à trouver des définitions suffisamment expressives pour pouvoir inclure BIP, mais aussi assez générales pour pouvoir être appliquées à d'autres langages. Ainsi, ce document est organisé autour de deux parties :

- Part I : *Design et vérification à l'aide de contrats de large systèmes de composants.* Nous fournissons une définition de *framework* de composants qui est assez abstraite pour inclure BIP et les *frameworks* HRC L0 et L1, mais peut également s'appliquer simplement à une variété d'autres *frameworks*.
- Part II : *Implémentation de systèmes avec interaction complexe dans un contexte distribué.* Ce travail s'applique de façon naturelle à BIP et sa politique de priorités, mais nous avons généralisé la contrainte à respecter à n'importe quelle propriété de sûreté.

Design et vérification de larges systèmes

Au fur et à mesure que les systèmes croissent en taille et en complexité, le nombre d'erreurs qu'ils contiennent croît également. De plus, ces erreurs deviennent de plus en plus difficiles à détecter et réparer. Vérifier la correction d'un système est un problème si difficile que les ordinateurs sont probablement les seuls produits vendus sans la garantie d'être sans défaut.

Cependant, pour certains systèmes, cette situation n'est pas acceptable parce qu'une erreur peut causer des pertes humaines (par exemple si un avion s'écrase), ou conduire à une perte financière lourde (par exemple s'il y a une erreur de fonctionnement dans un téléphone portable produit en très grande quantité). En conséquence, des méthodes formelles sont nécessaires pour assurer la *correction* de certains systèmes, notion qui doit être définie formellement, par exemple comme étant la conformité à un ensemble de *requirements*. De tels *requirements* prennent différentes formes en fonction du domaine d'application, par exemple des pré- et post-conditions pour les appels de fonction, ou des propriétés temporelles qui se divisent entre propriétés de sûreté ("rien de mal n'arrivera jamais") et les propriétés de vivacité ("quelque chose de bien finira par arriver"). Une méthode de vérification efficace pour des systèmes complexes de grande taille doit posséder les propriétés suivantes :

- *passage à l'échelle* : l'approche doit fonctionner pour des systèmes de très grande taille
- *prédictabilité* : les erreurs de *design* doivent être détectées aussi tôt que possible lors de la phase de design
- *réutilisabilité* : il doit être possible de réutiliser des parties du processus de vérification si un composant est remplacé par un autre similaire

Comme nous nous intéressons aux systèmes critiques, tester est nécessaire pour détecter des erreurs rapidement, mais pas suffisant car cela ne fournit pas de garantie de correction. Parmi les méthodes de vérification, le *model checking* est une approche totalement automatisée basée sur l'exploration exhaustive de l'espace d'états du modèle du système, qui est en général une machine d'états finie. Malheureusement, cette méthode souffre du fameux problème de l'*explosion de l'espace d'états* : elle devient vite irréalisable si les composants s'éxécutent de façon concurrente car le nombre d'états du modèle augmente exponentiellement par rapport au nombre de composants. Le *model checking* ayant été largement étudié, de nombreuses améliorations ont été proposées afin de résoudre ce problème. Elles se répartissent entre trois catégories, qui peuvent être combinées : les techniques symboliques [McM93], les techniques à base d'abstraction [CGL94, GS97, BMMR01, CGJ^{+}00] and les méthodes compositionnelles.

Les méthodes compositionnelles (pour une présentation exhaustive, cf [dRdBH^{+}01]) sont celles qui gèrent le mieux le problème de l'explosion de l'espace d'états. Elles appliquent la méthode *diviser*

pour mieux régner pour inférer, à partir de propriétés locales aux composants, une propriété (globale) du système. Ces approches incluent le *model checking* compositionnel [CLM89, Lon93] et la minimization compositionnelle [CGL94], qui peut être guidée par les propriétés [CLM89, LGS+95] ou bien par le contexte [GSL96]. *Assume-guarantee* est une autre approche [Jon83] basée sur la décomposition du système en plusieurs morceaux dont on prouve qu'ils satisfont tous une certaine garantie à la condition que leur environnement (c'est à dire le reste du système) satisfasse une certaine hypothèse (*assumption*).

La génération automatique d'*assumptions* utilisant des algorithmes d'apprentissage a été proposée dans [CGP03]. Cependant, ces techniques ont encore des difficultés à se montrer plus efficaces que l'approche monolithique [CAC08], bien que des progrès importants aient été accomplis récemment en rendant l'apprentissage implicite [CCF+10]. Pour les systèmes BIP, l'outil D-Finder [BBSN08, BBNS09] utilise des invariants de composant ainsi que des invariants d'interaction pour prouver des propriétés de sûreté de façon compositionnelle.

Les approches compositionnelles sont performantes par rapport au problème de l'explosion de l'espace d'états. Cependant, elles n'offrent pas d'*incrémentalité* pour utiliser une propriété d'une composition de components à un niveau hiérarchique plus élevé. Certains travaux dans cette direction existent déjà : par exemple D-Finder offre désormais quelques possibilités en ce sens et [Sin07] s'intéresse aux question de substitutabilité (remplacer un composant par un autre) dans le raisonnement par *assume-guarantee*. Cependant ces résultats restent insuffisants pour un contexte industriel, où les composants sont souvent construits par des équipes différentes voire même achetés à d'autres compagnies, et où la possibilité d'intégrer des composants dans un système préexistant est essentielle. C'est la raison pour laquelle nous voulons combiner les approches compositionnelles avec une méthodologie incrémentale.

Motivation for using contracts

Les *frameworks* de contrats [BCP07, BFM+08] et d'interface [dAH01a, LNW06] émergent comme le formalisme de choix lorsque les systèmes sont *designés* par de grandes équipes réparties en sous-équipes indépendantes, or lorsque la chaîne d'approvisionnement est complexe [Dam05, SPE]. Un des clés du raisonnement par contrats est que ces derniers peuvent être utilisés aussi bien pour le *design* que pour la vérification. Les contrats sont des contraintes de *design* sur les implémentations qui sont maintenues tout au long du cycle de vie des systèmes. Comme pour le *contrat-based design* [Mey92], nous utilisons les contrats pour contraindre, réutiliser et remplacer les implémentations.

Les contrats expriment à la fois l'*assumption* (hypothèse) faite par le component sur son environnement et la *guarantee* (garantie) concernant le comportement attendu du composant. Dans le context du *design* de *programmes*, par exemple dans les méthodologies de *design* orientées objet, les contrats sont généralement de simples pré- et post-conditions, comme dans [Mey92]. Dans le context du *design* de *systèmes*, des contrats plus expressifs sont nécessaires, par exemple pour spécifier des propriétés temporelles de sûreté et de vivacité. C'est le rôle des spécifications d'interface [GSL96, dAH01a].

Le raisonnement à base de contrats utilise pleinement la notion d'incrementalité. En effet, les contrats fournissent pour les composants une abstraction qui peut être utilisée pour la composition et la réutilisation. En particulier, parce qu'un contrat exprime des contraintes sur l'environnement dans lequel un composant peut être utilisé, il est possible de raisonner sur des systèmes *fermés* (i.e., qui ne peuvent plus être composés) plutôt que sur des systèmes *ouverts* qui peuvent être utilisés dans n'importe quel environnement. D'autre part, le raisonnement à base de contrats permet l'implémentabilité indépendante : dans une approche de *design* du haut vers le bas, un contrat est écrit pour chaque sous-composant du système, et ces contrats peuvent être raffinés indépendemment les uns des autres. De plus, comme les contrats sont écrits à chaque niveau de hiérarchie du système, ils évitent le problème de l'explosion de l'espace d'états. Enfin, cette approche peut toujours être combinée avec des approches compositionnelles à chaque niveau de hiérarchie.

Le projet SPEEDS proposait d'utiliser les contrats pour prouver des propriétés du *framework* de composants HRC. À la fois pour HRC L0 et HRC L1, des théories de contrats devaient être dévelopées. À la fin, ces théories devaient être unifiées pour combiner les résultats de vérification de leur chaînes d'outils respectives, qui sont basées sur des relations de raffinement différentes. Nous avons montré que prouver des propriétés de systèmes construits à partir de composants L0 et L1 ne nécessite pas un *framework* sémantique unificateur ; l'intégration des *frameworks* de composition dans un *framework* unificateur, comme présenté ici, suffit.

Contribution

Notre but n'est pas de proposer un nouveau *framework* générique de *design* mais plutôt de définir un ensemble minimal de propriétés qui doivent être satisfaites par une théorie de contrats pour permettre certaines règles de preuves. Nous rendons explicites certaines questions auxquelles doit répondre quelqu'un qui définit un *framework* de contrats ou qui se demande quel *framework* utiliser. Par exemple, est-ce que la règle de preuve qu'on appelle raisonnement circulaire est correcte dans mon cadre ? Est-il possible de décomposer un composant hiérarchique selon n'importe quelle partition de ses

sous-composants ? Répondre à ces questions offre des réponses sur l'applicabilité de raisonnements incrémentaux et compositionnels plus sophistiqués. Ainsi, notre approche vise à séparer la question des propriétés qui dépendent du *framework* utilisé de celles qui en sont indépendantes et résultent de règles de preuve génériques.

Notre définition de *framework* de composant est très abstraite : nos opérateurs de composition (que nous appelons *glues*) permettent de représenter une grande variété de modèles d'interaction depuis la communication input/output (I/O) jusqu'aux connecteurs BIP avec politique de priorités. En particulier, cela nous permet de travailler avec des *frameworks* sémantiques de bas niveau aussi bien qu'avec des *frameworks* syntaxiques. D'autre part, l'abstraction de ports, qui un ingrédient essentiel pour prouver le raffinement entre spécifications à différents niveaux de granularité, peut être considéré comme une *glue*. Nos contrats ont une partie structurelle, qui décrit la *glue* avec laquelle le composant doit être connecté à son environnement. Ainsi, dans les *frameworks* où l'interaction est complexe, nous montrons que les proproiétés structurelles et compositionnelles d'un composant peuvent être établies en s'appuyant fortement sur la structure du système et de façon moins importante sur les propriétés comportementales de l'environnement. C'est utile en particulier parce que la structure d'un système en construction peut être fixée bien avant certaines propriétés comportementales, qui doivent être raffinées tout au long du processus de *design*. Contrairement aux automates d'interface [dAH01a], mais à l'image des automates d'interface I/O [LNW06], nos contrats distinguent explicitement entre une *assumption* A représentant une propriété de l'environnement, et une guarantie G représentant une propriété que le composant en cours de *design* doit garantir dans tout environnement se comportant en accord avec A. Nous discutons ce choix dans le chapitre 2.

Munis de cette définition générique de *framework* de composant, nous nous intéressons aux relations de raffinement. Dans le *design* de systèmes, le raffinement est défini entre une spécification et une implémentation [LW94, dAH01a, Sin07]. Dans les *frameworks* de contrat, le raffinement prend plusieurs formes : raffinement par rapport à une spécification (*conformité*), raffinement entre contrats (*dominance*) and raffinement d'une implémentation par rapport à un contrat (*satisfaction*). Dans les théories de contrats existantes, les deux dernières sont dérivées de la première. Nous proposons une relation plus faible entre conformité and *raffinement sous contexte*, à partir de laquelle les relations de dominance et satisfaction sont dérivées. Cela nous permet d'obtenir des règles de raisonnement plus efficaces. D'autre part, nous généralisons l'usage qui est parfois fait du raffinement dans tout contexte (par contraste avec les raffinement dans un contexte donné) à des relations plus faibles, ce qui permet également d'obtenir des méthodes de raisonnement plus puissantes. Notons que nous ne discutons pas du problème de trouver des relations de raffinement appropriées : cela reste une étape

de *design* dans la construction du *framework* de composant, tout comme la spécification de contrats est une étape de *design* dans la construction de systèmes. Notre but est de faciliter le processus de *design* en spécifiant clairement quelles sont les propriétés nécessaires pour permettre certaines règles de preuve.

Nous discutons certaines règles pour le raisonnement compositionnel. En particulier, nous nous intéressons au *raisonnement circulaire* qui entraîne une règle de preuve intéressante pour prouver la dominance sans composer les contrats, ce qui est impossible dans le cas général et non souhaitable dans de nombreux *frameworks* concrets. Le *raisonnement circulaire* implique une contrainte assez forte sur la relation de raffinement sous contexte, qui peut ne pas être satisfaite par la relation disponible. Nous montrons comment relâcher cette contrainte, en particulier en combinant plusieurs relations de raffinement. Notons également que nous ne proposons pas de méthode pour construire un *framework* dans lequel le raisonnement circulaire est correct. Cette question a déja été étudiée dans [McM99, Mai03b]. En particulier, [Mai03a] prouve qu'une telle règle ne peut être à la fois correcte (*sound*) et complète. Enfin, nous présentons un ensemble d'applications qui met en avant le généralité de nos définitions. La pertinence du choix de nos définitions est mise en évidence dans ces exemples.

Organisation

- Le chapitre 1 présente des notions de base sur les systèmes de transition et leur relations de raffinement. Il décrit également trois *frameworks* qui sont souvent utilisés ou mentionnés dans le reste de la partie I.
- Le chapitre 2 présente notre méthodologie à base de contrats pour vérifier des systèmes de taille arbitraire. Elle définit nos *frameworks* de contrats et présente certaines règles pour raisonner dessus.
- Le chapitre 3 met cette méthodologie en perspective en discutant comment certains concepts habituels qui n'apparaissent pas dans nos définitions peuvent être intégrés. En particulier, nous justifions le choix d'éviter la composition de contrats dans certains *frameworks*.
- Les chapitres 4 à 7 présentent plusieurs *frameworks* de contrats, chacun mettant en avant un aspect différent de notre méthodologie. Plus précisément :
  - Le chapitre 4 présente un *framework* de contrats pour le niveau sémantique de BIP. Cet exemple simple donne une première illustration des définitions présentées dans les chapitres précédents.
  - Le chapitre 5 définit deux variantes du *framework* de composants BIP qui sont utilisées dans

le chapitre suivant. Ces variantes n'ont jamais été proposées auparavant et elles mettent en évidence l'importance d'avoir une sémantique compositionnelle cohérente avec la mise à plat de systèmes.

– Le chapitre 6 montre comment les automates d'interface I/O peuvent être simplement représentés en utilisant un *framework* de contrat pour lequel les preuves de cohérence sont techniques mais simples. Des preuves plus complexes peuvent alors être grandement simplifiées. Ensuite, nous présentons les *frameworks* HRC L0 and L1 et nous décrivons comment les résultats obtenus à partir d'outils pour L0 et L1 peuvent être utilisés ensemble grâce au raisonnement circulaire combiné.
– Le chapitre 7 présente et discute plusieurs relations de raffinement pour des *frameworks* basés sur des systèmes de transition étiquetés (LTS, *Labeled Transition Systems*) et généralise l'usage du raisonnement circulaire combiné du chapitre précédent. Un *framework* basé sur les systèmes de transition modaux (MTS, *Modal Transition Systems*) est ensuite proposé, pour lequel la cohérence structurelle est une propriété utile. Enfin, deux *frameworks* utilisant des priorités statiques sont décrits, pour les LTS puis les MTS, ce qui illustre que le raffinement dans tout contexte peut dans certains cas être strictement plus fort que la relation de conformité.

Bien que cela ne soit pas présenté dans cette thèse, nous avons également appliqué notre méthodologie à un *framework* de contrats plus complexe qui est assez expressif pour décrire des échanges de données entre composants et des propriétés de progrès. Ce *framework* a été appliqué à un algorithme pour le partage de ressources dans un réseau. Une description détaillée de ce travail est disponible dans [BHQG10c, BHQG10b].

Implémentation de systèmes distribués avec interactions complexes

Le travail présenté dans la partie II est motivé par le challenge posé par l'implémentation distribuée de systèmes BIP [BBBS08]. Ce problème peut se généraliser à la question de contrôler un système existant de façon à le forcer à satisfaire une contrainte de sûreté supplémentaire [RW92] — par exemple une politique de priorités. En pratique, cela est rendu possible par l'addition d'un processus superviseur, qui s'éxécute en synchronie avec le système. Les superviseurs sont en général des automates (d'états finis) qui observent le système et progressent selon les transitions qu'ils observent, et ont de plus la capacité de bloquer certaines transitions en fonction de leur état courant.

Ce problème est lié au problème de la synthèse à partir de spécifications temporelles (LTL, *Linear*

Temporal Logic), qui est 2EXPTIME *hard* pour les systèmes réactifs séquentiels et indécidable pour les systèmes concurrents, comme démontré par Pnueli and Rosner [PR90].

Une étude exacte de l'existence d'un contrôleur global (complètement synchronisé) peut être basée sur la théorie des jeux. De fait, on peut voir le problème comme l'implémentation d'une stratégie pour le jeu à deux joueurs suivant : un joueur, l'environnement, peut toujours choisir entre les transitions incontrôlables possibles tandis que l'autre joueur peut choisir entre les transitions contrôlables. Le but du contrôleur est de s'assurer que la contrainte est satisfaite par une éxécution choisie conjointement. Ce problème peut être résolu en utilisant des algorithmes basés sur les jeux de sûreté [Tho95].

Le problème du contrôle distributé

Maintenant, intéressons-nous à la question du contrôle distribué [RW92, YL02] : des contrôleurs locaux, un par processus ou ensemble de processus, peuvent interdire l'éxécution de certaines transitions afin d'éviter la violation de la contrainte imposée. À cause de la nature distribuée du système, chaque contrôleur a une vision limitée du système global. La synthèse de contrôleur distribué pour imposer une contrainte globale est un problème indécidable dans le cas général [Tri04, Thi05]. On peut rendre le problème décidable en réduisant la concurrence dans le système. Dans le pire des cas, il ne reste plus aucun concurrence et un contrôleur totalement global est construit. Cependant, même avec cette hypothèse assez forte, le problème de synthèse de contrôleur reste difficile à gérer. Une approche pragmatique est basée sur la vérification de properties de *connaissance* grâce auxquelles les contrôleurs locaux peuvent décider ou non de bloquer une transition. Deux approches différentes existent pour synthétiser des contrôleurs distribués, qui sont alors soit *conjonctifs* soit *disjonctifs*.

Un contrôleur *conjonctif* autorise une transition seulement si *tous* les contrôleurs locaux l'autorisent. En d'autres termes, il est suffisant pour bloquer une transition qu'un contrôleur local le décide. Le choix d'un contrôleur conjonctif est motivé par le besoin d'avoir un système qui se comporte *exactement* comme spécifié par un langage régulier donné. Dans ce cas, tous les comportements du système non contrôlé qui ne violent pas la contrainte imposée doivent être possibles. Il est alors nécessaire qu'au moins un processus sache quand une transition possible dans le système viole la contrainte, afin de la bloquer. Cette approche nécessite une connaissance suffisante de chaque processus pour permettre le maximum de transitions autorisées. Elle est suivie par [RR00], qui étudie la possibilité de construire un contrôleur utilisant de la connaissance (propriété qui s'appelle *Kripke observability* dans ce travail).

Un contrôleur *disjonctif* autorise une transition si *au moins un* contrôleur local l'autorise. Selon

cette approche, suivie par [BBPS09], il est possible d'interdire une transition même si elle ne viole pas la contrainte imposée mais l'absence de blocage doit être préservée. La construction du contrôleur est différente de celle du contrôle conjonctif car elle requiert seulement qu'au moins un processus autorise une transition. Cette approche préserve la correction du contrôleur même si la connaissance ne suffit pas à autoriser toutes les transitions qui ne violent pas la contrainte.

Dans [RR00] comme dans [BBPS09], la connaissance est utilisée comme un outil pour construire des contrôleurs distribués. À partir d'une connaissance précalculée qui reflète dans chaque état local toutes les situations possibles des autres processus, le contrôleur local d'un processus décide à l'éxécution si une action de ce processus peut être éxécutée sans violer la contrainte imposée. Parfois, cependant, la connaissance des processus individuels n'est pas suffisante. Alors, la connaissance conjointe de plusieurs processus peut être monitorée à l'aide de contrôleurs associés à des groupes de processus au lieu de processus individuels. Dans le pire des cas, un contrôleur global est construit, qui peut contrôler le système selon la contrainte. Malheureusement, cela cause la perte de la concurrence entre les processus qui sont monitorés conjointement.

Contribution

L'approche proposée ici étend l'approche de [BBPS09]. Nous supposons que les systèmes permettent l'addition de synchronisations supplémentaires. Cette hypothèse est raisonnable car il existe de toute façon un mécanisme de synchronisation pour implémenter le système de façon distribuée.

Plutôt que des synchronisations permanentes entre groupes de processus, nous proposons une méthode pour construire des contrôleurs distribués qui synchronisent les processus *temporairement*. Nous utilisons des techniques de *model checking* pour précalculer un ensemble minimal de points de synchronisations auxquels la connaissance conjointe de plusieurs processus peut être calculée durant de courtes phases de coordination. Après chaque synchronisation, les processus impliqués peuvent de nouveau progresser indépendemment les uns des autres jusqu'à ce qu'une autre synchronisation soit nécessaire.

Ces synchronisations temporaires entre plusieurs processus sont rendus possibles par l'utilisation d'un algorithme de coordination basé sur des échanges de messages, tel que α-core [PCT04]. De telles synchronisations impliquent un surcoût de communication important. En conséquence, il est essentiel de minimiser le nombre de messages résultant de ces synchronisations additionnelles. Nous obtenons des résultats dans ce domaine en réduisant le nombre de coordinateurs qui gèrent ces synchronisations.

Une des limitations de cette méthode est liée au fait que nous calculons la connaissance sur le

système que nous voulons contrôler et non sur le système contrôlé. Les processus ont alors moins de connaissance et donc il est possible que nous introduisions plus de synchronisations que nécessaire. Nous proposons des techniques qui réduisent l'impact de ce problème ainsi que comment ces nouvelles techniques peuvent combinées avec celles qui existent déjà, comme par exemple la connaissance avec mémoire parfaite. Enfin, nous montrons l'intérêt des contrôleurs à base de connaissance en tant que solution pragmatique au problème de la synthèse de contrôleur.

Organisation

- Le chapitre 8 généralise the travail de [BBPS09] à des contraintes de sûreté arbitraires.
- Le chapitre 9 présente notre méthode pour construire des contrôleurs distribués en synchronisant les processus *temporairement* ainsi que pour réduire le surcoût de communication dû aux synchronisations additionnelles. Il présente également des résultats expérimentaux.
- Le chapitre 10 décrit nos techniques pour éviter des synchronisations inutiles et comment combiner plusieurs techniques. De plus, il met en perspective les contrôleurs à base de connaissance, mettant en évidence leur intérêt.

Bien que cela ne soit pas présenté dans ce document, un travail préliminaire sur l'intégration de contraintes haut-niveau avec l'algorithme de coordination a déjà été entrepris. Des résultats sont présentés dans [BHGQ10].

Introduction (in English)

Contexte et motivation

L'informatique fait partie de notre quotidien depuis maintenant plusieurs décennies. Elle a bouleversé le fonctionnement de notre société. Les ordinateurs sont partout autour de nous, depuis les téléphones portables jusqu'aux avions, et ils font tout pour nous : ils conduisent, jouent de la musique, ils nous permettent de communiquer avec le reste du monde, nous donnent accès à un savoir infini... Récemment, les progrès considérables dans les capacités de nos ordinateurs ont été surpassés seulement par la demande de systèmes toujours plus sophistiqués. Développer de tels systèmes à la fois larges et complexes est un challenge. À cette fin, les *designers* appliquent une méthode qui a prouvé son utilité dans divers contextes à travers les âges : *diviser pour régner*. Son principe est simple mais efficace : un problème de grande taille est résolu plus facilement s'il est divisé en plusieurs sous-problèmes de plus petite taille, qui peuvent à leur tour être décomposés, et ainsi de suite jusqu'à obtenir des sous-problèmes suffisamment petits pour être résolus directement. Une fois ces sous-problèmes résolus, la solution du problème de départ est obtenue par composition des solutions des sous-problèmes. Dans l'univers du *design* de systèmes, cette approche est appelée *design à base de composants*. La solution du problème est un composant, qui est soit assez petit pour être construit directement — on l'appelle alors composant *atomique* — soit obtenu en composant des composants plus petits — on l'appelle alors composant *hiérarchique*. Ainsi, le problème de la construction de systèmes complexes de grande taille est résolu en écrivant des composants atomiques et en les assemblant pour former des composants hiérarchiques de plus en plus complexes jusqu'à obtenir le système voulu.

Un système construit de manière simple et claire est moins susceptible de contenir des erreurs. De plus, cela peut améliorer considérablement les résultats obtenus lors de la phase de vérification

ultérieure. Pour cette raison, il existe une grande variété de formalismes pour développer des systèmes à base de composants. Les langages synchrones comme LUSTRE [CPHP87] ou Signal [BGJ91] ont été transférés avec succès du monde académique à l'industrie. Les langages de modélisation de systèmes tels que SystemC [Sys] et Simulink [Sim] sont également largement utilisés dans l'industrie. Des *frameworks* comme Ptolemy [EJL+03] et 42 [MB07] s'attachent à gérer l'hétérogénéité en permettant de combiner plusieurs modèles de calcul et communication.

Récemment, de nombreux langages sont apparus dans lesquels l'interaction entre composants est complexe : les connecteurs ne servent pas seulement à transférer des données mais jouent également un rôle dans la synchronisation des composants. Parmi ces formalismes on trouve le Kell calculus [BS03] et le calcul de connectors Reo [Arb04]. Nous nous intéressons particulièrement au *framework* BIP [GS05, BBS06, BS08a] developpé à Verimag. BIP est un langage dans lequel la représentation de l'interaction est assez expressive pour décrire de nombreux modèles, depuis le rendez-vous jusqu'au broadcast. De plus, à l'image des composants, il est possible de définir des connecteurs hiérarchiques, c'est-à-dire des connecteurs définis comme la composition de plusieurs connectors. Dans de tels formalismes, il est essentiel de raisonner sur la structure du système. Notre travail est motivé par le *framework* de composants HRC (*Heterogenous Rich Component*, composant riche et hétérogène), divisé en deux parties L0 et L1, défini dans le projet SPEEDS [SPE] et utilisé dans le projet COMBEST [COM]. HRC L0 a été inspiré par les langages synchrones tandis que HRC L1 a été inspiré par BIP.

Dans cette thèse, nous avons étudié comment les systèmes complexes sont *designés*, vérifiés puis implémentés. En particulier, à cause de la grande variété de formalismes existants, nous nous sommes attaché à trouver des définitions suffisamment expressives pour pouvoir inclure BIP, mais aussi assez générales pour pouvoir être appliquées à d'autres langages. Ainsi, ce document est organisé autour de deux parties :

- Part I : *Design et vérification à l'aide de contrats de large systèmes de composants.* Nous fournissons une définition de *framework* de composants qui est assez abstraite pour inclure BIP et les *frameworks* HRC L0 et L1, mais peut également s'appliquer simplement à une variété d'autres *frameworks*.
- Part II : *Implémentation de systèmes avec interaction complexe dans un contexte distribué.* Ce travail s'applique de façon naturelle à BIP et sa politique de priorités, mais nous avons généralisé la contrainte à respecter à n'importe quelle propriété de sûreté.

Design et vérification de larges systèmes

Au fur et à mesure que les systèmes croissent en taille et en complexité, le nombre d'erreurs qu'ils contiennent croît également. De plus, ces erreurs deviennent de plus en plus difficiles à détecter et réparer. Vérifier la correction d'un système est un problème si difficile que les ordinateurs sont probablement les seuls produits vendus sans la garantie d'être sans défaut.

Cependant, pour certains systèmes, cette situation n'est pas acceptable parce qu'une erreur peut causer des pertes humaines (par exemple si un avion s'écrase), ou conduire à une perte financière lourde (par exemple s'il y a une erreur de fonctionnement dans un téléphone portable produit en très grande quantité). En conséquence, des méthodes formelles sont nécessaires pour assurer la *correction* de certains systèmes, notion qui doit être définie formellement, par exemple comme étant la conformité à un ensemble de *requirements*. De tels *requirements* prennent différentes formes en fonction du domaine d'application, par exemple des pré- et post-conditions pour les appels de fonction, ou des propriétés temporelles qui se divisent entre propriétés de sûreté ("rien de mal n'arrivera jamais") et les propriétés de vivacité ("quelque chose de bien finira par arriver"). Une méthode de vérification efficace pour des systèmes complexes de grande taille doit posséder les propriétés suivantes :

– *passage à l'échelle* : l'approche doit fonctionner pour des systèmes de très grande taille
– *prédictabilité* : les erreurs de *design* doivent être détectées aussi tôt que possible lors de la phase de design
– *réutilisabilité* : il doit être possible de réutiliser des parties du processus de vérification si un composant est remplacé par un autre similaire

Comme nous nous intéressons aux systèmes critiques, tester est nécessaire pour détecter des erreurs rapidement, mais pas suffisant car cela ne fournit pas de garantie de correction. Parmi les méthodes de vérification, le *model checking* est une approche totalement automatisée basée sur l'exploration exhaustive de l'espace d'états du modèle du système, qui est en général une machine d'états finie. Malheureusement, cette méthode souffre du fameux problème de l'*explosion de l'espace d'états* : elle devient vite irréalisable si les composants s'éxécutent de façon concurrente car le nombre d'états du modèle augmente exponentiellement par rapport au nombre de composants. Le *model checking* ayant été largement étudié, de nombreuses améliorations ont été proposées afin de résoudre ce problème. Elles se répartissent entre trois catégories, qui peuvent être combinées : les techniques symboliques [McM93], les techniques à base d'abstraction [CGL94, GS97, BMMR01, CGJ+00] and les méthodes compositionnelles.

Les méthodes compositionnelles (pour une présentation exhaustive, cf [dRdBH+01]) sont celles qui gèrent le mieux le problème de l'explosion de l'espace d'états. Elles appliquent la méthode *diviser*

pour mieux régner pour inférer, à partir de propriétés locales aux composants, une propriété (globale) du système. Ces approches incluent le *model checking* compositionnel [CLM89, Lon93] et la minimization compositionnelle [CGL94], qui peut être guidée par les propriétés [CLM89, LGS+95] ou bien par le contexte [GSL96]. *Assume-guarantee* est une autre approche [Jon83] basée sur la décomposition du système en plusieurs morceaux dont on prouve qu'ils satisfont tous une certaine garantie à la condition que leur environnement (c'est à dire le reste du système) satisfasse une certaine hypothèse (*assumption*).

La génération automatique d'*assumptions* utilisant des algorithmes d'apprentissage a été proposée dans [CGP03]. Cependant, ces techniques ont encore des difficultés à se montrer plus efficaces que l'approche monolithique [CAC08], bien que des progrès importants aient été accomplis récemment en rendant l'apprentissage implicite [CCF+10]. Pour les systèmes BIP, l'outil D-Finder [BBSN08, BBNS09] utilise des invariants de composant ainsi que des invariants d'interaction pour prouver des propriétés de sûreté de façon compositionnelle.

Les approches compositionnelles sont performantes par rapport au problème de l'explosion de l'espace d'états. Cependant, elles n'offrent pas d'*incrémentalité* pour utiliser une propriété d'une composition de components à un niveau hiérarchique plus élevé. Certains travaux dans cette direction existent déjà : par exemple D-Finder offre désormais quelques possibilités en ce sens et [Sin07] s'intéresse aux question de substitutabilité (remplacer un composant par un autre) dans le raisonnement par *assume-guarantee*. Cependant ces résultats restent insuffisants pour un contexte industriel, où les composants sont souvent construits par des équipes différentes voire même achetés à d'autres compagnies, et où la possibilité d'intégrer des composants dans un système préexistant est essentielle. C'est la raison pour laquelle nous voulons combiner les approches compositionnelles avec une méthodologie incrémentale.

Motivation for using contracts

Les *frameworks* de contrats [BCP07, BFM+08] et d'interface [dAH01a, LNW06] émergent comme le formalisme de choix lorsque les systèmes sont *designés* par de grandes équipes réparties en sous-équipes indépendantes, or lorsque la chaîne d'approvisionnement est complexe [Dam05, SPE]. Un des clés du raisonnement par contrats est que ces derniers peuvent être utilisés aussi bien pour le *design* que pour la vérification. Les contrats sont des contraintes de *design* sur les implémentations qui sont maintenues tout au long du cycle de vie des systèmes. Comme pour le *contrat-based design* [Mey92], nous utilisons les contrats pour contraindre, réutiliser et remplacer les implémentations.

Les contrats expriment à la fois l'*assumption* (hypothèse) faite par le component sur son environnement et la *guarantee* (garantie) concernant le comportement attendu du composant. Dans le context du *design* de *programmes*, par exemple dans les méthodologies de *design* orientées objet, les contrats sont généralement de simples pré- et post-conditions, comme dans [Mey92]. Dans le context du *design* de *systèmes*, des contrats plus expressifs sont nécessaires, par exemple pour spécifier des propriétés temporelles de sûreté et de vivacité. C'est le rôle des spécifications d'interface [GSL96, dAH01a].

Le raisonnement à base de contrats utilise pleinement la notion d'incrementalité. En effet, les contrats fournissent pour les composants une abstraction qui peut être utilisée pour la composition et la réutilisation. En particulier, parce qu'un contrat exprime des contraintes sur l'environnement dans lequel un composant peut être utilisé, il est possible de raisonner sur des systèmes *fermés* (i.e., qui ne peuvent plus être composés) plutôt que sur des systèmes *ouverts* qui peuvent être utilisés dans n'importe quel environnement. D'autre part, le raisonnement à base de contrats permet l'implémentabilité indépendante : dans une approche de *design* du haut vers le bas, un contrat est écrit pour chaque sous-composant du système, et ces contrats peuvent être raffinés indépendemment les uns des autres. De plus, comme les contrats sont écrits à chaque niveau de hiérarchie du système, ils évitent le problème de l'explosion de l'espace d'états. Enfin, cette approche peut toujours être combinée avec des approches compositionnelles à chaque niveau de hiérarchie.

Le projet SPEEDS proposait d'utiliser les contrats pour prouver des propriétés du *framework* de composants HRC. À la fois pour HRC L0 et HRC L1, des théories de contrats devaient être dévelopées. À la fin, ces théories devaient être unifiées pour combiner les résultats de vérification de leur chaînes d'outils respectives, qui sont basées sur des relations de raffinement différentes. Nous avons montré que prouver des propriétés de systèmes construits à partir de composants L0 et L1 ne nécessite pas un *framework* sémantique unificateur ; l'intégration des *frameworks* de composition dans un *framework* unificateur, comme présenté ici, suffit.

Contribution

Notre but n'est pas de proposer un nouveau *framework* générique de *design* mais plutôt de définir un ensemble minimal de propriétés qui doivent être satisfaites par une théorie de contrats pour permettre certaines règles de preuves. Nous rendons explicites certaines questions auxquelles doit répondre quelqu'un qui définit un *framework* de contrats ou qui se demande quel *framework* utiliser. Par exemple, est-ce que la règle de preuve qu'on appelle raisonnement circulaire est correcte dans mon cadre ? Est-il possible de décomposer un composant hiérarchique selon n'importe quelle partition de ses

sous-composants ? Répondre à ces questions offre des réponses sur l'applicabilité de raisonnements incrémentaux et compositionnels plus sophistiqués. Ainsi, notre approche vise à séparer la question des propriétés qui dépendent du *framework* utilisé de celles qui en sont indépendantes et résultent de règles de preuve génériques.

Notre définition de *framework* de composant est très abstraite : nos opérateurs de composition (que nous appelons *glues*) permettent de représenter une grande variété de modèles d'interaction depuis la communication input/output (I/O) jusqu'aux connecteurs BIP avec politique de priorités. En particulier, cela nous permet de travailler avec des *frameworks* sémantiques de bas niveau aussi bien qu'avec des *frameworks* syntaxiques. D'autre part, l'abstraction de ports, qui un ingrédient essentiel pour prouver le raffinement entre spécifications à différents niveaux de granularité, peut être considéré comme une *glue*. Nos contrats ont une partie structurelle, qui décrit la *glue* avec laquelle le composant doit être connecté à son environnement. Ainsi, dans les *frameworks* où l'interaction est complexe, nous montrons que les proproiétés structurelles et compositionnelles d'un composant peuvent être établies en s'appuyant fortement sur la structure du système et de façon moins importante sur les propriétés comportementales de l'environnement. C'est utile en particulier parce que la structure d'un système en construction peut être fixée bien avant certaines propriétés comportementales, qui doivent être raffinées tout au long du processus de *design*. Contrairement aux automates d'interface [dAH01a], mais à l'image des automates d'interface I/O [LNW06], nos contrats distinguent explicitement entre une *assumption* A représentant une propriété de l'environnement, et une guarantie G représentant une propriété que le composant en cours de *design* doit garantir dans tout environnement se comportant en accord avec A. Nous discutons ce choix dans le chapitre 2.

Munis de cette définition générique de *framework* de composant, nous nous intéressons aux relations de raffinement. Dans le *design* de systèmes, le raffinement est défini entre une spécification et une implémentation [LW94, dAH01a, Sin07]. Dans les *frameworks* de contrat, le raffinement prend plusieurs formes : raffinement par rapport à une spécification (*conformité*), raffinement entre contrats (*dominance*) and raffinement d'une implémentation par rapport à un contrat (*satisfaction*). Dans les théories de contrats existantes, les deux dernières sont dérivées de la première. Nous proposons une relation plus faible entre conformité and *raffinement sous contexte*, à partir de laquelle les relations de dominance et satisfaction sont dérivées. Cela nous permet d'obtenir des règles de raisonnement plus efficaces. D'autre part, nous généralisons l'usage qui est parfois fait du raffinement dans tout contexte (par contraste avec les raffinement dans un contexte donné) à des relations plus faibles, ce qui permet également d'obtenir des méthodes de raisonnement plus puissantes. Notons que nous ne discutons pas du problème de trouver des relations de raffinement appropriées : cela reste une étape

de *design* dans la construction du *framework* de composant, tout comme la spécification de contrats est une étape de *design* dans la construction de systèmes. Notre but est de faciliter le processus de *design* en spécifiant clairement quelles sont les propriétés nécessaires pour permettre certaines règles de preuve.

Nous discutons certaines règles pour le raisonnement compositionnel. En particulier, nous nous intéressons au *raisonnement circulaire* qui entraîne une règle de preuve intéressante pour prouver la dominance sans composer les contrats, ce qui est impossible dans le cas général et non souhaitable dans de nombreux *frameworks* concrets. Le *raisonnement circulaire* implique une contrainte assez forte sur la relation de raffinement sous contexte, qui peut ne pas être satisfaite par la relation disponible. Nous montrons comment relâcher cette contrainte, en particulier en combinant plusieurs relations de raffinement. Notons également que nous ne proposons pas de méthode pour construire un *framework* dans lequel le raisonnement circulaire est correct. Cette question a déja été étudiée dans [McM99, Mai03b]. En particulier, [Mai03a] prouve qu'une telle règle ne peut être à la fois correcte (*sound*) et complète. Enfin, nous présentons un ensemble d'applications qui met en avant le généralité de nos définitions. La pertinence du choix de nos définitions est mise en évidence dans ces exemples.

Organisation

- Le chapitre 1 présente des notions de base sur les systèmes de transition et leur relations de raffinement. Il décrit également trois *frameworks* qui sont souvent utilisés ou mentionnés dans le reste de la partie I.
- Le chapitre 2 présente notre méthodologie à base de contrats pour vérifier des systèmes de taille arbitraire. Elle définit nos *frameworks* de contrats et présente certaines règles pour raisonner dessus.
- Le chapitre 3 met cette méthodologie en perspective en discutant comment certains concepts habituels qui n'apparaissent pas dans nos définitions peuvent être intégrés. En particulier, nous justifions le choix d'éviter la composition de contrats dans certains *frameworks*.
- Les chapitres 4 à 7 présentent plusieurs *frameworks* de contrats, chacun mettant en avant un aspect différent de notre méthodologie. Plus précisément :
  - Le chapitre 4 présente un *framework* de contrats pour le niveau sémantique de BIP. Cet exemple simple donne une première illustration des définitions présentées dans les chapitres précédents.
  - Le chapitre 5 définit deux variantes du *framework* de composants BIP qui sont utilisées dans

le chapitre suivant. Ces variantes n'ont jamais été proposées auparavant et elles mettent en évidence l'importance d'avoir une sémantique compositionnelle cohérente avec la mise à plat de systèmes.

- Le chapitre 6 montre comment les automates d'interface I/O peuvent être simplement représentés en utilisant un *framework* de contrat pour lequel les preuves de cohérence sont techniques mais simples. Des preuves plus complexes peuvent alors être grandement simplifiées. Ensuite, nous présentons les *frameworks* HRC L0 and L1 et nous décrivons comment les résultats obtenus à partir d'outils pour L0 et L1 peuvent être utilisés ensemble grâce au raisonnement circulaire combiné.
- Le chapitre 7 présente et discute plusieurs relations de raffinement pour des *frameworks* basés sur des systèmes de transition étiquetés (LTS, *Labeled Transition Systems*) et généralise l'usage du raisonnement circulaire combiné du chapitre précédent. Un *framework* basé sur les systèmes de transition modaux (MTS, *Modal Transition Systems*) est ensuite proposé, pour lequel la cohérence structurelle est une propriété utile. Enfin, deux *frameworks* utilisant des priorités statiques sont décrits, pour les LTS puis les MTS, ce qui illustre que le raffinement dans tout contexte peut dans certains cas être strictement plus fort que la relation de conformité.

Bien que cela ne soit pas présenté dans cette thèse, nous avons également appliqué notre méthodologie à un *framework* de contrats plus complexe qui est assez expressif pour décrire des échanges de données entre composants et des propriétés de progrès. Ce *framework* a été appliqué à un algorithme pour le partage de ressources dans un réseau. Une description détaillée de ce travail est disponible dans [BHQG10c, BHQG10b].

Implémentation de systèmes distribués avec interactions complexes

Le travail présenté dans la partie II est motivé par le challenge posé par l'implémentation distribuée de systèmes BIP [BBBS08]. Ce problème peut se généraliser à la question de contrôler un système existant de façon à le forcer à satisfaire une contrainte de sûreté supplémentaire [RW92] — par exemple une politique de priorités. En pratique, cela est rendu possible par l'addition d'un processus superviseur, qui s'éxécute en synchronie avec le système. Les superviseurs sont en général des automates (d'états finis) qui observent le système et progressent selon les transitions qu'ils observent, et ont de plus la capacité de bloquer certaines transitions en fonction de leur état courant.

Ce problème est lié au problème de la synthèse à partir de spécifications temporelles (LTL, *Linear*

Temporal Logic), qui est 2EXPTIME *hard* pour les systèmes réactifs séquentiels et indécidable pour les systèmes concurrents, comme démontré par Pnueli and Rosner [PR90].

Une étude exacte de l'existence d'un contrôleur global (complètement synchronisé) peut être basée sur la théorie des jeux. De fait, on peut voir le problème comme l'implémentation d'une stratégie pour le jeu à deux joueurs suivant : un joueur, l'environnement, peut toujours choisir entre les transitions incontrôlables possibles tandis que l'autre joueur peut choisir entre les transitions contrôlables. Le but du contrôleur est de s'assurer que la contrainte est satisfaite par une éxécution choisie conjointement. Ce problème peut être résolu en utilisant des algorithmes basés sur les jeux de sûreté [Tho95].

Le problème du contrôle distributé

Maintenant, intéressons-nous à la question du contrôle distribué [RW92, YL02] : des contrôleurs locaux, un par processus ou ensemble de processus, peuvent interdire l'éxécution de certaines transitions afin d'éviter la violation de la contrainte imposée. À cause de la nature distribuée du système, chaque contrôleur a une vision limitée du système global. La synthèse de contrôleur distribué pour imposer une contrainte globale est un problème indécidable dans le cas général [Tri04, Thi05]. On peut rendre le problème décidable en réduisant la concurrence dans le système. Dans le pire des cas, il ne reste plus aucun concurrence et un contrôleur totalement global est construit. Cependant, même avec cette hypothèse assez forte, le problème de synthèse de contrôleur reste difficile à gérer. Une approche pragmatique est basée sur la vérification de properties de *connaissance* grâce auxquelles les contrôleurs locaux peuvent décider ou non de bloquer une transition. Deux approches différentes existent pour synthétiser des contrôleurs distribués, qui sont alors soit *conjonctifs* soit *disjonctifs*.

Un contrôleur *conjonctif* autorise une transition seulement si *tous* les contrôleurs locaux l'autorisent. En d'autres termes, il est suffisant pour bloquer une transition qu'un contrôleur local le décide. Le choix d'un contrôleur conjonctif est motivé par le besoin d'avoir un système qui se comporte *exactement* comme spécifié par un langage régulier donné. Dans ce cas, tous les comportements du système non contrôlé qui ne violent pas la contrainte imposée doivent être possibles. Il est alors nécessaire qu'au moins un processus sache quand une transition possible dans le système viole la contrainte, afin de la bloquer. Cette approche nécessite une connaissance suffisante de chaque processus pour permettre le maximum de transitions autorisées. Elle est suivie par [RR00], qui étudie la possibilité de construire un contrôleur utilisant de la connaissance (propriété qui s'appelle *Kripke observability* dans ce travail).

Un contrôleur *disjonctif* autorise une transition si *au moins un* contrôleur local l'autorise. Selon

cette approche, suivie par [BBPS09], il est possible d'interdire une transition même si elle ne viole pas la contrainte imposée mais l'absence de blocage doit être préservée. La construction du contrôleur est différente de celle du contrôle conjonctif car elle requiert seulement qu'au moins un processus autorise une transition. Cette approche préserve la correction du contrôleur même si la connaissance ne suffit pas à autoriser toutes les transitions qui ne violent pas la contrainte.

Dans [RR00] comme dans [BBPS09], la connaissance est utilisée comme un outil pour construire des contrôleurs distribués. À partir d'une connaissance précalculée qui reflète dans chaque état local toutes les situations possibles des autres processus, le contrôleur local d'un processus décide à l'éxécution si une action de ce processus peut être éxécutée sans violer la contrainte imposée. Parfois, cependant, la connaissance des processus individuels n'est pas suffisante. Alors, la connaissance conjointe de plusieurs processus peut être monitorée à l'aide de contrôleurs associés à des groupes de processus au lieu de processus individuels. Dans le pire des cas, un contrôleur global est construit, qui peut contrôler le système selon la contrainte. Malheureusement, cela cause la perte de la concurrence entre les processus qui sont monitorés conjointement.

Contribution

L'approche proposée ici étend l'approche de [BBPS09]. Nous supposons que les systèmes permettent l'addition de synchronisations supplémentaires. Cette hypothèse est raisonnable car il existe de toute façon un mécanisme de synchronisation pour implémenter le système de façon distribuée.

Plutôt que des synchronisations permanentes entre groupes de processus, nous proposons une méthode pour construire des contrôleurs distribués qui synchronisent les processus *temporairement*. Nous utilisons des techniques de *model checking* pour précalculer un ensemble minimal de points de synchronisations auxquels la connaissance conjointe de plusieurs processus peut être calculée durant de courtes phases de coordination. Après chaque synchronisation, les processus impliqués peuvent de nouveau progresser indépendemment les uns des autres jusqu'à ce qu'une autre synchronisation soit nécessaire.

Ces synchronisations temporaires entre plusieurs processus sont rendus possibles par l'utilisation d'un algorithme de coordination basé sur des échanges de messages, tel que α-core [PCT04]. De telles synchronisations impliquent un surcoût de communication important. En conséquence, il est essentiel de minimiser le nombre de messages résultant de ces synchronisations additionnelles. Nous obtenons des résultats dans ce domaine en réduisant le nombre de coordinateurs qui gèrent ces synchronisations.

Une des limitations de cette méthode est liée au fait que nous calculons la connaissance sur le

système que nous voulons contrôler et non sur le système contrôlé. Les processus ont alors moins de connaissance et donc il est possible que nous introduisions plus de synchronisations que nécessaire. Nous proposons des techniques qui réduisent l'impact de ce problème ainsi que comment ces nouvelles techniques peuvent combinées avec celles qui existent déjà, comme par exemple la connaissance avec mémoire parfaite. Enfin, nous montrons l'intérêt des contrôleurs à base de connaissance en tant que solution pragmatique au problème de la synthèse de contrôleur.

Organisation

– Le chapitre 8 généralise the travail de [BBPS09] à des contraintes de sûreté arbitraires.
– Le chapitre 9 présente notre méthode pour construire des contrôleurs distribués en synchronisant les processus *temporairement* ainsi que pour réduire le surcoût de communication dû aux synchronisations additionnelles. Il présente également des résultats expérimentaux.
– Le chapitre 10 décrit nos techniques pour éviter des synchronisations inutiles et comment combiner plusieurs techniques. De plus, il met en perspective les contrôleurs à base de connaissance, mettant en évidence leur intérêt.

Bien que cela ne soit pas présenté dans ce document, un travail préliminaire sur l'intégration de contraintes haut-niveau avec l'algorithme de coordination a déjà été entrepris. Des résultats sont présentés dans [BHGQ10].

Context and motivation

Computer science has been part of our life for a few decades now. The changes it has introduced in our modern society are quite astonishing: computers are everywhere from cellphones to laptops to airplanes, and they do almost everything for us: they drive for us, they play music, they allow us to communicate with the rest of the world, they give us access to unlimited knowledge etc. Lately, the huge progress in the abilities of our computers has been overtaken only by the demand for more and more sophisticated systems. Developing such large and complex systems is a challenging task. To tackle this issue, system designers apply an old method that has proved its effectiveness in various contexts throughout history: *divide and conquer*. Its principle is simple and powerful: a large problem is solved more easily if it is split into smaller subproblems, which can in turn be decomposed, and so on until the size of the subproblems becomes manageable. Once these subproblems have been solved, the solution to the original problem is defined as the composition of the solutions to the subproblems. In the world of system design, this approach is called *component-based design*. The solution to the

problem is a component, which can be either small enough to be designed directly — it is then called *atomic* — or built by composing smaller components — then being called *hierarchical*. Thus, the problem of designing large and complex systems boils down to writing atomic components and assembling them into hierarchical ones.

A system written in a clear and simple way is less error-prone. Furthermore, this can significantly improve the results provided by the later verification phase. For this reason, there is a large variety of formalisms for dealing with component-based frameworks. Synchronous languages such as LUSTRE [CPHP87] or Signal [BGJ91] have been successfully transferred from the academic world to the industrial one. System modeling languages such as SystemC [Sys] and Simulink [Sim] are also widely used in the industry. Frameworks such as Ptolemy [EJL+03] and 42 [MB07] focus on handling heterogeneity by allowing combination of several models of computation and communication.

In the past decade, many languages in which interaction is complex — connectors are not only used to transfer data from one component to another but play a role in the synchronization of components — have been designed. Such formalisms include the Kell calculus [BS03] and the connector calculus Reo [Arb04]. Among them, we are particularly interested in the BIP framework [GS05, BBS06, BS08a] developed at Verimag. BIP is a language in which interaction is expressive enough to encompass many schemes, from rendezvous to broadcast. Furthermore, as connectors are first-class entities just like components, it is possible to define hierarchical connectors, which are connectors defined as a composition of other connectors. In such frameworks, reasoning about the structure of the system is essential. Our work has a practical motivation in the component frameworks HRC L0 and L1 — standing for heterogeneous rich components — defined in the SPEEDS IP project [SPE] and used in the COMBEST project [COM]. HRC L0 is inspired by the synchronous languages while HRC L1 is inspired by BIP.

In this thesis, we have studied how complex systems are designed, verified and then implemented. In particular, because of the great variety of existing formalisms, we have focused on finding definitions expressive enough to encompass BIP, but general enough to apply to other languages. Thus, this document is organized in two parts, as follows:

- Part I: *Design and verification of large systems of components using contracts.* We provide a definition of component framework that is abstract enough to encompass the BIP and HRC L0 and L1 frameworks, but that also applies in a simple way to a variety of other frameworks.
- Part II: *Implementation of systems with complex interactions in a distributed setting.* This naturally applies to BIP and its priority policy, but we have generalized the constraint to be enforced to any safety property.

Design and verification of large systems

As systems grow in size and complexity, the number of errors that they contain also grows, and furthermore they become harder to detect and to fix. The problem of verifying the correctness of a system is so difficult to solve that a computer is probably the only product that is sold without the guarantee that it has no defect.

However, for some systems, this situation is not acceptable, either because an error may cost human lives (e.g. if an airplane crashes), or because it may result in a huge financial loss (e.g. if there is a potential malfunction in a cellphone that is produced in very large quantities). Therefore, formal methods are required in order to ensure the *correctness* of some systems, where this notion has to be formally defined, for example as conformance to some set of requirements. Such requirements are of various forms depending on the application domain. Examples are pre- and postconditions for methods calls, or temporal properties which divide into safety properties ("something bad never happens") and liveness properties ("something good eventually happens"). An efficient verification method for large and complex component-based systems should have the following qualities:

– *scalability*: the approach must scale to very large systems
– *predictability*: design errors should be detected as early as possible in the design phase
– *reusability*: there must be a way to reuse parts of the verification process if a component is replaced by another one that is similar but not identical

As we focus on critical systems, testing, although it is necessary in order to detect errors in a lightweight way, is not sufficient, as it provides no guarantee of correctness. Among the methods for formal verification, model checking is a fully automated one based on exhaustive exploration of the state space of the model of the system, which is usually a finite state machine. However, it suffers from the well-known *state space explosion* problem: the method soon becomes intractable if components execute concurrently, because the number of states of the model grows exponentially in the number of components. As model checking has been widely studied, many improvements have been proposed in order to solve this problem. They fall into three categories, which can be combined: symbolic techniques [McM93], abstraction techniques [CGL94, GS97, BMMR01, CGJ+00] and compositional methods.

Compositional approaches (for a comprehensive survey, see [dRdBH+01]) are those that tackle the state space explosion problem in the most efficient way. They apply divide-and-conquer to infer, from (local) properties of the components, a (global) property of the system. These approaches include compositional model checking [CLM89, Lon93] and compositional minimization [CGL94] — which can be property-driven [CLM89, LGS+95] or context-driven [GSL96]. Assume-guarantee

is another approach [Jon83] based on decomposing a system into several parts and proving for each part that it satisfies some guarantee under the assumption that the rest of the system satisfies its assumption. Automated assumption generation using machine learning algorithms has been proposed in [CGP03]. However, these techniques still have difficulty proving more efficient than the monolithic approach [CAC08], even though much progress has been made lately by making learning implicit [CCF+10]. For BIP systems, the D-Finder tool [BBSN08, BBNS09] uses component as well as interaction invariants to prove safety properties in a compositional manner.

Compositional approaches deal with the state-space explosion problem very efficiently. However, they lack *incrementality* to use a property of a composition at a higher level of hierarchy. Some works already exist — D-Finder now provides some limited results in this direction and [Sin07] tackles substitutability (replacing one component by another one) in assume-guarantee reasoning — yet this remains insufficient in an industrial context, where components are often designed by different teams or even bought from other companies, and where the ability to incorporate components into an existing system is essential. This is the reason why we would like to combine compositional approaches with an incremental methodology.

Motivation for using contracts

Contract [BCP07, BFM+08] and interface [dAH01a, LNW06] frameworks are emerging as the formalism of choice when systems are designed by large teams consisting of independently working subteams, or when the supply chain is complex [Dam05, SPE]. The key idea is that contracts are used both for design and verification. Contracts are design constraints for implementations which are maintained throughout the development and life cycle of the system. Like in contract-based design [Mey92], we use contracts to constrain, reuse and replace implementations.

Contracts express both the *assumption* made by the component with respect to its environment and the *guarantee* about the promised behavior of this component. In the context of *program* design, for example in object-oriented design methodologies, contracts are usually simple pre- and post conditions, as in [Mey92]. In the context of *system* design, more expressive contracts are needed, for example for specifying temporal safety and progress properties. This is the role of interface specifications [GSL96, dAH01a].

Contract-based reasoning makes full use of the notion of incrementality. Indeed, contracts provide for components an abstraction that is adequate for composition and reuse. In particular, because a contract expresses constraints about the environment into which a component may be used, it is possible to reason about *closed* systems — which cannot be composed anymore — rather than *open*

systems — which may be used in an unknown environment. Besides, contract-based reasoning can provide *independent implementability*: in a top-down design approach, contracts are written for each subcomponent of the system, and these contracts can be further refined independently of the other subcomponents. Furthermore, as contracts are written at each level of hierarchy in the system, they avoid the state-space explosion problem. Besides, this approach can still be combined with compositional approaches at each level of hierarchy.

The SPEEDS IP project proposed to use contracts for proving properties within the component framework HRC. For both HRC L0 and HRC L1, contract theories had to be developed. At the end, these theories had to be unified to combine verification results from their respective tool chains, which are based on different refinement relations. We show that proving properties of systems built from L0 and L1 components does not require a unifying semantic framework; an embedding of the composition frameworks into a unifying one, as presented here, is sufficient.

Contribution

Our goal is not to propose a new generic design framework but rather to define a minimal set of properties which a given contract theory should satisfy to allow some specific proofs. We explicit some questions which must be answered by someone defining a contract framework, or wondering which existing one to use. For example, is the proof rule called circular reasoning sound in my framework? Is it possible to decompose a hierarchical component according to any partition of its subcomponents? Answering these basic questions provides clues for using more sophisticated incremental and compositional reasoning. In that sense, our approach aims at a separation of concerns between framework-dependent properties and generic proof rules.

Our definition of component framework is very abstract: our composition operators (which we call *glues*) encompass a variety of interaction models from I/O communication to BIP connectors and priority policies. In particular, this allows us to work with low-level semantic frameworks as well as syntactic ones. Besides, port hiding, which is a key ingredient for proving refinement between specifications at different levels of granularity, can be seen as a glue. Our contracts have a structural part, which describes the glue with which the component is expected to be composed with its environment. Thus, in frameworks where interaction is complex, we show that structural and behavioral properties of a component can be established by relying heavily on the structure of the system and less importantly on the behavioral properties of its environment. This is particularly useful as the architecture of a system under construction may be fixed long before its behavioral properties, which need to be refined throughout the design process. Unlike interface automata [dAH01a], but like interface

I/O automata [LNW06], our contracts distinguish explicitly between an assumption A representing a property of the environment, and a guarantee G representing a property that the component under design must guarantee in an environment behaving in accordance with A. We discuss this choice in Chapter 2.

Equipped with this generic definition of component framework, we focus on refinement relations. In system design, refinement is defined between a specification and an implementation [LW94, dAH01a, Sin07]. In contract frameworks, it takes different forms: refinement with respect to a specification (*conformance*), refinement between contracts (*dominance*) and refinement of an implementation with respect to a contract (*satisfaction*). In existing contract theories, the latter two relations are derived from the former one. We require here only a loose coupling between conformance and *refinement under context*, from which dominance and satisfaction are derived. This can be used for providing more efficient proof rules. Besides, we generalize the use that is sometimes made of refinement in any context (by contrast with refinement in a given context) to weaker relations, thus providing again more powerful reasoning schemes. Note that we do not address the problem of finding appropriate refinement under context relations: this remains a design step in the building of the contract framework, just like specification of contracts is a design step in the building of systems. What we do is helping in the design process by stating clearly the properties required from such relations for allowing particular reasoning rules.

We discuss some rules for compositional reasoning. In particular, we focus on *circular reasoning*, as it entails an interesting rule for proving dominance that does not require composing contracts — which is impossible in the general case, and undesirable in most concrete frameworks. Soundness of circular reasoning imposes a relatively strong requirement on the refinement relation. Thus, it may not hold for the refinement under context at hand. We show how to relax this constraint, in particular by combining several refinement relations. Note that we do not discuss how to build a framework for which circular reasoning is sound. This has already been studied, e.g. in [McM99, Mai03b]. In particular, [Mai03a] proves that it cannot be sound and complete.

Finally, we present a variety of meaningful instantiations, which emphasizes the generality of our definitions. The relevance of the choices made in our definitions is illustrated in these examples.

Organization

- Chapter 1 presents basic notions about transition systems and their refinement relations. It also describes three frameworks which are used or often referred to in the rest of Part I.
- Chapter 2 presents our contract-based methodology for verifying systems of arbitrary size. It

defines contract frameworks and discusses rules for reasoning within them.

- Chapter 3 puts this methodology into perspective by discussing whether and how usual concepts which are missing from our definitions can be integrated into it. In particular, we justify our choice to avoid composing contracts in some frameworks.
- Chapters 4 to 7 present several contract frameworks, each of them emphasizing a different aspect of our methodology. More specifically:
  - Chapter 4 presents a contract framework for the semantic level of BIP. This simple example gives a first illustration of the definitions presented in the previous chapters.
  - Chapter 5 defines two variants of the BIP component framework which are used in the next chapter. These variants had never been formalized before and they emphasize the importance of providing a compositional semantics consistent with flattening.
  - Chapter 6 shows how I/O interface automata can be easily represented as a contract framework in which coherence proofs are technical but straightforward. More intricate proofs can then be drastically simplified. After this, we present the HRC L0 and L1 frameworks and describe how the results obtained from tools for both can be used together using combined circular reasoning.
  - Chapter 7 discusses several refinement relations for frameworks based on LTS and generalizes the use of combined circular reasoning made in the previous chapter. It then proposes a framework based on MTS, where structural consistency is a useful property. Finally, it defines two frameworks handling static priorities, for LTS and MTS, which illustrates that refinement in any context may be much stronger than conformance.

Although this is not presented in this thesis, we have also applied our methodology to a complex contract framework that is expressive enough to handle data exchange between components and progress properties. It is applied to an algorithm for sharing resources in a network. More detail can be found in [BHQG10c, BHQG10b].

Implementation of distributed systems with complex interactions

The practical motivation for the work presented in part II is the distributed implementation of BIP systems, which is a challenging issue [BBBS08]. It can be generalized to the problem of controlling an existing system in order to force it to satisfy some additional safety constraint [RW92] — e.g. a priority policy. In practice, this is done by adding a supervisor process, which is usually an automaton running synchronously with the controlled system. Supervisors are often (finite state) automata

observing the controlled system, progressing according to the transitions they observe and having the ability to block some transitions depending on their current state.

This problem is related to the synthesis problem from LTL specifications, which was shown by Pnueli and Rosner [PR90] to be 2EXPTIME hard for sequential reactive systems and undecidable for concurrent systems.

An exact check for the existence of a global (completely synchronized) controller can be based on game theory. Accordingly, one may present the problem as implementing a strategy for the following two player game. One player, the environment, can always choose between the enabled uncontrollable transitions, while the other player can choose between the enabled controllable ones. The goal of the controller is to ensure that the constraint is satisfied by the jointly selected execution. This can be solved using algorithms based on safety games [Tho95].

The distributed control problem

Now, suppose that the situation at hand is that of a distributed control [RW92, YL02]: local controllers, one per process or set of processes, may restrict the execution of some of the transitions if their occurrence violates (or may violate) the imposed constraint. Due to the distributed nature of the system, each controller has a limited view of the entire system. The problem of synthesizing a distributed controller that imposes some global constraint on a system is, in general, undecidable [Tri04, Thi05].

One can achieve decidability at the expense of reducing concurrency. In the worst case, no concurrency remains and a completely global controller is built. Even under this flexible design assumption, the synthesis problem remains highly intractable. One practical method for designing controllers is based on checking knowledge properties upon which the processes can make their decisions whether to allow or block transitions.

Two different approaches have been proposed in order to tackle the distributed control problem. The first one is based on *conjunctive* (distributed) controllers while the second one uses *disjunctive* (distributed) controllers.

A *conjunctive* controller allows a transition to be fired only if *all* local controllers allow it. In other words, it is sufficient for a transition to be blocked that one local controller decides it. The choice of a conjunctive controller is motivated by the desire to make the system behave *exactly* according to a given regular language. In this case, all behaviors of the uncontrolled system which do not violate the additional constraint imposed by the language should be allowed. It is then necessary that if a transition is enabled by the system under control but must be blocked according to the additional constraint, at least one process knows that fact and is thus able to prevent the execution of the transition.

This approach requires sufficient knowledge to allow any transition enabled according to a given regular specification. This is the approach followed by [RR00], where knowledge-based controllability (termed *Kripke observability*) is studied as a basis for constructing a distributed controller.

A *disjunctive* controller allows a transition to be fired if *at least one* of the local controllers supports it. In this approach, which is the one chosen in [BBPS09], we are allowed to limit the possible choices in order to impose the given global constraint. The construction is different from conjunctive control, as it requires that *at least one* process knows that the occurrence of *some* enabled transition preserves the correctness of the imposed constraint, hence supports its execution. This approach preserves the correctness of the controller even if there is not sufficient knowledge (in individual processes) to allow every globally feasible transition.

In [RR00] as well as in [BBPS09], knowledge is used as a tool for constructing distributed controllers. The *knowledge* of a process in any particular local state includes the properties that are common to all reachable (global) states containing it. Note that there are several definitions for knowledge, depending on how much of the local history may be encoded in the local state.

In [BBPS09], distributed control is achieved by first precalculating the knowledge of individual processes in each of their local state using model checking. Based on that precalculated knowledge, reflecting in a given local state all the possible situations of the other processes, the local controller of a process decides at runtime whether an action of that process can be executed without violating the imposed constraint.

Sometimes, however, the knowledge of individual processes is not sufficient. Then, the joint knowledge of several processes may be monitored using fixed controllers associated with groups of processes instead of individual processes. In the worst case, a completely global controller is built, which can control the system according to the property. Unfortunately, this causes the loss of concurrency among the processes that are jointly monitored.

Contribution

The approach suggested here extends the knowledge-based approach of [BBPS09]. We assume that systems are flexible to the addition of further synchronizations — which seems reasonable as there exists a synchronization mechanism for implementing the system in a distributed manner.

Instead of permanent synchronizations via fixed groups of processes, we propose a method for constructing distributed controllers that synchronize processes *temporarily*. We use model-checking techniques to precalculate a minimal set of synchronization points, where joint knowledge, i.e., knowledge common to several processes, can be achieved during short coordination phases. After

each synchronization, the participating processes can again progress independently until a further synchronization is called for.

Temporary multiprocess synchronizations are achieved by a coordination algorithm based on asynchronous message passing, such as the α-core algorithm [PCT04]. Such synchronizations are expensive as they incur communication overhead. Therefore, an important part of our task is to minimize the number of messages resulting from additional synchronizations. This is done by reducing the number of coordinators involved in those synchronizations.

One major deficiency of this method is due to the fact that we calculate the knowledge on the system that we want to control, and not on the controlled system. Processes have less knowledge, and as a result, we may introduce far more synchronizations than needed. We show techniques that reduce the impact of this problem, and how these new techniques can be profitably coupled with existing ones, such as knowledge with perfect recall, and then we advocate knowledge-based controllers with additional synchronizations as a practical solution to the distributed control problem.

Organization

- Chapter 8 generalizes the work of [BBPS09] to arbitrary safety constraints.
- Chapter 9 presents our method for constructing distributed controllers by synchronizing processes *temporarily*, and how the communication overhead induced by these additional synchronizations can be reduced. It also presents some experimental results that we have obtained.
- Chapter 10 describes our techniques for avoiding unnecessary synchronizations, and how several techniques can be combined. Furthermore, it puts knowledge-based controllers into perspective, showing their interest in practice.

Although this is not presented in this document, a preliminary work on how to integrate high-level constraints into the underlying coordination algorithm has already been undertaken. Some results are presented in [BHGQ10].

Instructions for the reader

- The preliminary part about Labeled Transition Systems (Section 1.1.1) may be skipped by a reader familiar with inclusion of traces, simulation and ready-simulation.
- Similarly, the reader already acquainted with Modal Transition Systems and their classical refinement relation may skip Section 1.1.2.
- The presentation of the BIP framework (Section 1.2.1) should be read before Chapter 4, as there are several ways of defining BIP and it specifies the definitions which are used in this thesis.
- Although related, Part I and Part II can be read in any order.
- Proofs are either just after theorems, or at the end of the chapter in which theorems appear.

Conventions and notations

- Unless otherwise stated, all sets are supposed to be finite in this thesis.
- As we are only dealing with integer numbers here, $[i, j]$ denotes, for i and j integers such that $i \leq j$, the set of integers that are greater than or equal to i and less than or equal to j.
- The set of powersets of a set X is denoted 2^X rather than $\mathcal{P}(X)$.
- When representing graphically transition systems, we show only states which are reachable from the initial state. We use two slightly graphical representations for transition systems:
  - One that represents states as circles. The initial state is pointed at by an arrow with no origin state.
  - Another where states are implicit and only transitions are represented. In this case, the convention is that the initial state is at the top.
- For labeled transition systems, we sometimes adopt the convention that in any state q, there exists a transition $q \xrightarrow{\emptyset} q$. As we do not deal with data transformation, such a transition is equivalent to an absence of transition. In this case, $\emptyset$ is implicitly defined as a label of any transition system.

Part I

Contract-Based Design and Verification of Component-Based Systems

Chapter 1

Preliminaries and related work

In this chapter, we first recall some basic definitions about labeled transition systems and modal transition systems and their usual refinement relations, which are used throughout this document. Then, we present the BIP framework, which is one the main motivations for our work. We briefly discuss BIP in its generality, then we present two variants which had never been formalized before and that we use in this thesis. Finally, we review the state of the art in the domain of interface theories, from which our work on contract frameworks is inspired. In particular, we emphasize the properties of these theories that do not extend to frameworks with complex interaction such as BIP.

1.1 Preliminaries

1.1.1 Labeled transition systems

Labeled transition systems are used to describe abstractly the behavior of systems. They define how these systems can evolve from one state to another by firing a transition associated with a label that names the operation performed during the transition.

Definition 1.1.1 (Labeled transition system) *A* labeled transition system *(LTS) is defined as a tuple* $(Q, q^0, \Sigma, \longrightarrow)$ *where* Q *is a set of states,* $q^0 \in Q$ *is the initial state,* Σ *is a set of labels and* $\longrightarrow \; \subseteq Q \times \Sigma \times Q$ *is a transition relation.*

For $a \in \Sigma$, we call q the *origin* and q' the *destination* of a transition (q, a, q'). We use the following notations: $(q, a, q') \in\longrightarrow$ is denoted $q \xrightarrow{a} q'$. Given $q \in Q$ and $a \in \Sigma$, if there exists a state $q' \in Q$ such that $q \xrightarrow{a} q'$, then a is said to be *enabled* in q, which is denoted $q \xrightarrow{a}$. In the sequel, whenever an LTS S_i is introduced, it is understood that it is of the form $S_i = (Q_i, q_i^0, \Sigma_i, \longrightarrow_i)$.

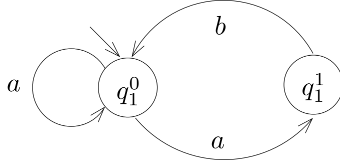


Figure 1.1 – A first LTS S_1

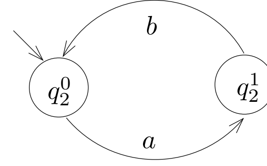


Figure 1.2 – A deterministic LTS S_2

Let us consider a very simple LTS which we will use as a running example.

Example 1.1.2 *We define* $S_1 = (Q_1, q_1^0, \Sigma_1, \longrightarrow_1)$ *by:* $Q_1 = \{q_1^0, q_1^1\}$, $\Sigma = \{a, b\}$ *and* $\longrightarrow_1 = \{(q_1^0, a, q_1^0), (q_1^0, a, q_1^1), (q_1^1, b, q_1^0)\}$.

Graphically, S_1 is represented in Figure 1.1, where circles denote states and arrows are transitions whose labels are written on the arrows. We show only states which are reachable from the initial state. The initial state is represented using an arrow with no origin state.

Definition 1.1.3 (Determinism) *An LTS S is* deterministic *if for every state $q \in Q$ and every label a, there is a most one state q' such that $q \xrightarrow{a} q'$.*

An LTS which is not deterministic is called *non-deterministic*. Determinism expresses that in every state, once a label has been chosen, there is at most one possible destination. For example, S_1 is non-deterministic because there are two transitions labeled by a with q_1^0 as origin. On the contrary, S_2 as shown in Figure 1.2 is deterministic. Note that there exist other (stricter) notions of non-determinism.

Now, given a non-deterministic LTS S, it is always possible to build a deterministic LTS S^{det} which is related to S by some equivalence relation, namely equality of the set of traces. Before presenting determinization, let us introduce traces [HU79]. We do not consider infinite traces.

Definition 1.1.4 (Set of traces) *A (finite)* trace *of an LTS S is a sequence of labels $a_1.a_2 \ldots a_n$ for which there exists a sequence of states $q_0.q_1 \ldots q_n$ such that $q_0 = q^0$ and for every $0 \leq i < n$ there is a transition $q_i \xrightarrow{a_{i+1}} q_{i+1}$. The* length *of the trace is n. The set of traces of S is denoted $Tr(S)$.*

Traces are in general denoted σ. The sequence $q_0.q_1 \ldots q_n$ is called the sequence of states *corresponding to $a_1.a_2 \ldots a_n$*. Traces express the sequence of labels that can be observed when executing an LTS. Note that any prefix of a trace of an LTS is also a trace of this LTS. As an example, the traces of S_1 are all the sequences of a and b such that every b is preceded by at least one a, while traces of S_2 are such that every b is preceded by exactly a. Hence, $Tr(S_2) \subseteq Tr(S_1)$ but $Tr(S_1) \not\subseteq Tr(S_2)$.

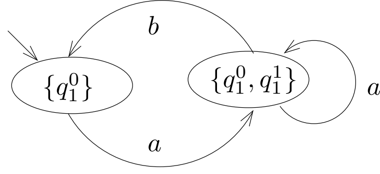


Figure 1.3 – The determinization S_1^{det} of S_1

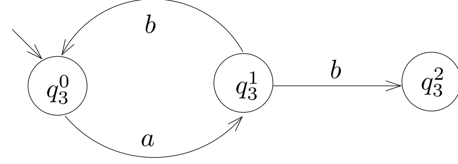


Figure 1.4 – An LTS S_3 with a deadlock

Property 1.1.5 *In a deterministic LTS, every trace has only one corresponding sequence of states.*

Proof. As the initial state is unique, the proof is a simple induction on the length of the traces. □

Definition 1.1.6 (Determinization) *The* determinization *of an LTS S, denoted S^{det}, is the LTS $(2^Q, \{q^0\}, \Sigma, \longrightarrow_{det})$ where $\longrightarrow_{det}$ consists of triples in $2^Q \times \Sigma \times 2^Q$ and is the smallest relation such that for $\mathcal{Q}, \mathcal{Q}' \subseteq Q$, $\mathcal{Q} \xrightarrow{a}_{det} \mathcal{Q}'$ if and only if $\mathcal{Q}' \neq \emptyset$ and $\mathcal{Q}' = \{q' \mid \exists q \in \mathcal{Q}\ s.t.\ q \xrightarrow{a} q'\}$.*

The condition $\mathcal{Q}' \neq \emptyset$ ensures that there exists a transition in the determinized LTS only if there exists at least one corresponding transition in the original LTS. Note that for a given S, S^{det} is unique and deterministic. Uniqueness and determinism come from the fact that in every state $\mathcal{Q}'$ is uniquely defined. The determinization of S_1 is shown on Figure 1.3. Only states which are reachable are represented, where reachability is defined as follows.

Definition 1.1.7 (Reachable states) *In an LTS S, a state $q \in Q$ is* reachable *if it appears in at least one trace of S. The set of reachable states of S is denoted $reach(S)$.*

One should note that the number of states of a determinization is exponentially larger than the one of the original system. However, some (or in some cases many) of these states may not be reachable.

Property 1.1.8 *For any LTS S, it holds that $Tr(S) = Tr(S^{det})$.*

Proof. We prove by induction the following property: for any $l \geq 0$, the set of traces of length l of S is equal to the set of traces of length l of S^{det}, and for any σ of length l in these sets, if $\mathcal{Q}_0.\mathcal{Q}_1 \ldots \mathcal{Q}_{l-1}$ is the (unique) sequence of states corresponding to σ in S^{det}, then we have $\mathcal{Q}_{l-1} = \{q_{l-1} \in Q \mid q_0.q_1 \ldots q_{l-1}$ is a sequence of states in S corresponding to $\sigma\}$. □

Comparing sets of traces is one possibility for comparing LTS, it is not the only one. In particular, as illustrated by Property 1.1.8, equality of trace sets does not take non-determinism into account. In particular, it does not express properties related to deadlocks, i.e. states in which the system cannot progress anymore.

	S_1	S_2	S_1^{det}	S_3
S_1	✓		✓	
S_2	✓	✓	✓	✓
S_1^{det}			✓	
S_3	✓	✓	✓	✓

Table 1.1 – Simulation relations between S_1, S_2, S_1^{det} and S_3

Definition 1.1.9 (Deadlock) *A state $q \in Q$ in which no $a \in \Sigma$ is enabled is a* deadlock.

Definition 1.1.10 (Deadlock-freedom) *An LTS without any deadlock state is called* deadlock-free.

As an example, S_2 and S_3 represented in Figure 1.4 have the same set of traces, yet S_2 is deadlock-free while S_3 is not.

We now introduce *simulation* [Mil89], which is a relation taking into account non-determinism.

Definition 1.1.11 (Simulation) *Let S_1 and S_2 be two LTS. A relation $\mathcal{R} \subseteq Q_1 \times Q_2$ is a* simulation *relation of S_2 by S_1 iff $q_1^0 \, \mathcal{R} \, q_2^0$ and for any pair $(q_1, q_2) \in Q_1 \times Q_2$ and any $q_1' \in Q_1$:*

$$q_1 \, \mathcal{R} \, q_2 \text{ and } q_1 \xrightarrow{a}_1 q_1' \text{ implies that there exists } q_2' \in Q_2 \text{ such that } q_2 \xrightarrow{a}_2 q_2' \text{ and } q_1' \, \mathcal{R} \, q_2'$$

S_1 simulates S_2 *if and only if there exists such a relation.*

Intuitively, an LTS S_1 simulates S_2 if any reachable state q_1 of S_1 can be mapped to a state q_2 in S_2 such that all labels enabled in q_1 (w.r.t S_1) are also enabled in q_2 (w.r.t. S_2).

If we look at the four examples that we have presented so far, we have the relations represented in Table 1.1: ✓denotes that there exists a simulation of the LTS associated with the column by the LTS corresponding to the line. An empty cell denotes that there exists no such simulation. In particular, this table illustrates the fact that simulation is reflexive (for any S, identity is a simulation of S by S) and that for any S, S simulates S^{det}.

The unique simulation relation between S_1 and S_1^{det} is shown in Figure 1.5, left: a green dashed arrow from q to q' denotes that $q \, \mathcal{R} \, q'$. Similarly, the unique simulation between S_3 and S_2 is shown on the right of Figure 1.5. Note that S_2 simulates S_3 and vice versa, but not with the same relation.

Property 1.1.12 *For any LTS S, S simulates S^{det}.*

Proof. Let $\mathcal{R} \subseteq Q \times 2^Q$ be a relation defined as follows: $q \, \mathcal{R} \, \mathcal{Q}$ if and only if $q \in \mathcal{Q}$, for any $q \in Q$ and $\mathcal{Q} \subseteq Q$. Relation $\mathcal{R}$ is a simulation of S^{det} by S. □

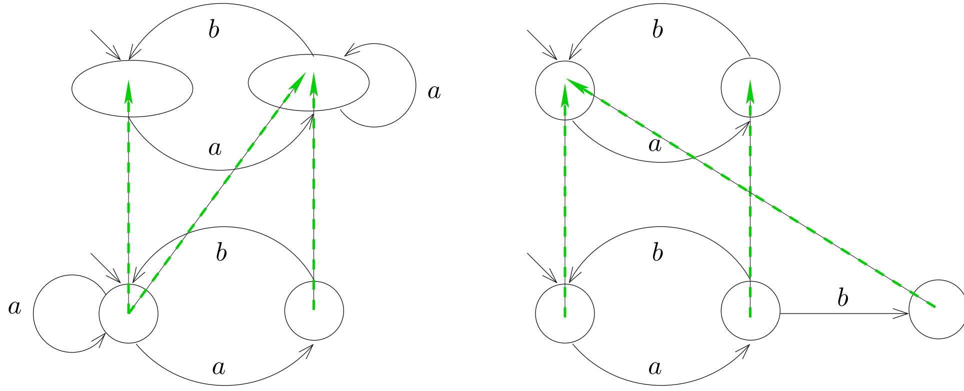


Figure 1.5 – Simulation relation between S_1 and S_1^{det} (left) and S_3 and S_2 (right)

Property 1.1.13 *If S_1 simulates S_2 then $Tr(S_1) \subseteq Tr(S_2)$.*
Proof. Again, the proof is a simple induction on the length of the traces. □

Simulation is strictly stronger than inclusion of traces. Furthermore, inclusion of traces and simulation can be related by the following equivalence.

Property 1.1.14 *$Tr(S_1) \subseteq Tr(S_2)$ if and only if S_1^{det} simulates S_2^{det}.*
Proof. We decompose this proof into two implications.
$\Longrightarrow$: Suppose every trace of S_1 is a trace of S_2. We define $\mathcal{R} \subseteq 2^{Q_1} \times 2^{Q_2}$ as follows: $\{q_1^0\} \, \mathcal{R} \, \{q_2^0\}$, and if $\mathcal{Q}_1 \, \mathcal{R} \, \mathcal{Q}_2$ and $\mathcal{Q}_1 \xrightarrow{a} \mathcal{Q}_1'$, then $\mathcal{Q}_1' \, \mathcal{R} \, \mathcal{Q}_2'$, where $\mathcal{Q}_2' = \{q_2' \mid \exists q_2 \in \mathcal{Q}_2 \; s.t. \; q_2 \xrightarrow{a} q_2'\}$. Relation $\mathcal{R}$ is a simulation of S_2^{det} by S_1^{det}.
$\Longleftarrow$: is a direct consequence of Property 1.1.13 and Property 1.1.8. □

In some cases, we will need a relation more discriminating than simulation: we will use *ready-simulation*, which imposes that a transition in the simulated LTS should have a counterpart in the simulating LTS.

Definition 1.1.15 (Ready-simulation) *Let K_1 and K_2 be two LTS. A relation $\mathcal{R} \subseteq Q_1 \times Q_2$ is a* ready-simulation *if it is a simulation such that for any pair $(q_1, q_2) \in Q_1 \times Q_2$, if $q_1 \, \mathcal{R} \, q_2$ then:*

$$q_2 \xrightarrow{a} \textit{ implies } q_1 \xrightarrow{a}$$

Table 1.2 shows which of the simulations presented in Table 1.1 are also ready-simulations: a ready-simulation relation is denoted by ✓and a simulation which is not a ready-simulation by ✗. In

	S_1	S_2	S_1^{det}	S_3
S_1	✓		✗	
S_2	✓	✓	✗	✓
S_1^{det}			✓	
S_3	✗	✗	✗	✓

Table 1.2 – Ready-simulation relations between S_1, S_2, S_1^{det} and S_3

particular, notice that the simulations represented in Figure 1.5 are not ready-simulations. This shows that ready-simulation is sufficient to handle properties related to deadlock-freedom.

Finally, we define an even stronger relation called bisimulation. It is mentioned only is the next section, so we not discuss it in detail.

Definition 1.1.16 (Bisimulation) *Let K_1 and K_2 be two LTS. A relation $\mathcal{R} \subseteq Q_1 \times Q_2$ is a* bisimulation *if it is a simulation and furthermore $\mathcal{R}^{-1}$ is a simulation of K_1 by K_2.*

1.1.2 Modal transition systems

Modal transition systems (MTS, [LX90]) are labeled transition systems where transitions have in addition a *modality*, which is either *may* or *must* and allows distinguishing between impossible, possible and required behaviors of a component. MTS express loose specifications in so far as they encode into a single automaton an over- and an under-approximation of the expected behavior of a system under design without implicitly relying on state information. They are therefore suitable to reason about both safety and progress properties in the context of the composition that we have chosen.

Definition 1.1.17 (Modal transition system) *A* modal transition system *(MTS) is a tuple $M = (Q, q^0, \Sigma, \dashrightarrow, \longrightarrow)$, where Q is a set of states, $q^0 \in Q$ is an initial state, Σ is a set of labels, while $\dashrightarrow$ and $\longrightarrow$ in $Q \times \Sigma \times Q$ are transition relations such that:*

$$\forall q, q' \in Q, \forall a \in \Sigma,\ q \xrightarrow{a} q' \implies q \overset{a}{\dashrightarrow} q'$$

A transition in $\dashrightarrow$, called a *may*-transition, *may* be present as well as not in an implementation of M while a transition in $\longrightarrow$, called *must*-transition, *must* be present. Thus $\dashrightarrow$ represents the over-approximation and $\longrightarrow$ the under-approximation specified by M.

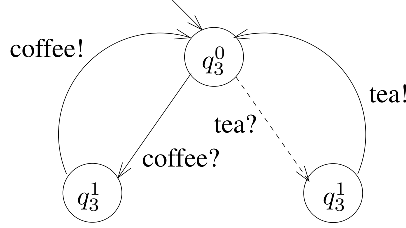


Figure 1.6 – An MTS describing a coffee machine

This condition that *must*-transitions must also be *may*-transitions is called *modal consistency*. As for LTS, an MTS M_i is implicitly associated with a tuple $(Q_i, q_i^0, \Sigma_i, \dashrightarrow_i, \longrightarrow_i)$.

Figure 1.6 shows an example of MTS: full lines represent *must*-transitions and dashed lines *may*-transitions. *May*-transitions which are also *must*-transitions are not represented. Notice that an MTS cannot express that the transitions related to tea should be either both present or both absent.

MTS are typically used for writing abstract specifications of systems. In this context, refining an MTS means choosing, for every *may*-transition, whether it should be removed, transformed into a *must*-transition or kept like this. *Must*-transitions must be preserved and no transition may be added.

Definition 1.1.18 (Modal refinement) *Consider two MTS called M_1 and M_2. M_1* refines *M_2 if and only if there exists a relation $\mathcal{R} \subseteq Q_1 \times Q_2$ such that $q_1^0 \,\mathcal{R}\, q_2^0$ and for any pair $(q_1, q_2) \in Q_1 \times Q_2$ and any $q_1' \in Q_1$, $q_2' \in Q_2$, whenever $q_1 \,\mathcal{R}\, q_2$ the following holds:*
- *$q_1 \overset{a}{\dashrightarrow} q_1'$ implies that there exists $q_2' \in Q_2$ such that $q_2 \overset{a}{\dashrightarrow} q_2'$ and $q_1' \,\mathcal{R}\, q_2'$*
- *$q_2 \overset{a}{\longrightarrow} q_2'$ implies that there exists $q_1' \in Q_1$ such that $q_1 \overset{a}{\longrightarrow} q_1'$ and $q_1' \,\mathcal{R}\, q_2'$*

For example, the MTS of Figure 1.6 may be refined in 3 different ways, as the *may*-transition can be preserved, removed or transformed into a *must*-transition. Note that an MTS with only *may*-transitions can be seen as an LTS, and in this case modal refinement corresponds to simulation.

1.2 Related work

1.2.1 The BIP framework

BIP [GS05, BBS06, BS08a] is a framework for designing component-based systems with complex interactions. Its main principle is that there should be a clear separation between the behavioral and the architectural parts of systems. Indeed, such a separation allows efficient structural verification

techniques [BBSN08]. The BIP framework has been fully implemented in a toolset including a frontend for generation of C++ code from BIP specifications, an execution engine, and analysis tools for state-space exploration and deadlock detection [BBNS09].

In BIP, the classic notion of input/output is replaced by the more expressive notion of multi-party interaction, where each partner imposes constraints on when the interaction may take place. Thus, no notion of input completeness is required. This rendezvous-like interaction mechanism is related to process algebras such as CCS [Mil80] or CSP [Hoa85], and so is the restriction to a strictly local notion of state. It has been shown in [BS08b] that the set of glues that can be defined within the BIP framework is, according to a definition taking into account the ability to coordinate components, more expressive than composition operators of CCS, CSP and SCCS [Mil83]. In fact, any glue definable by a set of SOS-rules — in a simple restriction of the GSOS format — is definable in BIP.

In BIP, systems and components are built by superposing three layers of modeling: Behavior, Interaction, and Priority — hence the name. Interaction and priority form the architecture of the system. The behavior of atomic components is typically represented as an LTS; interaction describes how components communicate and synchronize, with priority filtering among the interactions enabled in a given state, e.g., a backup interaction should not happen if a normal interaction is also possible. There exist various flavors of BIP. Here is an overview of the variants that have been studied so far:

- B Behaviors may be LTS or Petri nets. In this thesis, we also use MTS whenever some notion of progress is required. In general, behaviors are enriched with variables and guards.
- I Interactions may be structured into (potentially hierarchical) connectors. They may also involve data transfer. Some variants (in particular those with data transfer) offer the ability to internalize some ports (that is, internal ports do not appear at the interface of the component) and to associate new ports with connectors, which we call *encapsulation*.
- P Priorities may be static or dynamic.

Two different semantics have also been provided: a one-shot semantics allows firing a single connector in a given state. On the contrary, a multi-shot semantics allows firing several connectors at the same time, provided that they do not involve the same ports.

In this section, we introduce the BIP framework based on the way it was presented in [GS05]. We discuss the issue of finding a semantics that is *compositional*, i.e., a semantics such that the semantics of a composite component can be deduced from the semantics of its constituting components. This problem has not been explicitly tackled so far, and we will examine it further in Chapter 5, where we formalize two variants of the BIP framework for which we provide compositional semantics. We also explain why it is not possible in the general case to provide such compositional semantics in the form

of a behavior.

In contrast with [GS05] but without loss of generality, the BIP framework as we present it here does not structure interactions into connectors. We have chosen this presentation because it makes the presentation simpler and furthermore connectors are particularly useful in presence of encapsulation, which is not the case here, as the interface of a composite component is defined by the interface of its constituting components: it is not possible to hide, add or rename ports.

Let us introduce first some notations. In this section as well as in the rest of this thesis, components are denoted K_1, K_2 etc. Sets of components are sometimes denoted $\mathcal{K}^1, \mathcal{K}^2$ etc. A component K interacts with its environment through its *ports*, which form the *interface* of K, thus defining what can be observed from the behavior of K by its environment. We suppose given a set of ports $Ports$ containing all ports that can possibly be defined. Sets of ports are denoted P, or $\mathcal{P}$ when they correspond to the interface of a component. It is understood that they are all subsets of $Ports$.

To define our component framework, we proceed as follows: we first define atomic components which are identified with their behavior, then interactions and finally priorities. Based on these definitions, we introduce composite components and their semantics. Moreover, we define an operation called *flattening* which allows representing any composite component as a composition of atomic components using a single interaction model and priority order, that is in the layered BIP form. We show that this operation is consistent with the semantics previously introduced in the sense that the semantics of a component is equivalent (defined as bisimilar) to the semantics of its flattened form.

Let us start by defining the behavioral layer of BIP. We use the term *atomic component* rather than behavior to emphasize their role in component-based design.

Definition 1.2.1 (Atomic component) *An* atomic component *on a interface $\mathcal{P}$ is represented by an LTS $K = (Q, q^0, 2^{\mathcal{P}}, \longrightarrow)$.*

We choose here to label transitions of atomic components by sets of ports rather than ports in order to encompass the case where such components interact with their environment through several ports at the same time — e.g. in a synchronous system.

An *interaction* is characterized by a non-empty set of ports which synchronize, generally involving more than one component and possibly involving several ports of the same component. This loose definition allows us to define an interaction model on a set of ports without considering to which component these ports belong to. Hence the following definition.

Definition 1.2.2 (Interaction model) *An* interaction α *in a set of ports P is a non-empty set of ports such that $\alpha \subseteq P$. An* interaction model $\mathcal{I}$ *on a set of ports P is a set of interactions α in P.*

Note that by defining interactions as sets of ports through which these components synchronize, we restricted their use in this thesis to synchronization purposes. In particular, no data exchange or transformation can be specified.

Now that we have formalized the first two layers of the BIP framework, namely atomic components for behavior and interaction models for interaction, let us define priority. Priorities are used to arbitrate between simultaneously enabled interactions, for example to enforce scheduling policies.

Definition 1.2.3 (Priority order) *A* priority order $\prec$ *on an interaction model* $\mathcal{I}$ *is a strict partial order on* $\mathcal{I}$.

Atomic components, interaction models and priority orders form the basis of our framework. What we now focus on is how these concepts can be used in component-based design. In particular, we define composite (non-atomic) components and provide a semantics for them. In fact, we provide two such semantics, one that is compositional and another one for closed systems. Having a compositional semantics is useful especially if it is in the form of an atomic component, as in this case one can reason only about atomic components. The reason why we need two different semantics is the following: the intended goal of priorities is to filter away some transitions in states where another transition with a preferred interaction is possible. However, this cannot be done in a compositional manner. Intuitively the reason is that an interaction that has lower priority than another one in a local state of a component K may be part of an interaction with maximal priority in the corresponding global state of a composite component containing K. Thus, all enabled transitions must be kept until the system is closed, that is, cannot be composed anymore. Independently of this, priorities also must be preserved. This means that the compositional semantics of a component on an interface $\mathcal{P}$ has to be defined as a pair $(B, \prec)$ where B is an LTS labeled by $2^{\mathcal{P}}$ and $\prec$ is a priority order on $2^{\mathcal{P}}$.

Definition 1.2.4 (Component, Composite component) *A* component *is either an atomic component or it is inductively defined as the composition of a set of components* $\{K_i\}_{i=1}^n$ *with disjoint interfaces* $\{\mathcal{P}_i\}_{i=1}^n$ *using an interaction model* $\mathcal{I}$ *on* $\mathcal{P} = \bigcup_{i=1}^n \mathcal{P}_i$ *and a priority order* $\prec$ *on* $\mathcal{I}$. *Such a composition is called a* composite component *on* $\mathcal{P}$ *and it is denoted* $\mathcal{I}_\prec\{K_i\}_{i=1}^n$.

Note that the interface of a composite component is the union of the interfaces of its constituting components. From now on, when we say that a component is of the form $\mathcal{I}_\prec\{K_1, \dots, K_n\}$, it is understood that all K_i are components with disjoint interfaces, $\mathcal{I}$ is an interaction model on $\bigcup_{i=1}^n \mathcal{P}_i$ and $\prec$ is a priority order on $\mathcal{I}$. The pair $(\mathcal{I}, \prec)$, also denoted $\mathcal{I}_\prec$, is a *glue* on P in the sense of [Sif05], because it expresses how to compose a set of components so as to make them interact.

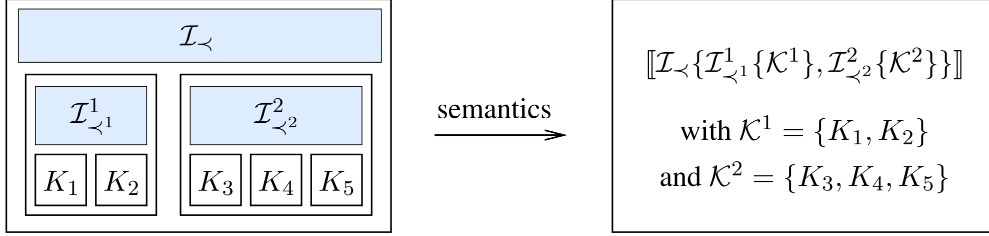


Figure 1.7 – Semantics of a composite component

Definition 1.2.5 (Glue) *Given a set of ports $P \subseteq Ports$, an interaction model $\mathcal{I}$ on P and a priority order $\prec$ on $\mathcal{I}$ form together a partial function which associates with every set of components $\{K_i\}_{i=1}^n$ with disjoint interfaces $\{\mathcal{P}_i\}_{i=1}^n$ such that $P = \bigcup_{i=1}^n \mathcal{P}_i$ a component on P. We call such a function a* glue *on P.*

In order to define semantics of components, we first define how we compose priority orders.

Definition 1.2.6 (Composition of priorities) *Given two priority orders $\prec_1$ and $\prec_2$ on two interaction sets denoted respectively $\mathcal{I}_1$ and $\mathcal{I}_2$, their composition denoted $\prec_1 \times \prec_2$ is the least priority order (if it exists) on $\mathcal{I}_1 \cup \mathcal{I}_2$ containing $\prec_1$ and $\prec_2$.*

We can now define our two semantics of components. The compositional semantics of a component consists of an LTS and a priority order while the closed one is simply an LTS. In both cases, the semantics of a composite component is inductively defined based on the *compositional* semantics of its constituting components.

Definition 1.2.7 (Compositional semantics) *The compositional semantics of a component K is a pair $(B^K, \prec^K)$ defined as follows:*

- *if K is an atomic component, then $B^K = K$ and $\prec^K = \emptyset$*
- *if K is of the form $\mathcal{I}_{\prec}\{K_1, \ldots, K_n\}$: denote $(B_i, \prec_i)$ the compositional semantics of K_i for $i \in [1, n]$, with $B_i = (Q_i, q_i^0, 2^{\mathcal{P}_i}, \rightsquigarrow_i)$. Then $\prec^K = \prec \times \prec_1 \times \ldots \times \prec_n$ if it exists, and $B^K = (Q, q^0, 2^{\mathcal{P}}, \rightsquigarrow)$ is defined by: $Q = Q_1 \times \ldots \times Q_n$, $q^0 = (q_1^0, \ldots, q_n^0)$, $\mathcal{P} = \bigcup_{i=1}^n \mathcal{P}_i$ and given two states $q = (q_1, \ldots, q_n)$ and $q' = (q'_1, \ldots, q'_n)$ in Q and an interaction $\alpha \in \mathcal{I}$, $q \overset{\alpha}{\rightsquigarrow} q'$ if and only if:*
  - *$\forall i$ such that $\alpha \cap \mathcal{P}_i = \emptyset$, $q_i^1 = q_i^2$*
  - *$\forall i$ such that $\alpha \cap \mathcal{P}_i = \alpha_i$ for some $\alpha_i \neq \emptyset$, $q_i \overset{\alpha_i}{\rightsquigarrow}_i q'_i$*

Note that the compositional semantics of a component may not be defined, as its constituting priority orders may be contradictory. According to this definition, components not involved in an interaction do not move when this interaction takes place. Also, as already explained, note that priorities do not play any role in the definition of B^K. Compared to the compositional semantics $(B^K, \prec^K)$ of a component K, the closed semantics is the LTS obtained by filtering away in B^K transitions which do not have maximal priority according to $\prec^K$. The closed semantics of component K, which we denote $[\![K]\!]$, is defined as follows.

Definition 1.2.8 (Closed semantics) *Let K be a component. Denote $(B^K, \prec^K)$ the compositional semantics of K with $B^K = (Q, q^0, 2^{\mathcal{P}}, \rightsquigarrow)$. The* closed semantics *of K, denoted $[\![K]\!]$, is defined as $(Q, q^0, 2^{\mathcal{P}}, \longrightarrow)$, where given two states $q = (q_1, \ldots, q_n)$ and $q' = (q'_1, \ldots, q'_n)$ in Q and an interaction $\alpha \in 2^{\mathcal{P}}$, $q \xrightarrow{\alpha} q'$ if and only if $q \overset{\alpha}{\rightsquigarrow} q'$ and $\nexists \alpha' \in \mathcal{I}$ such that $\alpha \prec \alpha'$ and $q \overset{\alpha'}{\rightsquigarrow}$.*

Figure 1.8 illustrates this definition on an example. For clarity, a singleton interaction $\{p\}$ is denoted p, and union of interactions is represented using a dot, thus $\{p_1, p_2\}$ is denoted $p_1 . p_2$. Note that ports a and f are exported by defining singleton interactions.

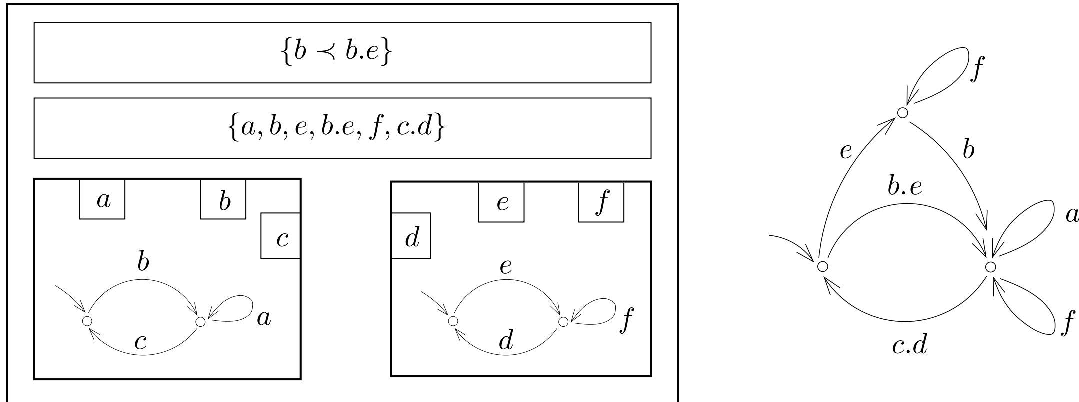


Figure 1.8 – Closed semantics of a composite component

A key feature of BIP is that glues themselves can be composed, which is done layerwise. This allows “flattening” composite components, that is, representing them using only atomic components and a single interaction model and priority layer (see Figure 1.9) — which is the representation in three layers (Behavior, Interaction, Priority) advocated in the introduction. We have already defined composition of priorities, let us define now composition of interaction models.

Definition 1.2.9 (Composition of interaction models) *Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be two interaction sets on respectively P_1 and P_2. Their composition, denoted $\mathcal{I}_1 * \mathcal{I}_2$, is the set of interactions $\alpha \subseteq P_1 \cup P_2$ such that $\forall i,\ \alpha \cap P_i = \emptyset$ or $\alpha \cap P_i \in \mathcal{I}_i$.*

Note that this composition is associative and commutative. Also, there is no condition on the set of ports on which interaction models are defined for composing them. An interaction appears in the composition $\mathcal{I}_1 * \mathcal{I}_2$ if its projection onto P_1 and P_2 appears in respectively $\mathcal{I}_1$ and $\mathcal{I}_2$.

The composition $\circ$ of glues is then trivially defined as layerwise composition of the interaction and the priority layers: $\mathcal{I}^1_{\prec^1} \circ \mathcal{I}^2_{\prec^2} = (\mathcal{I}^1 * \mathcal{I}^2)_{\prec^1 \times \prec^2}$.

Definition 1.2.10 (Flat component) *A component is called* flat *if it is atomic or of the form $\mathcal{I}_\prec\{K_1, \ldots, K_n\}$, where all K_i are atomic components. A component that is not flat is called* hierarchical.

A hierarchical component K is of the form $\mathcal{I}_\prec\{K_1, \ldots, K_n\}$ such that at least one K_i is composite. Thus, such a K can be represented as $\mathcal{I}^2_{\prec^2}\{\mathcal{I}^1_{\prec^1}\{\mathcal{K}^1\}, \mathcal{K}^2\}$, where $\mathcal{K}^1$ and $\mathcal{K}^2$ are sets of components.

Definition 1.2.11 (Flattening of components) *The* flattened form *of a component K is defined inductively as follows:*

- *if K is a flat component, then its flattened form is equal to K.*
- *otherwise, K is of the form $\mathcal{I}^2_{\prec^2}\{\mathcal{I}^1_{\prec^1}\{\mathcal{K}^1\}, \mathcal{K}^2\}$, and then its flattened form is the flattened form of $(\mathcal{I}^2_{\prec^2} \circ \mathcal{I}^1_{\prec^1})\{\mathcal{K}^1 \cup \mathcal{K}^2\}$.*

Finally, the following theorem relates (when used inductively) the semantics of a hierarchical component and its flattened form.

Theorem 1.2.12 *$[\![\mathcal{I}^2_{\prec^2}\{\mathcal{I}^1_{\prec^1}\{\mathcal{K}^1\}, \mathcal{K}^2\}]\!]$ and $[\![(\mathcal{I}^2_{\prec^2} \circ \mathcal{I}^1_{\prec^1})\{\mathcal{K}^1 \cup \mathcal{K}^2\}]\!]$ are bisimilar.*

For the sake of clarity, we use the following notations:

- $\mathcal{K}^1 = \{K_k\}_{k=1}^m$ and $\mathcal{K}^2 = \{K_k\}_{k=m+1}^n$
- $Q^1 = Q_1 \times \ldots \times Q_m$ and $Q^2 = Q_{m+1} \times \ldots \times Q_n$
- $\mathcal{P}^1 = \bigcup_{k=1}^m \mathcal{P}_k$, $\mathcal{P}^2 = \bigcup_{k=m+1}^n \mathcal{P}_k$ and $\mathcal{P} = \mathcal{P}^1 \cup \mathcal{P}^2 = \bigcup_{k=1}^n \mathcal{P}_k$

Note that $\mathcal{I}^1$ is defined on $\mathcal{P}^1$ while $\mathcal{I}^2$ is defined on $\mathcal{P}$ (and not on $\mathcal{P}^2$).

Proof. Let $(B_h, \prec_h)$ and $(B_f, \prec_f)$ be the compositional semantics of respectively the hierarchical component $\mathcal{I}^2_{\prec^2}\{\mathcal{I}^1_{\prec^1}\{\mathcal{K}^1\}, \mathcal{K}^2\}$ and its flattened form $(\mathcal{I}^2_{\prec^2} \circ \mathcal{I}^1_{\prec^1})\{\mathcal{K}^1 \cup \mathcal{K}^2\}$. We show that $\prec_h = \prec_f$

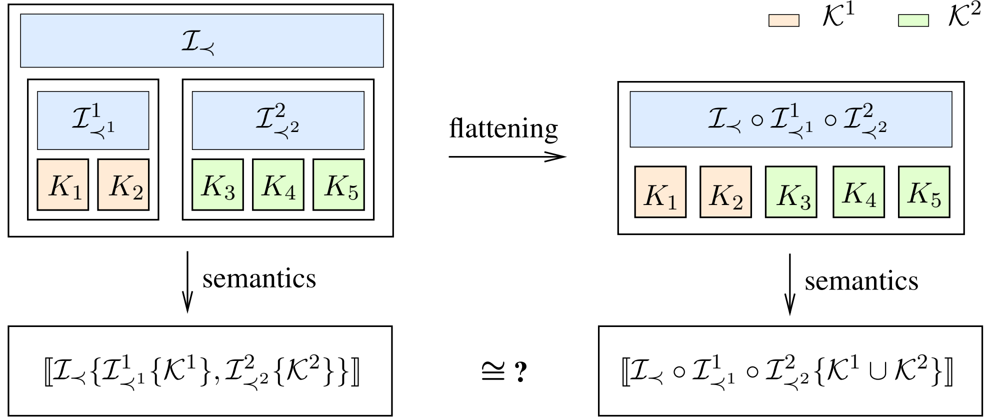


Figure 1.9 – Flattening of a hierarchical component

and that B_h and B_f are bisimilar. Denote $(B_i, \prec_i)$ the compositional semantics of K_i for $i \in [1, n_2]$, and $(B_s, \prec_s)$ that of component $\mathcal{I}^1_{\prec^1}\{\mathcal{K}^1\}$.

By definition, $\prec_s = \prec^1 \times \prec_1 \times \ldots \times \prec_m$ and then $\prec_h = \prec^2 \times \prec_s \times \prec_{m+1} \times \ldots \times \prec_n$. Also by definition, $\prec_f = (\prec^2 \times \prec^1) \times \prec_1 \times \ldots \times \prec_n$. Hence $\prec_h = \prec_f$.

Let us now show the bisimulation. We define $\mathcal{R} \subseteq ((Q^1) \times Q^2) \times (Q^1 \times Q^2)$ as:

$$((q_1, \ldots, q_m), q_{m+1}, \ldots, q_n) \, \mathcal{R} \, (q'_1, \ldots, q'_n) \triangleq \forall i \in [1, n] : q_i = q'_i$$

We show that this relation is a bisimulation. The initial states are trivially related.

Let us show that $((q_1, \ldots, q_m), q_{m+1}, \ldots, q_n) \overset{\alpha}{\rightsquigarrow}_h ((q'_1, \ldots, q'_m), q'_{m+1}, \ldots, q'_n)$ if and only if $(q_1, \ldots, q_n) \overset{\alpha}{\rightsquigarrow}_f (q'_1, \ldots, q'_n)$ for $\alpha \in 2^{\mathcal{P}}$. Let $\alpha_s = \alpha \cap \mathcal{P}^1$ and $\alpha_i = \alpha \cap \mathcal{P}_i$ for $i \in [1, n]$.

By definition, $((q_1, \ldots, q_m), q_{m+1}, \ldots, q_n) \overset{\alpha}{\rightsquigarrow}_h ((q'_1, \ldots, q'_m), q'_{m+1}, \ldots, q'_n)$ if and only if:

- $\alpha \in \mathcal{I}^2$
- either $\alpha_s = \emptyset$ and $(q_1, \ldots, q_m) = (q'_1, \ldots, q'_m)$; or $(q_1, \ldots, q_m) \overset{\alpha_s}{\rightsquigarrow}_s (q'_1, \ldots, q'_m)$
- for all $i \in [m+1, n]$, either $\alpha_i = \emptyset$ and $q_i = q'_i$; or $q_i \overset{\alpha_i}{\rightsquigarrow}_i q'_i$

Similarly, $(q_1, \ldots, q_n) \overset{\alpha}{\rightsquigarrow}_f (q'_1, \ldots, q'_n)$ if and only if:

- $\alpha \in \mathcal{I}^2 * \mathcal{I}^1$
- for all $i \in [1, n]$, either $\alpha_i = \emptyset$ and $q_i = q'_i$; or $q_i \overset{\alpha_i}{\rightsquigarrow}_i q'_i$

Let us suppose that $((q_1, \ldots, q_m), q_{m+1}, \ldots, q_n) \overset{\alpha}{\rightsquigarrow}_h ((q'_1, \ldots, q'_m), q'_{m+1}, \ldots, q'_n)$ and show that $(q_1, \ldots, q_n) \overset{\alpha}{\rightsquigarrow}_f (q'_1, \ldots, q'_n)$.

If $\alpha_s = \emptyset$, then as $\alpha \cap \mathcal{P} \in \mathcal{I}^2$ and $\alpha \cap \mathcal{P}^1 = \emptyset$, we have by definition of $*$ that $\alpha \in \mathcal{I}^2 * \mathcal{I}^1$. Besides, $\alpha_s = \emptyset$ implies that $\forall i \in [1,m] : \alpha_i = \emptyset$ and $q_i = q'_i$, hence $(q_1, \ldots, q_n) \overset{\alpha}{\rightsquigarrow}_f (q'_1, \ldots, q'_n)$.

If $\alpha_s \neq \emptyset$, then $(q_1, \ldots, q_m) \overset{\alpha_s}{\rightsquigarrow}_s (q'_1, \ldots, q'_m)$, which implies that $\alpha_s \in \mathcal{I}^1$ and $\forall i \in [1,m]$, either $\alpha_i = \emptyset$ and $q_i = q'_i$; or $q_i \overset{\alpha_i}{\rightsquigarrow}_i q'_i$. Besides, we have: $\alpha \cap \mathcal{P} \in \mathcal{I}^2$ and $\alpha \cap \mathcal{P}^1 \in \mathcal{I}^1$, so $\alpha \in \mathcal{I}^2 * \mathcal{I}^1$. As a result, $(q_1, \ldots, q_n) \overset{\alpha}{\rightsquigarrow}_f (q'_1, \ldots, q'_n)$.

Now, let us suppose that $(q_1, \ldots, q_n) \overset{\alpha}{\rightsquigarrow}_f (q'_1, \ldots, q'_n)$. There remains for us to show that $((q_1, \ldots, q_m), q_{m+1}, \ldots, q_n) \overset{\alpha}{\rightsquigarrow}_h ((q'_1, \ldots, q'_m), q'_{m+1}, \ldots, q'_n)$. As $\mathcal{I}^2$ is defined on $\mathcal{P}$, $\alpha \in \mathcal{I}^2 * \mathcal{I}^1$ implies that $\alpha \in \mathcal{I}^2$ (because α is an interaction and thus cannot be empty).

If $\forall i \in [1,m] : \alpha_i = \emptyset$, then $\alpha_s = \emptyset$ and it holds that $(q_1, \ldots, q_m) = (q'_1, \ldots, q'_m)$. Otherwise, that is if $\alpha_s \neq \emptyset$, then $\alpha_s \in \mathcal{I}^2 * \mathcal{I}^1$ implies that $\alpha_s \in \mathcal{I}^1$ and so $(q_1, \ldots, q_m) \overset{\alpha_s}{\rightsquigarrow}_s (q'_1, \ldots, q'_m)$. Hence the result. □

1.2.2 Interface theories

Originally, contracts were introduced in the context of object-oriented programming [Mey92]. Since then, they have been adapted to many other application domains, e.g. to Web services [PS07, BZ07, Pad08, CGP08], where the emphasis is on compatibility and independent implementability (also called substitutability).

Contracts are of particular interest for speculative design, where distributed teams use a notion of rich component and contract for working concurrently on partial designs. *Interfaces* [GSL96, dAH01a] have been proposed for this purpose and focus on incremental design through composition of interfaces. As for contracts, there exist many different interface theories: timed interfaces [dAHS02], resource interfaces [CdAHS03], Web service interfaces [BCH05], relational interfaces [TLHL09] etc. Interface theories are based on a fixed model of composition — usually synchronous input/output (I/O) composition. Any interface theory must ensure *independent implementability*, that is, that implementations can be derived from specifications independently of each other and *associativity*, i.e., the order of composition does not affect the properties of the global system.

In their seminal paper [dAH01a] and later in [dAH01b] and [dAH05], de Alfaro and Henzinger introduced *interface automata* as behavioral contracts. An interface automaton describes how a component and its environment are expected to interact via input/output composition. As interface automata are not necessarily input-enabled, a single automaton represents both the *assumption* made by the component on its environment and the *guarantee* provided by this component:

- Assumption: an output transition $t!$ in the interface automaton must correspond to an input transition $t?$ in the environment. Indeed, $t!$ is considered legal by the interface, thus it must be accepted by the environment. Besides, absence of an input transition $t?$ in the interface automaton must correspond to absence of a corresponding output $t!$ transition in the environment. That is, an implementation of the interface is not required to accept t, so the environment must not offer it.
- Guarantee: an input transition in the interface automaton must correspond to an input transition in the implementation, and absence of an output transition in the interface automaton must correspond to absence of a corresponding output transition in the implementation.

As a direct consequence, refinement is defined as alternating simulation [AHKV98]: a implementation accepts more inputs and provides fewer outputs than its interface.

In [dAH01a], the motivation for interfaces is to check *compatibility* in open systems: two interfaces are compatible if there exists an environment in which they can be used together, that is, the guarantee of one does not violate the assumption of the other. This approach is optimistic in contrast with component refinement, where components have to respond to any environment, thus leading to a heavy defensive design style. Note that working with an open system as here implicitly defines an interface for the environment matching that of the component, thus closing the system — but allowing the environment interface to be refined as components are added.

In [LNW06], Larsen, Nyman and Wąsowski introduce interface I/O automata, which are inspired from interface automata, but differ in several ways. An I/O interface automaton consists of two I/O automata [Lyn96], thus separating explicitly the assumption (called the *environment*) and the guarantee (called the *specification*). Unlike interface automata, such I/O automata are input-enabled. [LNW06] then focuses on composition of interfaces and provides a system of inequalities whose maximal solution it the most general composition. Two operations, *zip* and *unzip* allow translation from interface automata to interface I/O automata.

An extension of interface I/O automata with modality has been proposed in [LNW07]. However, the separation between assumption and guarantee is dropped there. This approach was developed further by Raclet et al. in [RBB$^+$09b, RBB$^+$09a] using residuation [Rac08] and allowing for multiple viewpoints and component reuse [DHJP08].

Keeping assumption and guarantee separate has several advantages. First, it improves reusability, as checking refinement under a new context can be achieved much more easily, for example if the new environment refines the old one in any context. Second, an interface is necessarily a composition representing how the global (closed) system should behave. Keeping assumption and guarantee

separate — the guarantee then representing a component property — avoids storing their product.

The approach proposed in [BCP07, BCF$^+$07, BFM$^+$08] for the L0 framework of the SPEEDS project is again based on disjoint assumptions and guarantees. There, a new issue appears, called consistency: the assumption and the guarantee of a contract may be contradictory. This problem occurs neither for interface automata nor for interface I/O automata, because an interface automaton is an implementation of itself, and the guarantee of an interface I/O automaton is an implementation of this automaton.

Our setting in this thesis is more general than all these approaches: we use *glues*, that is, sets of operators closed by composition which map sets of behaviors into behaviors [Sif05]. We do not suppose the existence of an algebra of contracts allowing composition and conjunction of contracts. Indeed, in frameworks for which the least upper bound of two components with the same interface is not defined, composition and conjunction of contracts are not always definable. Besides, in frameworks with rendezvous interaction, there is no *unzip* function from a single interface to a pair assumption/guarantee — only a relation. Hence, we keep assumptions and guarantees separate. In our framework, consistency is irrelevant for the same reason as for interface I/O automata. Finally, we do not address compatibility in its generality but we provide a structural counterpart for this notion as well as for consistency.

Chapter 2

Defining contract frameworks

In this chapter, we present the design methodology that we propose. In particular, we give a generic definition of contract framework, and state the necessary ingredients for using circular reasoning within such a framework.

2.1 Methodology

From a macroscopic point of view, we adopt a top-down design and verification methodology in which high-level requirements are pushed progressively from the level of the system to the level of atomic components — which we call implementations. As usual, this is just a convenient representation; in real life, the final picture is always achieved after several iterations alternatively going up and down the hierarchy. So far, we keep the notion of component very abstract, as this methodology applies to any component framework, provided some conditions — which will be listed later — are fulfilled. Our approach is based on contracts which we suppose in general to be provided by the system designer: building a contract for a (possibly hierarchical) component is considered as a design step.

For a component K, a *contract* describes 1) the interface $\mathcal{P}_K$ of K 2) the interaction gl between K and its environment (denoted E), and 3) an abstraction A of the expected behavior of E and an abstraction G of the promised behavior of K. The idea is that the system component is refined into a set of subcomponents assembled using a complex interaction layer. Contracts are associated with each of the subcomponents in such a way that if we can build implementations satisfying the contracts of the subcomponents, then their composition satisfies the system contract.

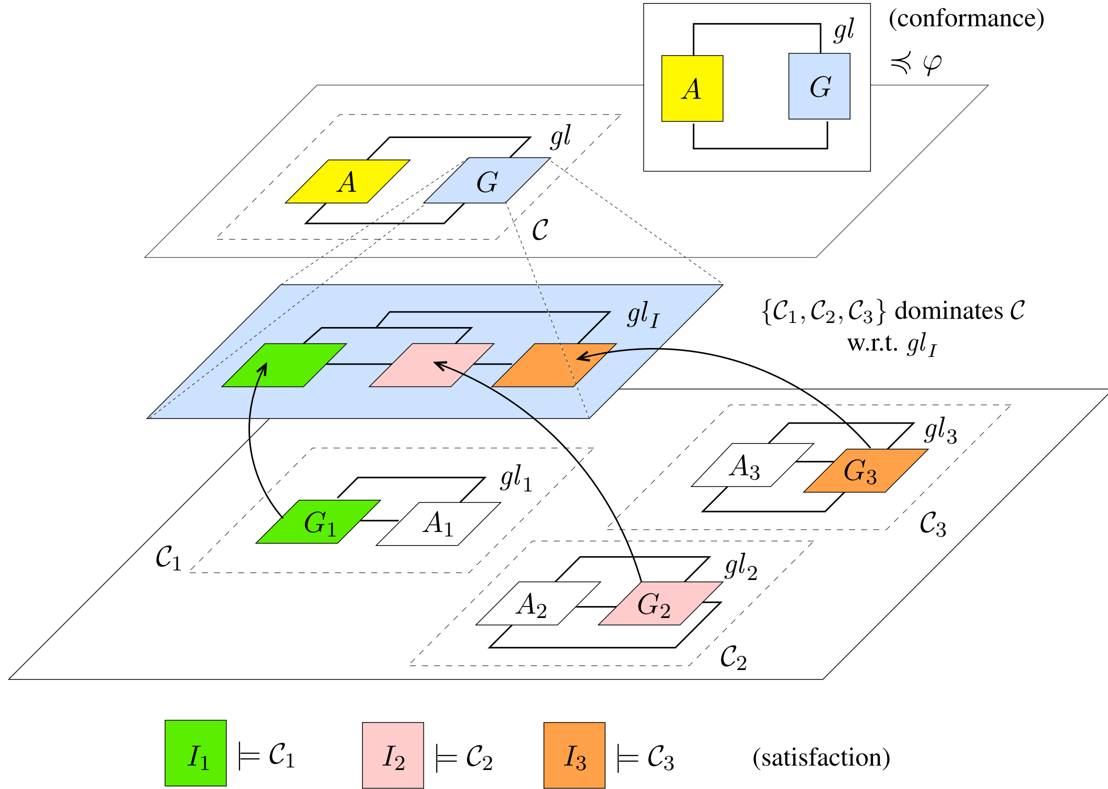


Figure 2.1 – Method for proving $gl\{E, gl_I\{I_1, I_2, I_3\}\} \preccurlyeq \varphi$

Our methodology is illustrated in Figure 2.1. We suppose given a global requirement φ which the system K under construction, together with an environment abstracted by a property A, has to satisfy. Both φ and A are expressed w.r.t. the interface $\mathcal{P}_K$ of K. We proceed as follows: (1) define a *contract* $\mathcal{C}$ for $\mathcal{P}_K$ which *conforms* to φ; (2) define K as a composition of subcomponents K_i and a contract $\mathcal{C}_i$ for each of them; possibly iterate this step if needed; (3) prove that any set of implementations I_i for K_i *satisfying* the contracts $\mathcal{C}_i$, when composed, satisfy the top-level contract $\mathcal{C}$ (*dominance*) — and thus guarantee φ; (4) provide such implementations.

The global requirement φ appears at the top, while the implementations I_i are at the bottom. Note that K is not directly represented in the figure: it has the same interface as G and is obtained by composing the implementations I_i with gl_I. That is, $K = gl_I\{I_1, I_2, I_3\}$. Similarly, E appears only in the conclusion, namely that $gl\{E, K\} \preccurlyeq \varphi$.

The correctness proof for a particular system is split into 3 phases: *conformance* (denoted $\preccurlyeq$) of the top-level contract $\mathcal{C}$ to φ, *dominance* between the contracts $\mathcal{C}_i$ and $\mathcal{C}$, and *satisfaction* (denoted $\models$)

of the $\mathcal{C}_i$ by the implementations I_i. Thus, *conformance* relates properties of closed systems — as a contract defines a closed system made of the composition of its assumption and guarantee — while *dominance* relates contracts and *satisfaction* relates components to contracts.

2.2 Definitions

Let us define now the various notions that have been informally introduced in the previous section. We develop a generic framework that supports hierarchical components and mechanisms to reason about composition. The following notions and properties form the basis of this framework. We use *glue* operators [Sif05] to generalize the parallel composition found in most traditional frameworks.

2.2.1 Component framework

The notion of component is intentionally kept very abstract to encompass various frameworks. It may be e.g. a labeled transition system, but it may also have a structural part, e.g. be a BIP component. We will see several examples of such component frameworks in the next chapters, some providing a low-level semantic representation of components and some a high-level syntactic one.

Definition 2.2.1 (Component framework) *A* component framework *is a tuple* $(\mathcal{K}, \cong, GL, \circ)$ *where:*

- $\mathcal{K}$ *is a set of* components. *Each component* $K \in \mathcal{K}$ *has as* interface *a set of* ports*, denoted* $\mathcal{P}_K$.
- $\cong \subseteq \mathcal{K} \times \mathcal{K}$ *is an equivalence relation. In general, this equivalence is derived from equality or equivalence of semantic sets.*
- GL *is a set of* glue (composition) operators. *A glue is a partial function* $2^{\mathcal{K}} \longrightarrow \mathcal{K}$ *transforming a set of components into a new component. Each* $gl \in GL$ *is associated with a set of ports* S_{gl} *from the original set of components — called its* support set *— and a new interface* $\mathcal{P}_{gl}$ *for the new component — called its* exported interface. *A composition* $K = gl(\{K_1, \ldots, K_n\})$ *is defined if* $K_1, \ldots, K_n \in \mathcal{K}$ *have disjoint interfaces,* $S_{gl} = \bigcup_{i=1}^{n} \mathcal{P}_{K_i}$ *and the interface of* K *is* $\mathcal{P}_{gl}$*, the exported interface of* gl.
- $\circ$ *is a partial operation on* GL *to hierarchically compose glues.* $gl \circ gl'$ *is defined if* $\mathcal{P}_{gl'} \subseteq S_{gl}$. *Then, its support set is* $S_{gl} \setminus \mathcal{P}_{gl'} \cup S_{gl'}$ *and its interface is* $\mathcal{P}_{gl}$ *(see Figure 2.2). Furthermore, this operation must be consistent with* $\cong$ *in the sense that* $gl\{gl'\{\mathcal{K}^1\}, \mathcal{K}^2\} \cong (gl \circ gl')\{\mathcal{K}^1 \cup \mathcal{K}^2\}$ *for any sets of components* $\mathcal{K}^i$ *such that all terms are defined. It is left-associative.*

To simplify notation, we write $gl\{K_1, \ldots, K_n\}$ instead of $gl(\{K_1, \ldots, K_n\})$.

Figure 2.2 shows how hierarchical components (the colored ones) are built from atomic ones (the white components). Incidentally, there is no explicit distinction between atomic and hierarchical

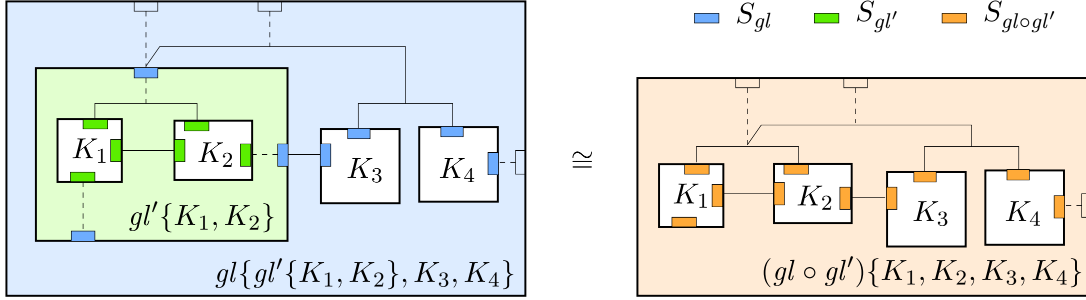


Figure 2.2 – A hierarchical component and its equivalent “flattened” form

components: if there exists a way of representing a hierarchical component as an atomic one, this has to be expressed as a function preserving equivalence. We made this choice because in frameworks working at the semantic level, atomic and hierarchical components are the same. For example, in the BIP semantic framework of Chapter 4 where components are labeled transition systems (LTS), a composition of LTS is an LTS.

Figure 2.2 also illustrates the coherence condition that $\circ$ must ensure. Note that the representation of glues — drawing full lines for representing connectors which can even be hierarchical — is just one among other possible sets of glues. Dashed lines show how the exported interface is built based on inner ports and connectors. This representation is inspired by the BIP framework.

2.2.2 Contract framework

We now have to define the relations mentioned in the methodology, namely: conformance, satisfaction and dominance. In general, satisfaction and dominance are derived from the definition of conformance. Here, we loosen the coupling between these relations in order to obtain stronger reasoning schemata for dominance. More precisely, we introduce a relation called *refinement under context*, from which we derive satisfaction and dominance. Refinement under context is related to, but not necessarily derived from, conformance.

Before giving the formal definition of contract framework, we introduce the notion of *context*, first to describe how a component may be connected to its environment (i.e. the rest of the system) and then to express a property of this environment. Thus, a context restricts how a component may be further composed. In the sequel, the set of contexts associated with a component framework is denoted Ω.

Definition 2.2.2 (Context) *A* context *for an interface $\mathcal{P}$ is a pair (E, gl) where $E \in \mathcal{K}$ is such that $\mathcal{P} \cap \mathcal{P}_E = \emptyset$ and $gl \in GL$ is defined on $\mathcal{P} \cup \mathcal{P}_E$.*

Definition 2.2.3 (Contract framework) *A* contract framework *is defined by a structure of the form* $(\mathcal{K}, \cong, GL, \circ, \preccurlyeq, \{\sqsubseteq_\omega\}_{\omega\in\Omega})$ *where:*

- $(\mathcal{K}, \cong, GL, \circ)$ *is a component framework, whose set of contexts is* Ω*.*
- $\preccurlyeq \subseteq \mathcal{K} \times \mathcal{K}$ *is a* conformance *relation, that is, a preorder over the set of components with the same interface.*
- $\{\sqsubseteq_\omega\}_{\omega\in\Omega}$ *is a set of* refinement under context *relations, one for each context* ω *in* Ω*, such that:*
  - *given a context* ω *for an interface* $\mathcal{P}$*,* $\sqsubseteq_\omega$ *is a preorder over the set of components on* $\mathcal{P}$
  - *for any* K_1*,* K_2 *on the same interface* $\mathcal{P}$ *and for any context* (E, gl) *for* $\mathcal{P}$*, it holds that: if* $K_1 \sqsubseteq_{E,gl} K_2$*, then* $gl\{K_1, E\} \preccurlyeq gl\{K_2, E\}$
  - *for any* K_1*,* K_1'*,* K_2 *on the same interface* $\mathcal{P}$ *and for any context* (E, gl) *for* $\mathcal{P}$*, we have: if* $K_1 \cong K_1'$*, then* $K_1 \sqsubseteq_{E,gl} K_2$ *if and only if* $K_1' \sqsubseteq_{E,gl} K_2$*.*

The coherence conditions that must be satisfied by the refinement under context relation — one with respect to conformance and one with respect to equivalence — come from the fact that we have chosen in this definition to keep these three notions decoupled: we still need to make sure that refinement under context allows deducing conformance for closed systems, and that equivalent components have the same refinement properties.

Example 2.2.4 *Typical notions of conformance* $\preccurlyeq$ *when components are labeled transitions systems (LTS) are* trace inclusion *(see Definition 1.1.4) and its structural counterpart,* simulation *(Definition 1.1.11).*

Example 2.2.5 *Refinement under context (denoted* $\sqsubseteq^{\preccurlyeq}$*) is usually defined as the weakest preorder implying conformance and preserved by composition:*

$$K_1 \sqsubseteq^{\preccurlyeq}_{E,gl} K_2 \triangleq gl\{K_1, E\} \preccurlyeq gl\{K_2, E\}$$

Example 2.2.6 *Conformance itself is another candidate for refinement under context if it is preserved by composition, that is, if:* $K_1 \preccurlyeq K_2$ *implies* $gl\{K_1, E\} \preccurlyeq gl\{K_2, E\}$ *for any* E*.*

However, defining refinement under context as $K_1 \sqsubseteq_{E,gl} K_2 \triangleq K_1 \preccurlyeq K_2$ *means in fact not taking the environment into account, thus it is of limited interest. In some cases, as in Chapter 7, conformance actually corresponds to refinement in any context. This is not always the case, and a counter-example is given in Chapter 7.*

Notation 2.2.7 (Refinement in any context) *If* K_1 *refines* K_2 *in any context for* $\mathcal{P}_{K_1} = \mathcal{P}_{K_2}$ *then we omit the context and simply write* $K_1 \sqsubseteq K_2$*.*

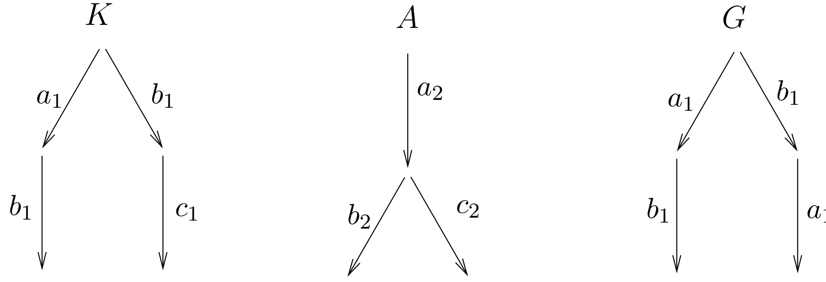


Figure 2.3 – $K \sqsubseteq^{\preccurlyeq}_{A,gl} G$ for conformance defined as simulation.

Definition 2.2.8 (Contract) *A* contract $\mathcal{C}$ *for an interface* $\mathcal{P}$ *consists of:*

- *a context* $\mathcal{E} = (A, gl)$ *for* $\mathcal{P}$*; A is called the* assumption
- *a component G on* $\mathcal{P}$ *called the* guarantee

We write $\mathcal{C} = (A, gl, G)$ rather than $\mathcal{C} = ((A, gl), G)$. The interface of the environment is implicitly defined by gl while A expresses a constraint on it and G a constraint on the refinements of K. The "mirror" contract $\mathcal{C}^{-1}$ of $\mathcal{C}$ is (G, gl, A), i.e. a contract for the environment. In the rest of this document, a contract $\mathcal{C}_i$ is implicitly defined as (A_i, gl_i, G_i).

Definition 2.2.9 (Satisfaction) *A component K* satisfies *a contract* $\mathcal{C} = (A, gl, G)$*, denoted* $K \models \mathcal{C}$*, if and only if* $K \sqsubseteq_{A,gl} G$.

Example 2.2.10 *Suppose that components are LTS, conformance is simulation and refinement under context is the usual derived notion defined in Example 2.2.5. Suppose also that composition gl is defined as the synchronization between actions with the same letter (a_1 synchronizes with a_2 and the corresponding label in the composition is $a_1|a_2$ etc.) and interleaving of others. Then Figure 2.3 shows K, A and G such that K satisfies the contract (A, gl, G). Indeed, even though K does not simulate G (after b_1 is fired it offers c_1 instead of a_1), it still behaves like G in the context of (A, gl), which prevents b_1 from taking place.*

A contract $\mathcal{C} = (A, gl, G)$ defines a closed system, namely $gl\{A, G\}$. Thus we say that $\mathcal{C}$ *conforms to* a property φ (as required by the methodology of Figure 2.1) when $gl\{A, G\}$ conforms to φ. In some interface theories [GSL96, dAH01a], the component $gl\{A, G\}$ is used to represent the contract — which consists only of A and G, as gl is predefined. This is possible because there is only one maximal pair (A, G) corresponding to an interface. This does not hold in general. Indeed, in frameworks with rendezvous interaction, several pairs (A, G) can correspond to the same interface

(see Example 2.2.11), as the component and its environment can both prevent a rendezvous from taking place. This is one reason for keeping assumptions and guarantees separate.

Example 2.2.11 *Consider the same framework as in Example 2.2.10 and suppose that $gl\{A, G\}$ is an LTS with only one label $a_1|a_2$ that does nothing. Since $gl\{A, G\}$ is obtained by synchronization, it is sufficient that at least one of the two component forbids its action a_i for the rendezvous not to take place. There are thus three possible incomparable pairs (A, G) of length 1: one where A does noting but G offers a_2, a second where A offers a_1 but G does nothing and the third where both do nothing.*

A second reason for keeping assumption and guarantee distinct in our definition of contract (as is the case in the interface I/O automata of [LNW06]) is related to the design process: A is intended to be an abstraction of the environment of the component while G is an abstraction of the expected behavior of the component under study. Thus, if some hypotheses are modified during the design process, assumption and guarantee can be modified independently of each other.

Another advantage of our notion of contract with respect to system design is its structural part. It allows us to separate the architecture and the requirements of the system under construction, which evolve independently during the development process. In particular, in frameworks where interaction is rich, refinement can be ensured by relying heavily on the structure of the system and less importantly on the behavioral properties of the environment. This is discussed in Chapter 7.

So, we now have, given a contract framework, a definition of conformance and satisfaction. Dominance is defined in the next section.

2.2.3 Dominance

Dominance is the key notion that distinguishes reasoning in a contract framework from reasoning in a component framework. Intuitively, dominance can be seen as a refinement relation between contracts. More specifically, a contract $\mathcal{C}$ dominates a contract $\mathcal{C}'$ whenever every implementation of $\mathcal{C}$, that is, every component satisfying $\mathcal{C}$, is also an implementation of $\mathcal{C}'$. Thus dominance is a very strong property, as it implies that *all* components satisfying the dominating contract ($\mathcal{C}$) also satisfy the dominated one ($\mathcal{C}'$). Once dominance has been established, there is no need to handle implementations anymore, only assumptions and guarantees which are expected to be much smaller. More specifically, assumptions and guarantees should be of the same complexity at each level of hierarchy while implementations go much larger.

In this section, we provide a definition of dominance involving a composition of components for which we will provide later a sufficient condition for proving such dominance without having to deal

with actual implementations. Let us start with a first definition of dominance for two contracts with the same glue part.

Definition 2.2.12 (Binary dominance) *Let $\mathcal{C}$ and $\mathcal{C}'$ be two contracts for the same interface $\mathcal{P}$, with $\mathcal{C} = (A, gl, G)$ and $\mathcal{C}' = (A', gl', G')$ — implying that $\mathcal{P}_G = \mathcal{P}_{G'}$. $\mathcal{C}$* dominates *$\mathcal{C}'$ if and only if $gl = gl'$ (and as a consequence $\mathcal{P}_A = \mathcal{P}_{A'}$) and:*

$$\text{for any } K \text{ on } \mathcal{P}, \text{ if } K \models \mathcal{C} \text{ then } K \models \mathcal{C}'$$

The question of how to relate contracts which are defined on different interfaces, or which have equivalent rather than equal glues, is discussed in Section 3.1.

If one cannot compose contracts or wants to avoid it (see discussion in Section 3.2), a dominance check involves in general not just a pair of contracts. A typical situation would be the one depicted in Figure 2.1, where a set of contracts $\{\mathcal{C}_i\}_{i=1}^n$ are attached to disjoint interfaces $\{\mathcal{P}_i\}_{i=1}^n$. Besides, a glue gl_I is defined on $S_{gl_I} = \bigcup_{i=1}^n \mathcal{P}_i$ and a contract $\mathcal{C}$ is given for $\mathcal{P}_{gl_I}$.

We thus need a broader notion of dominance than the binary version presented above: a set of contracts $\{\mathcal{C}_i\}_{i=1}^n$ dominates a contract $\mathcal{C}$ w.r.t. a glue gl_I if and only if any set of components satisfying the contracts $\mathcal{C}_i$, when composed using gl_I, makes a component satisfying $\mathcal{C}$. Formally, this is defined as follows.

Definition 2.2.13 (Dominance for a composition) *Let $\mathcal{C}$ be a contract for $\mathcal{P}$, $\{\mathcal{C}_i\}_{i=1}^n$ a set of contracts for respectively $\{\mathcal{P}_i\}_{i=1}^n$ and gl_I a glue such that $S_{gl_I} = \bigcup_{i=1}^n \mathcal{P}_i$ and $\mathcal{P}_{gl_I} = \mathcal{P}$. Then $\{\mathcal{C}_i\}_{i=1}^n$* dominates $\mathcal{C}$ with respect to *gl_I iff for any set of components $\{K_i\}_{i=1}^n$ on respectively $\{P_i\}_{i=1}^n$:*

$$\text{if for every } i \in [1, n],\ K_i \models \mathcal{C}_i, \text{ then } gl_I\{K_1, \ldots, K_n\} \models \mathcal{C}$$

How do we prove dominance? The definition we have given is semantic, and concretely we do not want to manipulate implementations in order to establish dominance. More generally, what are the additional proof rules that we need in order to reason within contract frameworks as they are introduced? The following section answers those questions.

2.3 Reasoning within a contract framework

Compositional reasoning means proving properties of a system based on local properties of its components. It is usually based on the following rule: if an implementation I conforms to its specification S, then whenever composed with any component E it still conforms to S — this is called

composability — and so $I \parallel E$ conforms to $S \parallel E$, where $\parallel$ denotes generically parallel composition. This rule allows concluding from $I_1 \preccurlyeq S_1$ and $I_2 \preccurlyeq S_2$ that $I_1 \parallel I_2 \preccurlyeq S_1 \parallel S_2$. The proof given below uses this rule twice, once for composing I_1 with I_2, the second time for composing I_2 with S_1. Then, the result is obtained by commutativity of parallel composition and transitivity of conformance.

$$\frac{\dfrac{I_1 \preccurlyeq S_1}{I_1 \parallel I_2 \preccurlyeq S_1 \parallel I_2} \qquad \dfrac{I_2 \preccurlyeq S_2}{I_2 \parallel S_1 \preccurlyeq S_2 \parallel S_1}}{I_1 \parallel I_2 \preccurlyeq S_1 \parallel S_2}$$

2.3.1 Compositionality

Let us consider now refinement under context and see how we can adapt the previous rule to our problem. First, composability obviously does not hold anymore: an implementation refines a specification in a given context, and there is no guarantee about what happens in another context. The only rule that is specified in our definition of contract framework is that refinement under context implies conformance: $I \sqsubseteq_{Egl} S$ implies $gl\{I, E\} \preccurlyeq gl\{S, E\}$. This rule is not sufficient to derive $gl\{I_1, I_2\} \preccurlyeq gl\{S_1, S_2\}$ from $I_1 \sqsubseteq_{I_2,gl} S_1$ and $I_2 \sqsubseteq_{I_1,gl} S_2$. In fact, we only have:

$$\frac{I_1 \sqsubseteq_{I_2,gl} S_1}{gl\{I_1, I_2\} \preccurlyeq gl\{S_1, I_2\}} \qquad \frac{I_2 \sqsubseteq_{I_1,gl} S_2}{gl\{I_2, I_1\} \preccurlyeq gl\{S_2, I_1\}}$$

This means that we need the following, stronger rule if we want to apply compositional reasoning:

$$\frac{I_1 \sqsubseteq_{I_2,gl} S_1 \qquad I_2 \sqsubseteq_{I_1,gl} S_2}{gl\{I_1, I_2\} \preccurlyeq gl\{S_1, S_2\}}$$

This rule is still not sufficient because it does not allow *incremental design*, i.e. incorporating parts of the environment into the component under study: indeed, it only allows proving conformance, not refinement under context. It is thus useless if one wants to establish dominance. So we generalize again this rule by introducing a environment E for I_1 and I_2:

$$\frac{I_1 \sqsubseteq_{gl_{E_2}\{I_2,E\},gl_1} S_1 \qquad I_2 \sqsubseteq_{gl_{E_1}\{I_1,E\},gl_2} S_2 \qquad gl_1 \circ gl_{E_2} = gl_2 \circ gl_{E_1} = gl_E \circ gl}{gl\{I_1, I_2\} \sqsubseteq_{E,gl_E} gl_1\{S_1, S_2\}}$$

We will actually always use a weaker version of this rule. The reason is that we will never use this rule alone, but always in conjunction with other proof rules, so we only need the following property.

Definition 2.3.1 (Compositionality) *A set of refinement under context relations $\{\sqsubseteq_\omega\}_{\omega\in\Omega}$ is said to*

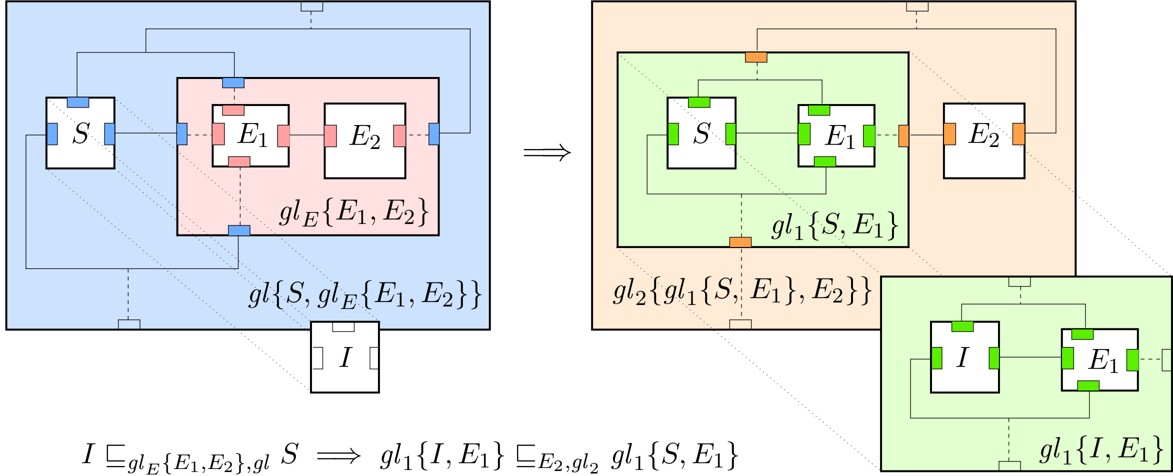


Figure 2.4 – Compositionality, i.e. preservation of refinement by composition

be preserved by composition *if and only if the following rule applies whenever all terms are defined:*

$$\frac{I \sqsubseteq_{E,gl} S \qquad E = gl_E\{E_1, E_2\} \qquad gl \circ gl_E = gl_2 \circ gl_1}{gl_1\{I, E_1\} \sqsubseteq_{E_2,gl_2} gl_1\{S, E_1\}} \quad (\textbf{CMP})$$

This property, which we call compositionality*, is illustrated in Figure 2.4.*

Note that provided it is always possible to find gl_{E_1} and gl_{E_2} such that $gl_2 \circ gl_1 = gl_{E_1} \circ gl_{E_2}$ and all terms below are defined, compositionality is weaker than the previous proof rule. This comes directly by using reflexivity of refinement under context before applying that rule:

$$\frac{I \sqsubseteq_{gl_E\{E_1,E_2\},gl} S \qquad E_1 \sqsubseteq_{gl_{E_2}\{I,E_2\},gl_{E_1}} E_1 \qquad gl \circ gl_E = gl_2 \circ gl_1 = gl_{E_1} \circ gl_{E_2}}{gl_1\{I, E_1\} \sqsubseteq_{E_2,gl_2} gl_1\{S, E_1\}}$$

The additional condition about the existence of glues allowing decomposition of the system as $gl_{E_1}\{E_1, gl_{E_2}\{I, E_2\}\}$ may be puzzling. It arises from the fact that the conditions expected from the set of glues GL and its composition operator $\circ$ in the definition of component framework are very limited: we only require that there always exists a flattened form of hierarchical components.

2.3.2 Circular reasoning

So, we now have a proof rule which relates different refinement under context relations. However, this rule does not allow to derive *independent implementability*. That is, the premises of (**CMP**)

require that an implementation refine its specification in the *actual* context in which it is used. This is highly undesirable for at least two reasons: one is that implementations are expected to be very complex, thus manipulating them is likely to be intractable; the other is that whenever a small change occurs in the implementation of a part of the system, *all* the proofs have to be started all over again. To avoid this situation, we need another property of refinement which allows proving refinement using the abstract environment provided by the specifications rather than the concrete one provided by the implementations. Most frameworks offer the following rule:

Definition 2.3.2 (Assume-guarantee) *If a component K refines G in an abstract context (A, gl) and if E refines A in any context, then K also refines G in the concrete context (E, gl).*

$$\frac{K \sqsubseteq_{A,gl} G \qquad E \sqsubseteq A}{K \sqsubseteq_{E,gl} G} \quad (\textbf{AG})$$

This rule is in general called assume-guarantee, because the component assumes a property that is satisfied by its environment and guarantees a property based on this assumption. It is quite limited because the environment has to satisfy a much stronger property than the component. As the environment of one component consists of other components, this means that in order to apply this rule one has to find a way of "breaking the symmetry" of the dependency between component and environment by finding a component which can guarantee its property independently of its environment. This property may then be used as an assumption for a second component etc. When the system is complex and there are many component, finding such a proof strategy may be very complicated. This is why the following rule, which implies the previous one and is commonly referred to as circular reasoning because it is symmetric, is more interesting.

Definition 2.3.3 (Circular reasoning) *If a component K refines G in an abstract context (A, gl) and if E refines A in the abstract context (G, gl), then K refines G in the concrete context (E, gl).*

$$\frac{K \sqsubseteq_{A,gl} G \qquad E \sqsubseteq_{G,gl} A}{K \sqsubseteq_{E,gl} G} \quad (\textbf{CR})$$

This property can be proved in a given framework by an induction based on the semantics of composition and refinement [McM99, Mai03b]. Chapter 4 shows a framework in which (**CR**) is sound.

However, circular reasoning is not sound in general. In particular, it is unsound when composition is based on synchronizations (as they exist in e.g. in Petri nets or process algebras) or instantaneous mutual dependencies between inputs and outputs (as they exist in synchronous formalisms). Example 2.3.4 explains two reasons for the non validity of circular reasoning for $\sqsubseteq^{\preccurlyeq}$ which are illustrated

in Figures 2.5 and 2.6.

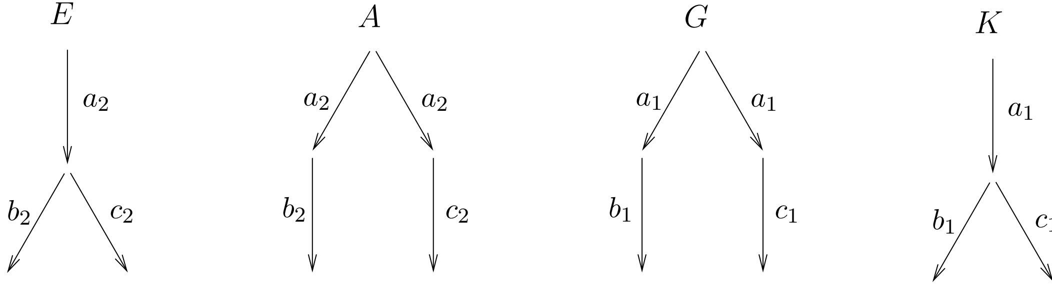


Figure 2.5 – A counterexample to (**CR**) due to non-determinism.

Figure 2.6 – A counterexample to (**CR**) due to strong synchronization.

Example 2.3.4 *Suppose, as in the previous examples, that components are LTS and composition gl is defined as the synchronization between actions with the same letter and interleaving of others. Suppose also that conformance is simulation and refinement under context is the usual derived notion:*

$$K_1 \sqsubseteq^{\preccurlyeq}_{E,gl} K_2 \triangleq gl\{K_1, E\} \preccurlyeq gl\{K_2, E\}$$

The examples in Figures 2.5 and 2.6 are both counterexamples to the circular rule, that is: $K \sqsubseteq^{\preccurlyeq}_{A,gl} G$ and $E \sqsubseteq^{\preccurlyeq}_{G,gl} A$ but $K \not\sqsubseteq^{\preccurlyeq}_{E,gl} G$. Figure 2.5 shows that non-determinism of the abstract environment is a problem. In Figure 2.6, both the assumption A and the guarantee G forbid b to occur. This allows their respective refinements according to $\sqsubseteq^{\preccurlyeq}$, E and K, to offer b — since they can rely on G respectively A to forbid its actual occurrence. But obviously, the composition of the implementations $gl\{E, K\}$ now allows b.

Because circular reasoning is not sound for all refinements under context, it may be useful to use a stronger (more restrictive) definition of refinement under context in order to make circular reasoning sound. This will be illustrated later, especially in Chapter 6 which is related to the SPEEDS project.

2.3.3 A sufficient condition for dominance

We provide now a sufficient condition for dominance when circular reasoning is sound. It relies on the fact that local assumptions are indeed discharged, that is, implied by the environment defined by the guarantees of the peers and the global assumption A. Notations are those introduced in Definition 2.2.13.

Theorem 2.3.5 *If* $\forall i,\ \exists gl_{E_i},\ gl \circ gl_I = gl_i \circ gl_{E_i}$ *and circular reasoning is sound, then to prove that* $\{\mathcal{C}_i\}_{i=1}^n$ *dominates* $\mathcal{C}$ *w.r.t.* gl_I*, it is sufficient to prove that:*

$$\begin{cases} gl_I\{G_1, \ldots, G_n\} \models \mathcal{C} \\ \forall i, gl_{E_i}\{A, G_1, \ldots, G_{i-1}, G_{i+1}, \ldots, G_n\} \models \mathcal{C}_i^{-1} \end{cases}$$

This condition shows that the proof of a dominance relation boils down to a set of satisfaction checks, one for proving refinement between the guarantees, the second for discharging individual assumptions. This result is particularly significant because one can check dominance without having to compose neither implementations nor contracts.

Note that there is no mention of how to find the set of glues $\{gl_{E_i}\}_{i=1}^n$ which are required for the property to hold. As will be shown in the actual frameworks which are presented in the next chapters, this is not always trivial.

2.4 Verifying systems of arbitrary size

We formalize here how apply our design and verification methodology can be generalized to recursively defined systems.

2.4.1 Formal methodology

Consider a component grammar consisting of:

- a set of terminal symbols $\{A, I_1, \ldots, I_k\}$ representing implementations;
- a set of nonterminal symbols $\{S, K_0, K_1, \ldots, K_n\}$ representing hierarchical components; S, which defines the top-level closed system, is the axiom;
- a set of rules corresponding to design steps which define each non-terminal either as a composition of subsystems or as an implementation:
  - $S \longrightarrow gl\{A, K_0\}$.

– For $i \in [0, n]$, at least one rule either of the form $K_i \longrightarrow I_j$ (where $j \in [1, k]$) or of the form $K_i \longrightarrow gl_{\Sigma_i}\{K_j\}_{j\in\Sigma_i}$, where Σ_i is a set of indices in $[0, n]$ and gl_{Σ_i} is a glue on the union of the interfaces of K_j.

Rewrite rules express how a hierarchical component is either associated with an implementation or decomposed into a set of subcomponents. In particular, K_0 represents the system under design and A is a property of its real environment. Thus, S stands for the system *along with* its environment.

Given such a grammar, we provide a methodology for both design and verification of systems which can be represented as words accepted by this grammar. This approach is based on the four steps presented in Figure 2.1, namely conformance, decomposition, dominance and satisfaction. We choose again a top-down presentation, but one can proceed in a different order.

1. formulate a global requirement φ characterizing the closed system S made of K_0 and its environment E, define a contract $\mathcal{C} = (A, gl, G)$ associated with K_0 and prove that $gl\{A, G\} \preccurlyeq \varphi$
2. define for every non terminal K_i a contract $\mathcal{C}_{K_i} = (A_{K_i}, gl_{K_i}, G_{K_i})$ such that for every rule $K_l \longrightarrow gl_{\Sigma_l}\{K_j\}_{j\in\Sigma_l}$ having an occurrence of K_i on the right-hand side, there exists gl_{E_i} such that $gl_{K_l} \circ gl_{\Sigma_l} = gl_{K_i} \circ gl_{E_i}$
3. for each $K_i \longrightarrow gl_{\Sigma_i}\{K_j\}_{j\in\Sigma_i}$, show that $\{\mathcal{C}_{K_j}\}_{j\in\Sigma_i}$ dominates $\mathcal{C}_{K_i}$ w.r.t gl_{Σ_i}
4. prove that implementations satisfy their contract: $K_i \longrightarrow I_j \implies I_j \models \mathcal{C}_{K_i}$

Theorem 2.4.1 *Let $\mathcal{G}$ be a grammar such that all methodology steps have been completed to guarantee a requirement φ. Any component system corresponding to a word accepted by $\mathcal{G}$ satisfies φ.*

Proof. By a simple induction on the number of steps required for deriving the accepted word from S, we can prove that the system represented by K_0 satisfies its contract (A, gl, G), that is, $K_0 \sqsubseteq_{A,gl} G$. This implies, as one of the coherence conditions of our contract framework, that $gl\{A, K_0\} \preccurlyeq gl\{A, G\}$. As conformance is transitive, we have $gl\{A, K_0\} \preccurlyeq \varphi$. □

Corollary 2.4.2 *If circular reasoning is sound, we can modify the methodology as follows to be able to refine also the environment of the system: (1) E appears instead of A as a terminal symbol; (2) the initial rewrite rule $S \longrightarrow gl\{A, K_0\}$ is replaced with $S \longrightarrow gl\{E, K_0\}$; (3) Step 4. is enriched with a proof that the actual environment E satisfies the "mirror" contract of $\mathcal{C}$, that is, $E \models (G, gl, A)$.*

Proof. As $K_0 \sqsubseteq_{A,gl} G$ (by induction as before) and $E \sqsubseteq_{G,gl} A$, we obtain by circular reasoning that $K_0 \sqsubseteq_{E,gl} G$, which implies that $gl\{E, K_0\} \preccurlyeq gl\{E, G\}$. Besides, $E \sqsubseteq_{G,gl} A$ implies that $gl\{E, G\} \preccurlyeq gl\{A, G\}$. Thus, we obtain, again by transitivity, that $gl\{E, K_0\} \preccurlyeq \varphi$. □

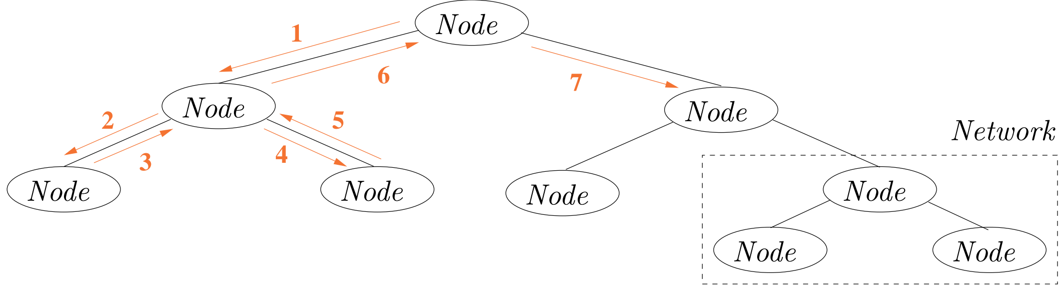


Figure 2.7 – The overall structure of the application.

2.4.2 An application to resource sharing in a network

In [BHQG10c], which is not presented in this thesis, we have applied this methodology to a non-trivial algorithm [DDHL09] for sharing resources by means of a token ring in networks structured as binary trees. Figure 2.7 illustrates the structure of such networks and the colored arrows show in which order tokens are transmitted across the system. The top-level requirement φ has both a safety and a progress part. Thus, the contract framework that is defined is quite complex as it has to be expressive enough to encompass safety and progress while handling data transfer. In this context, not having to prove correctness of the proof rules in the concrete setting is very helpful.

Networks are defined according to the following grammar $\mathcal{G}$, where $\{E, I_{\text{Node}}\}$ are terminals and $\{Sys, Net, Node\}$ nonterminals with axiom Sys. The rules are:

$$\begin{aligned}
Sys &\longrightarrow gl_{\text{Net}}\{E, Net\} \\
Net &\longrightarrow Node \\
Net &\longrightarrow gl\{Node, Net, Net\} \\
Node &\longrightarrow I_{\text{Node}}
\end{aligned}$$

Our goal is to prove that every system built according to $\mathcal{G}$, thus consisting of a network together with an environment E that gives back tokens and privilege immediately, conforms to φ. This is achieved by following the four verification steps discussed before. More precisely:

1. We formulate a contract $\mathcal{C}_{\text{Net}} = (A_{\text{Net}}, gl_{\text{Net}}, G_{\text{Net}})$ associated with Net (which plays here the role of K_0) and we prove that $gl_{\text{Net}}\{A_{\text{Net}}, G_{\text{Net}}\}$ conforms to φ.
2. We define $\mathcal{C}_{\text{Node}} = (A_{\text{Node}}, gl_{\text{Node}}, G_{\text{Node}})$ associated with $Node$ (note that we already have a contract for Net). We do not explain here how the gl_{E_i} are computed.
3. We show that $\mathcal{C}_{\text{Node}}$ dominates $\mathcal{C}_{\text{Net}}$ and that $\{\mathcal{C}_{\text{Node}}, \mathcal{C}_{\text{Net}}, \mathcal{C}_{\text{Net}}\}$ dominates $\mathcal{C}_{\text{Net}}$ w.r.t. gl.

4. We prove that I_{Node} satisfies $\mathcal{C}_{\mathrm{Node}}$ and that E satisfies $\mathcal{C}_{\mathrm{Net}}^{-1} = (G_{\mathrm{Net}}, gl_{\mathrm{Net}}, A_{\mathrm{Net}})$.

Note that this proof implies that *all* networks that can be built according to this grammar satisfy φ. In [BHQG10c], we use the sufficient condition provided by Theorem 2.3.5 for proving dominance. Then, a prototype implemented in Java discharges automatically the single conformance check and all the satisfaction checks — i.e., those relating implementations to contracts and those resulting from the dominance problem.

2.5 Proofs

Theorem 2.3.5 *If* $\forall i,\ \exists gl_{E_i},\ gl \circ gl_I = gl_i \circ gl_{E_i}$ *and circular reasoning is sound, then to prove that* $\{\mathcal{C}_i\}_{i=1}^{n}$ *dominates* $\mathcal{C}$ *w.r.t.* gl_I*, it is sufficient to prove that:*

$$\begin{cases} gl_I\{G_1,\dots,G_n\} \models \mathcal{C} \\ \forall i,\ gl_{E_i}\{A, G_1,\dots,G_{i-1},G_{i+1},\dots,G_n\} \models \mathcal{C}_i^{-1} \end{cases}$$

Proof. For every $i \in [1,n]$, let K_i be a component on $\mathcal{P}_i$. Suppose the following:

1. $\forall i,\ \exists gl_{E_i},\ gl \circ gl_I = gl_i \circ gl_{E_i}$
2. $gl_I\{G_1,\dots,G_n\} \sqsubseteq_{A,gl} G$
3. $\forall i,\ gl_{E_i}\{A, G_1,\dots,G_{i-1},G_{i+1},\dots,G_n\} \sqsubseteq_{G_i,gl_i} A_i$
4. $\forall i,\ K_i \sqsubseteq_{A_i,gl_i} G_i$

We aim at proving $gl_I\{K_1,\dots,K_n\} \models \mathcal{C}$, that is: $gl_I\{K_1,\dots,K_n\} \sqsubseteq_{A,gl} G$. For this, we show by induction that for any l in $[0,n]$, for any partition $\{J,K\}$ of $[1,n]$ such that $|J| = l$:

$$\begin{cases} gl_I\{\mathcal{K}^J \cup \mathcal{G}^K\} \sqsubseteq_{A,gl} G \\ \forall i \in K,\ gl_{E_i}\{A, \mathcal{E}_i^{J,K}\} \sqsubseteq_{G_i,gl_i} A_i \end{cases}$$

with $\mathcal{K}^J = \{K_j\}_{j\in J}$, $\mathcal{G}^K = \{G_k\}_{k\in K}$ and with $\mathcal{E}_i^{J,K} = \mathcal{K}^J \cup (\mathcal{G}^K \backslash \{G_i\})$.

- $l = 0$. By (2) and (3) the property holds.
- $0 \le l < n$. We suppose that our property holds for l. Let $\{J', K'\}$ be a partition of $[1,n]$ such that $|J'| = l+1$. Let q be an element of J'. We fix $J = J' \backslash \{q\}$ and $K = K' \cup \{q\}$.

Step 1 We first prove that $gl_I\{\mathcal{K}^{J'} \cup \mathcal{G}^{K'}\} \sqsubseteq_{A,gl} G$.

$$\begin{cases} K_q \sqsubseteq_{A_q,gl_q} G_q \text{ from (4)} \\ gl_{E_q}\{A,\ \mathcal{E}_q^{J,K}\} \sqsubseteq_{G_q,gl_q} A_q \end{cases}$$

The second property is our recurrence hypothesis, as $q \in K$. Thus, by circular reasoning (**CR**):

$$K_q \sqsubseteq_{gl_{E_q}\{A,\mathcal{E}_q^{J,K}\},gl_q} G_q$$

As refinement under context is preserved by composition, we obtain by (**CMP**):

$$gl_I\{K_q, \mathcal{E}_q^{J,K}\} \sqsubseteq_{A,gl} gl_I\{G_q, \mathcal{E}_q^{J,K}\}$$

This is equivalent to $gl_I\{\mathcal{K}^{J'} \cup \mathcal{G}^{K'}\} \sqsubseteq_{A,gl} gl_I\{\mathcal{K}^J \cup \mathcal{G}^K\}$.
Finally, by using the recurrence hypothesis: $gl_I\{\mathcal{K}^{J'} \cup \mathcal{G}^{K'}\} \sqsubseteq_{A,gl} G$.

Step 2 We now have to prove that:

$$\forall i \in K',\ gl_{E_i}\{A, \mathcal{E}_i^{J',K'}\} \sqsubseteq_{G_i,gl_i} A_i$$

We fix $i \in K'$. We have proved in step 1 that:

$$K_q \sqsubseteq_{gl_{E_q}\{A,\mathcal{E}_q^{J,K}\},gl} G_q$$

$K = K' \cup \{q\}$, so $i \in K$. Thus, by compositionality (**CMP**), we obtain:

$$gl_{E_i}\{K_q, A, \mathcal{E}_q^{J,K\setminus\{i\}}\} \sqsubseteq_{G_i,gl_i} gl_{E_i}\{G_q, A, \mathcal{E}_q^{J,K\setminus\{i\}}\}$$

This boils down to $gl_{E_i}\{A, \mathcal{E}_i^{J',K'}\} \sqsubseteq_{G_i,gl_i} gl_{E_i}\{A, \mathcal{E}_i^{J,K}\}$.
Hence, using the recurrence hypothesis: $gl_{E_i}\{A, \mathcal{E}_i^{J',K'}\} \sqsubseteq_{G_i,gl_i} A_i$.

Conclusion By applying our property to $l = n$, we get:

$$gl_I\{K_1, \ldots, K_n\} \sqsubseteq_{A,gl} G$$

□

Chapter 3

Beyond the definitions

In this chapter, we put our methodology into perspective. In Section 3.1, we discuss features which may seem to be missing from our contract frameworks. In fact, some of them can be expressed using our definitions while others require only slight changes in the theory. Then, Section 3.2 recalls notions which are at the core of most interface theories: consistency and compatibility. Since our definition of component framework is generic enough to encompass frameworks with complex interaction, we introduce a structural counterpart for these two notions. Then, we justify our choice not to require an algebra of contracts. Finally, in Section 3.3, we propose a novel mechanism for establishing dominance when circular reasoning is not sound, which is based on the combined use of two refinement under context relations. A somewhat surprising corollary is that this mechanism can also be used to relax circular reasoning when it is unnecessarily strong.

3.1 Possible extensions

The definition of contract framework that we have introduced is kept as simple as possible in order to encompass a large variety of frameworks. As a result, it may seem too limited for some specific frameworks. In particular, we now discuss four features which are not explicitly mentioned in our approach and which are useful in some contexts:

– *structuring* allows recomposing a hierarchical component according to any partition of its subcomponents
– *projection* permits to project the behavior of a component onto a subset of its ports
– *equivalence of glues* is used to reason not only about systems with exactly the same glues

– *defining glues on a set of interfaces* rather than a set of ports allows enforcing stronger conditions on glues.
– *well-formedness* deals with the preservation of properties which are not directly expressible in the formalism used to represent components.

For each of these extensions, we discuss whether and how our definitions can be adapted.

3.1.1 Structuring

In a component framework, two types of equivalence-preserving transformations of components are of particular interest. The first one transforms any hierarchical component into a "flat" one, i.e., a component consisting of a unique glue composing all its atomic (non-hierarchical) subcomponents. This is achieved by the composition operator $\circ$ on glues which ensures by definition that $gl\{gl'\{\mathcal{K}^1\}, \mathcal{K}^2\}$ is equivalent to $(gl \circ gl')\{\mathcal{K}^1 \cup \mathcal{K}^2\}$ for any sets of components $\mathcal{K}^i$ such that all terms are defined.

Symmetrically, it is also interesting to be able to *structure* a composition according to a given partition of the subcomponents. In particular, this allows focusing on a subsystem, by representing how this subsystem is connected to its environment, that is, the rest of the system.

Definition 3.1.1 (Structuring) *A component framework allows* structuring *if for any component $K = gl\{\mathcal{K}^1 \cup \mathcal{K}^2\}$, it is possible to find glues gl_1 and gl_2 such that $gl_2\{gl_1\{\mathcal{K}^1\}, \mathcal{K}^2\}$ is defined and equivalent to K.*

Example 3.1.2 *Consider a component $K = gl\{K_1, \ldots, K_5\}$. As illustrated in Figure 3.1, we can transform K into the equivalent component $gl_{1,2}\{gl_1\{K_1, K_2\}, gl_2\{K_3, K_4, K_5\}\}$. This is done by applying structuring twice: first, with the partition $\mathcal{K}^1 = \{K_1, K_2\}$ and $\mathcal{K}^2 = \{K_3, K_4, K_5\}$ to build $gl_1'\{gl_1\{K_1, K_2\}, K_3, K_4, K_5\}$; then, with $\mathcal{K}^1 = \{K_3, K_4, K_5\}$ and $\mathcal{K}^2 = \{gl_1\{K_1, K_2\}\}$ in order to obtain the result.*

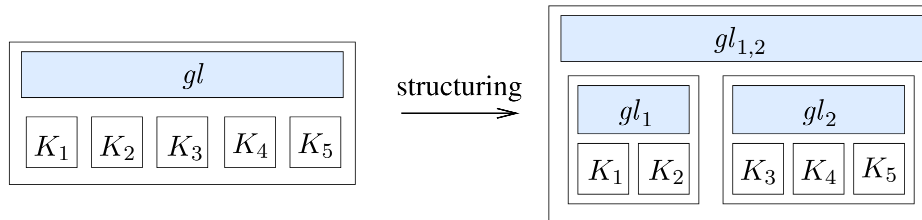


Figure 3.1 – Structuring a system to find gl_1, gl_2 and $gl_{1,2}$ such that $gl = gl_{1,2} \circ gl_1 \circ gl_2$.

Note that by flattening again the system, we obtain the following:

$$\begin{aligned} gl_{1,2}\{gl_1\{K_1, K_2\}, gl_2\{K_3, K_4, K_5\}\} &\cong (gl_{1,2} \circ gl_1)\{K_1, K_2, gl_2\{K_3, K_4, K_5\}\} \\ &= (gl_{1,2} \circ gl_1)\{gl_2\{K_3, K_4, K_5\}, K_1, K_2\} \\ &\cong (gl_{1,2} \circ gl_1 \circ gl_2)\{K_3, K_4, K_5, K_1, K_2\} \\ &= (gl_{1,2} \circ gl_1 \circ gl_2)\{K_1, K_2, K_3, K_4, K_5\} \end{aligned}$$

Remember that $\circ$ *is left-associative.*

Structuring is a key property because it lifts the conditions imposed on glues in the compositionality rule (**CMP**): given a system $K = gl\{S, gl_E\{E_1, E_2\}\}$, it is always possible to find gl_1 and gl_2 such that $K \cong gl_2\{gl_1\{S, E_1\}, E_2\}\}$ by flattening and then structuring.

Frameworks allowing data transfer do not always allow structuring: consider a framework as in the variant of BIP presented in [BJS09], where connectors are allowed to perform some operations on data, e.g., computing the maximum value of the variables associated to the ports involved in the interaction, and then setting all the variables to this value. Then, although flattening is still possible, structuring is not in general, intuitively because composition of function is always defined, but it is not always possible to decompose a function of three arguments into two functions of 2 arguments.

The condition related to glues in the sufficient condition for dominance 2.3.5 is different from structuring. In this case, two glues are given, corresponding to gl and gl_2 in the definition of structuring, and the problem is to find — if it exists — gl_1 such that $gl = gl_2 \circ gl_1$.

Definition 3.1.3 (Compatibility of glues) *Consider two glues gl and gl_2 with the same exported interface. We say that gl and gl_2 are* compatible *if there exists gl_1 such that $gl = gl_1 \circ gl_2$.*

In the context of the dominance problem, the goal is to relate the glue gl_i provided in each $\mathcal{C}_i$ to the actual environment of component K_i, as illustrated in Figure 3.2: gl is the glue defined in the top-level contract and gl_I defines how subcomponents are composed. Thus, the actual environment of subcomponent K_1 consists of components K_2 to K_4 and A, the top-level assumption. However, the glue gl_1 provided in the contract for K_1 has been defined for an abstract environment A_1, hence the need for a glue gl_{E_1} representing the environment of K_1 as a single component with the same interface as A_1.

We call *compatibility inference* the ability of a component framework to determine for any two glues whether they are compatible, furthermore providing the glue that makes them compatible if it is the case — gl_1 in the definition of compatibility, gl_{E_1} in the dominance problem. The BIP

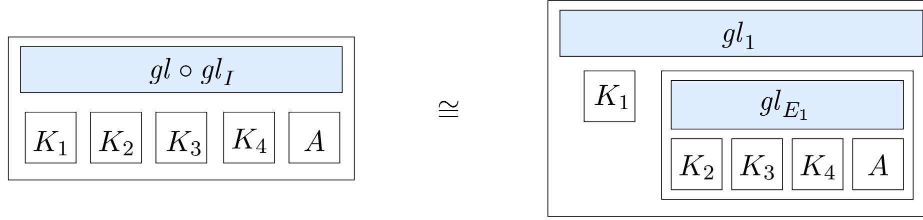


Figure 3.2 – gl_{E_1} allows relating the glue gl_1 provided in $\mathcal{C}_1$ to the actual environment for K_1.

semantic and the I/O component frameworks presented in Chapter 4 and 6 both allow structuring and compatibility inference, but the two variants of BIP presented in Chapter 5 do not. Intuitively, this is because the exported port of a connector may have an arbitrary name, and so if a port of K_1 is connected to two ports of A_1 in the same way (the allowed interactions are the same), then it is not possible to determine automatically which actual port should be mapped onto which abstract port.

3.1.2 Projection

The basic idea of incremental design is that details are abstracted away from low-level implementations to top-level specifications. This requires hiding ports of the lower-level contracts which do not appear at the interface of the top-level contract. For this, we need a notion of projection of the behavior of a component onto a subset of its interface. A simple and elegant way of doing this is to represent such a projection as a glue.

As shown in Figure 3.3, the projection Π of an interface $\mathcal{P}$ onto a subset of its ports $\mathcal{P}' \subseteq \mathcal{P}$ is a glue with support set $S_\Pi = \mathcal{P}$ and exported interface $\mathcal{P}_\Pi = \mathcal{P}'$. K is represented as a composition because this will typically be the case. Furthermore, it must be consistent with the given notions of equivalence of components $\cong$, refinement under context $\{\sqsubseteq_\omega\}_{\omega \in \Omega}$ and conformance $\preccurlyeq$. As there are already coherence conditions between those relations, we only need the following additional condition, as illustrated in Figure 3.3:

$$\textit{if } K \sqsubseteq_{A, gl \circ \Pi} G\textit{, then } \Pi\{K\} \sqsubseteq_{A, gl} \Pi\{G\}$$

Note that in terms of satisfaction of contracts, this condition is equivalent to:

$$\textit{if } K \models (A, gl \circ \Pi, G)\textit{, then } \Pi\{K\} \models (A, gl, \Pi\{G\})$$

In other words, $\{(A, gl \circ \Pi, G)\}$ dominates $(A, gl, \Pi\{G\})$ with respect to Π — this is the definition

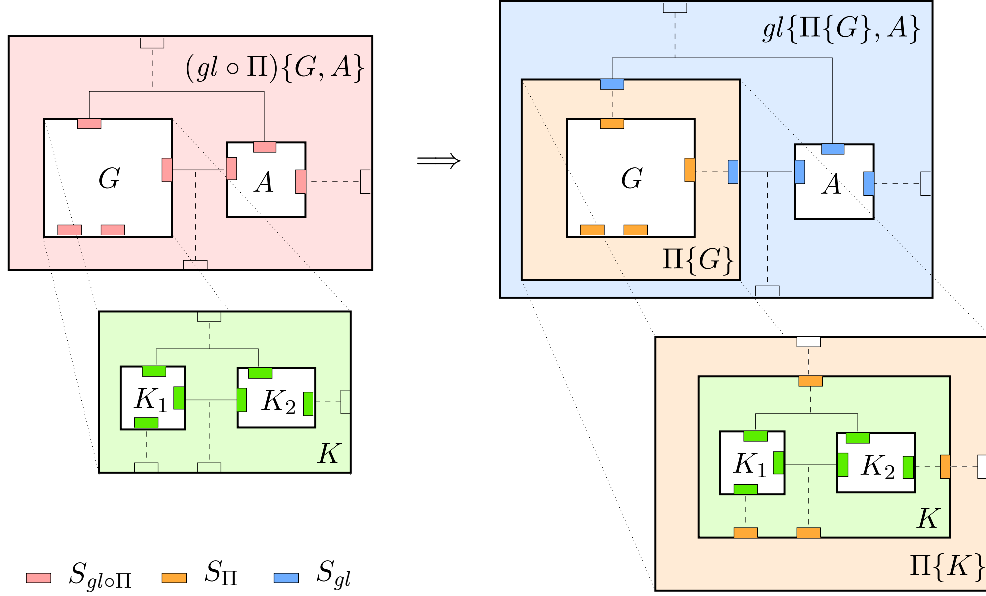


Figure 3.3 – Property of a projection: $K \sqsubseteq_{A,gl\circ\Pi} G \implies \Pi\{K\} \sqsubseteq_{A,gl} \Pi\{G\}$.

of dominance for a composition with a single subcomponent. Thus, contracts for different interfaces are related: one can use $(A, gl, \Pi\{G\})$ instead of $(A, gl \circ \Pi, G)$ at a higher level of hierarchy.

Example 3.1.4 *In a semantic framework where components are LTS, the usual projection for LTS can be represented as a set of glues as described here. Formally, given an LTS K on an interface $\mathcal{P}$ and a subset $\mathcal{P}' \subseteq \mathcal{P}$, the projection of K onto $\mathcal{P}'$ is the LTS $K' = (Q, q^0, \mathcal{P}', \longrightarrow')$, where $q_1 \xrightarrow{\alpha'}' q_2$ if and only if $q_1 \xrightarrow{\alpha} q_2$ for some $\alpha \subseteq \mathcal{P}$ such that $\alpha' = \alpha \cap \mathcal{P}'$. As by convention $\emptyset$ is a label of all our LTS, we do not need to introduce τ-transitions.*

3.1.3 Equivalence of glues

In frameworks where glues are rich, it may be of interest to consider an equivalence relation on glues. A typical case is that of BIP interaction models, which may have differently structured connectors, but still correspond to the same set of interactions. It would seem natural to define such relation with respect to equivalence of components, as in the following:

> Two glues gl and gl' with the same support set and exported interface are equivalent if and only if $gl\{K_1, \ldots, K_n\} \cong gl'\{K_1, \ldots, K_n\}$ for any set of components such that all terms are defined.

For example, consider interaction models as they are presented in Definition 5.2.2 of the presentation of the BIP framework, and equivalence of components as syntactic equality of their compositional semantics. Then, two interaction models Γ and Γ' are equivalent according to the proposed definition if they have the same set of blackbox multi-shot interactions, i.e. $\mathcal{M}_{bb}(\Gamma) = \mathcal{M}_{bb}(\Gamma')$ and furthermore every such interaction m is associated with the same set of white-box multi-shot interactions $wb(m)$ — this comes directly from the definition of compositional semantics.

However, this condition for equivalence of glues is unnecessarily strong. Indeed, $\cong$ is typically a relation that is preserved by composition, whereas equivalence of glues is needed only for closed systems. In our example, for example, one would like to define equivalence of interaction models as equality of set of interactions $\mathcal{I}(\Gamma) = \mathcal{I}(\Gamma')$, which preserves equivalence with respect to the closed semantics of BIP rather than the compositional one. For this reason, we do not provide a definition for equivalence of glues, but only focus on how to work with such a relation within our contract framework: if the following variant of the compositionality rule holds, where equivalence of glues is denoted $\sim$, then it is possible to relax the sufficient condition for dominance 2.3.5.

$$\frac{I \sqsubseteq_{E,gl} S \qquad E = gl_E\{E_1, E_2\} \qquad gl \circ gl_E \sim gl_2 \circ gl_1}{gl_1\{I, E_1\} \sqsubseteq_{E_2,gl_2} gl_1\{S, E_1\}} \quad (\mathbf{CMP}_\sim)$$

If ($\mathbf{CMP}_\sim$) holds, then instead of requiring that for every contract $\mathcal{C}_i$ there exists a glue gl_{E_i} such that $gl \circ gl_I = gl_i \circ gl_{E_i}$, it is sufficient to require that $gl \circ gl_I \sim gl_i \circ gl_{E_i}$. The reason for this is that gl_{E_i} is used in the proof only for applying compositionality. Note also that equivalence of glues can be combined with structuring and compatibility inference.

3.1.4 Defining glues on a partition

In our definitions, a glue is defined on a support set, independently of how this support set is partitioned into interfaces. This is the case in the frameworks that we present in the following chapters, and this simplifies notations. However, some frameworks require that the subset be given as a partition. The first variant of the BIP framework that was presented in Chapter 1 is an example: indeed, ports of a same component may not be connected.

The definition of glue can be then extended: a glue is associated with a set of disjoint interfaces S_{gl} (rather than a set of ports) and a composition $K = gl\{K_1, \ldots, K_n\}$ is defined if $S_{gl} = \{\mathcal{P}_{K_i}\}_{i=1}^n$ (instead of $S_{gl} = \bigcup_{i=1}^n \mathcal{P}_{K_i}$). Besides, $gl \circ gl'$ is defined if $\mathcal{P}_{gl'} \in S_{gl}$ (and not $\mathcal{P}_{gl'} \subseteq S_{gl}$). Then, its support set is $S_{gl} \backslash \{\mathcal{P}_{gl'}\} \cup S_{gl'}$ (with respect to $S_{gl} \backslash \mathcal{P}_{gl'} \cup S_{gl'}$). In the sequel, every glue is defined on a union of interfaces, which we now replace by a set of interfaces.

3.1.5 Well-formedness

In this section, we justify informally the need for a notion of *well-formedness* of systems to deal with a property that cannot be expressed in the considered component framework. Consider for example a framework where components are represented by modal transition systems — one such framework is presented in Chapter 7. It is possible to prove that an MTS is deadlock-free, but it is not possible to express deadlock-freedom as an MTS. Another scenario is as follows: under some unexpected input, a reactive component may fail to produce an output because its internal computation does not terminate. Here, it is probably not even possible to prove termination of a given specification.

Let us focus first on the deadlock-freedom example. We define conformance as refinement of MTS, that is: K_1 refines K_2 if and only if all *must*-transitions are preserved from K_2 to K_1 and all *may*-transitions of K_1 are also allowed in K_2. Intuitively, deadlock-freedom is preserved by conformance because it is derived from *must*-transitions, which are preserved by refinement. As a consequence, if one proves that $gl\{A, G\}$ is deadlock-free, then for any implementation K satisfying $\mathcal{C} = (A, gl, G)$, the system $gl\{A, K\}$ is also deadlock-free — remember that $K \sqsubseteq_{A,gl} G$ implies $gl\{A, K\} \preccurlyeq gl\{A, G\}$. Thus, deadlock-freedom has to be proven only once.

However, a top-level description is typically not fine-grained enough to guarantee deadlock-freedom. Thus, we are also interested in preserving well-formedness in a bottom-up fashion.

This may be achieved by strengthening the definition of dominance — stating that a contract $\mathcal{C} = (A, gl, G)$ dominates $\mathcal{C}' = (A', gl, G')$ if any implementation of $\mathcal{C}$ is also an implementation of $\mathcal{C}'$ — by adding the following constraint:

$$\textit{for any } E \textit{ on } \mathcal{P}_A\textit{, if } E \models (G', gl, A') \textit{ then } E \models (G, gl, A)$$

That is, $\mathcal{C}$ dominates $\mathcal{C}'$ if and only if any implementation of $\mathcal{C}$ is an implementation of $\mathcal{C}'$ *and* any implementation of $\mathcal{C}'^{-1}$ is an implementation of $\mathcal{C}^{-1}$. This ensures, provided circular reasoning is sound, that the assumption of the low-level contract is indeed satisfied in the actual environment of the component.

Lemma 3.1.5 *Consider a component K on an interface $\mathcal{P}$, and two contracts $\mathcal{C}_c$ and $\mathcal{C}_{abs}$ for $\mathcal{P}$ with $gl_c = gl_{abs}$ that we denote gl. Suppose that $K \models \mathcal{C}_c$ while $\mathcal{C}_c$ dominates $\mathcal{C}_{abs}$ and $E \models \mathcal{C}_{abs}^{-1}$. If circular reasoning is sound, then $gl\{E, K\} \preccurlyeq gl\{A_c, G_c\}$.*

Proof. We have:

1. $K \sqsubseteq_{A_c,gl} G_c$

2. $\mathcal{C}_c$ dominates $\mathcal{C}_{abs}$
3. $E \sqsubseteq_{G_{abs},gl} A_{abs}$

From (2) and (3), we get, by our new definition of dominance, that:

4. $E \sqsubseteq_{G_c,gl} A_c$

As a result, using (**CR**) on (1) and (4), we have:

5. $E \sqsubseteq_{K,gl} A_c$

This in turn implies that:

6. $gl\{E, K\} \preccurlyeq gl\{A_c, K\}$

Besides, (1) implies that:

7. $gl\{A_c, K\} \preccurlyeq gl\{A_c, G_c\}$

Finally, by transitivity, we get from (6) and (7) the result, that is:

8. $gl\{E, K\} \preccurlyeq gl\{A_c, G_c\}$

□

This property means that it is sufficient to prove well-formedness on *any* contract in the design process — not only on the top-level contract — to ensure that the concrete system is also well-formed. Note that this well-formedness condition is obviously preserved by our sufficient condition for dominance, which is based on requiring that all assumptions are verified.

If we look back at our example of an unexpected input leading to a component never producing an output, we find that the stronger definition of dominance defined here is also helpful in this case. Indeed, Step 5. in the previous proof ensures that the concrete assumption made by the component about its inputs is indeed satisfied. This allows using other tools, or even an informal argument to prove well-formedness (termination in this example).

3.2 Additional notions

In this section, we focus on notions which usually appear in existing interface theories but are not present in our definition of framework, for example consistency and compatibility, In particular, we present our reasons for not using an algebra of contracts.

3.2.1 Consistency

A notion of consistency is used in [BCP07, BCF+07, BFM+08] in order to check that a contract can actually be satisfied. This is needed because assumptions and guarantees are sets of behaviors and

may thus arbitrarily constrain ports which are controlled by the environment or by the component. Informally, consistency is defined as follows:

A contract is consistent *if there exists at least one implementation satisfying it.*

In the context of this thesis, as refinement under context is a preorder, there does always exist a component satisfying a contract (A, gl, G), namely G. Intuitively, the difference between both approaches is that we consider guarantees as the expression of what the component should offer and not of how it should actually behave.

Hence consistency is not relevant here. However, if contracts are used in top-down design, refining a component into several subcomponents may lead to inconsistency in the sense that G may not be obtainable as a composition using a given glue. This is what we call *structural consistency*.

Definition 3.2.1 (Structural consistency) *Let $\mathcal{C} = (A, gl, G)$ be a contract and gl_I a glue such that $\mathcal{P}_{gl_I} = \mathcal{P}_G$ and $S_{gl_I} \cap \mathcal{P}_G = \emptyset$. Consider a partition $\{\mathcal{P}_i\}_{i=1}^n$ of S_{gl}. Then $\mathcal{C}$ is* consistent with respect to *gl_I and $\{\mathcal{P}_i\}_{i=1}^n$ if and only if there exist $K_1, \ldots, K_n$ on respectively $\mathcal{P}_1, \ldots, \mathcal{P}_n$ such that $gl_I\{K_1, \ldots, K_n\} \models \mathcal{C}$.*

Figure 3.4 presents an example based on MTS, which we describe informally here and in detail in Chapter 7. The contract $\mathcal{C} = (A, gl, G)$ represented in Figure 3.4 states that in an environment that may always offer a_2 and b_2, a component satisfying $\mathcal{C}$ must offer only a_1 in state q_0 and then it must offer b_1 and possibly a_1 in state q_1. However, this behavior cannot be obtained by a composition using gl_I. Indeed, firing a_1 modifies the conditions imposed on b_1, while these two interactions are part of two different components which cannot observe each other. Thus, b_1 must be offered and forbidden in states which are indistinguishable by the component in charge of it.

3.2.2 Compatibility

Compatibility is a notion that we have not studied in detail in this thesis. The reason is that it is useful in a bottom-up design approach, but no so much in a top-down approach as we propose. Informally, compatibility is expressed as follows:

Two components are compatible *if there is an environment where they can work together.*

In terms of contracts, two contracts (A_1, G_1) and (A_2, G_2) for disjoint interfaces are *compatible* if there exists at least one environment E such that $E \parallel G_1 \sqsubseteq A_2$ and $E \parallel G_2 \sqsubseteq A_1$ — that is, such that both assumptions can be discharged. Compatibility can be established by finding a winning strategy in a two-player game as described in [dAHS02, CdAHS03].

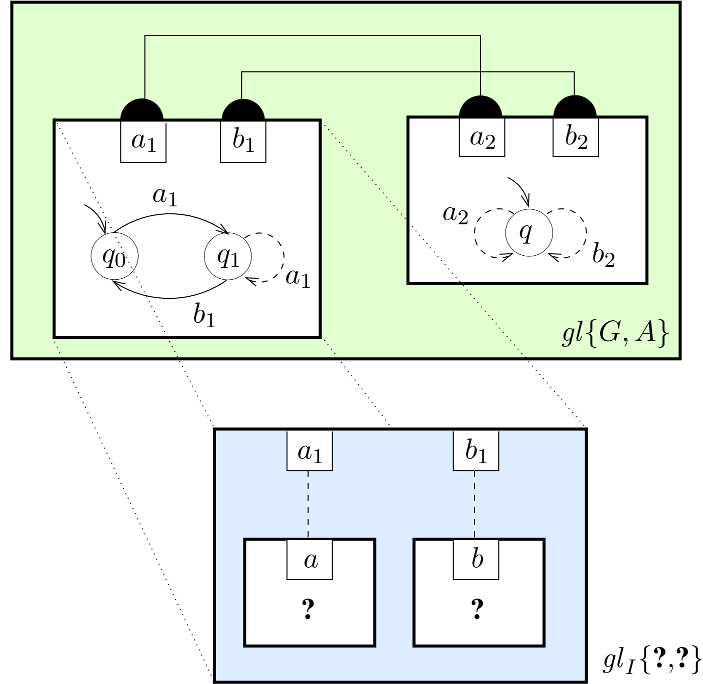


Figure 3.4 – An example of structural inconsistency.

In a context where glues are parametrized, compatibility requires the existence of a environment E and furthermore four different glues that relate the two contracts and E. This is illustrated on Figures 3.5 and 3.6: given two contracts $\mathcal{C}_1$, $\mathcal{C}_2$ and a glue gl defining how two components K_1, K_2 satisfying these contracts are intended to be composed, the goal is to find a context for the obtained component $gl\{K_1, K_2\}$ and two glues gl_{E_1}, gl_{E_2} that relate assumption A_1 (respectively A_2) to the actual environment of K_1 (resp. K_2), namely a composition of E and K_2 (resp. of E and K_1).

Definition 3.2.2 (Compatibility of contracts) *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be disjoint interfaces. Two contracts $\mathcal{C}_1 = (A_1, gl_1, G_1)$ and $\mathcal{C}_2 = (A_2, gl_2, G_2)$ for respectively $\mathcal{P}_1$ and $\mathcal{P}_2$ are* compatible *if and only if there exists a component E and two glues gl_{E_1} and gl_{E_2} such that $gl_{E_1}\{E, G_2\} \models \mathcal{C}_1^{-1}$ and $gl_{E_2}\{E, G_1\} \models \mathcal{C}_2^{-1}$, all terms are defined and furthermore: $gl_1 \circ gl_{E_1} = gl_2 \circ gl_{E_2}$.*

$\mathcal{C}_1$ and $\mathcal{C}_2$ are compatible with respect to *a glue gl on $\mathcal{P}_1 \cup \mathcal{P}_2$ if and only if they are compatible and there exists a glue gl_E such that $gl_1 \circ gl_{E_1} = gl_2 \circ gl_{E_2} = gl_E \circ gl$.*

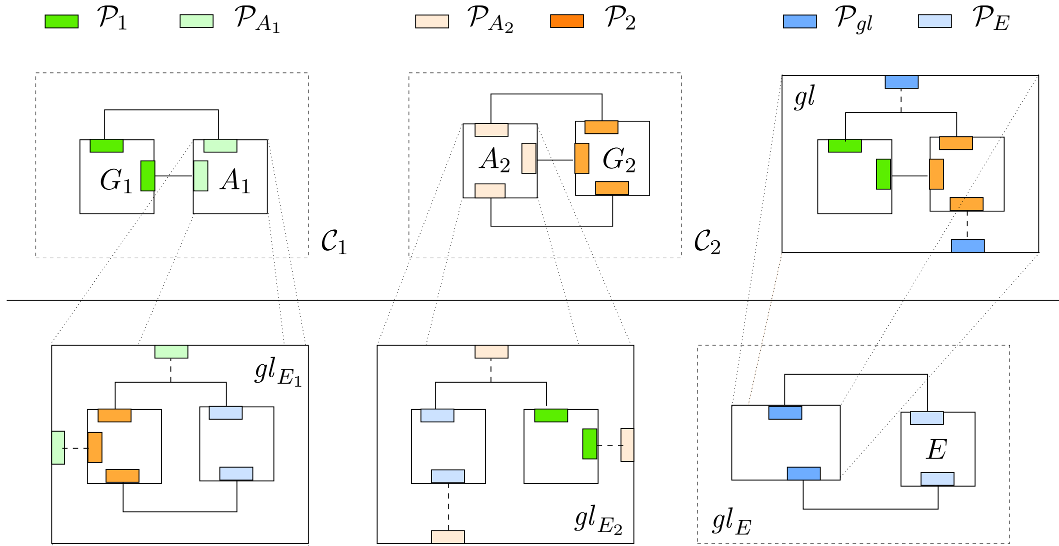


Figure 3.5 – Given $\mathcal{C}_1$, $\mathcal{C}_2$ and gl, does there exist appropriate gl_{E_1}, gl_{E_2} and (E, gl_E)?

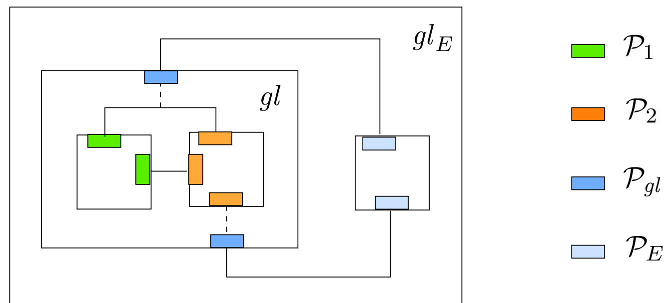


Figure 3.6 – Structure of the system under design

Property 3.2.3 *Consider two contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ for disjoint interfaces $\mathcal{P}_1$ and $\mathcal{P}_2$. If $\mathcal{C}_1$ and $\mathcal{C}_2$ are compatible w.r.t. gl, then $\{\mathcal{C}_1, \mathcal{C}_2\}$ dominates $\mathcal{C} = (E, gl_E, gl\{G_1, G_2\})$, where we use the notations of Definition 3.2.2.*

Proof. This property is a direct application of the sufficient condition for dominance 2.3.5. □

Compatibility focuses on assumptions being actually discharged. In this respect, it is related to well-formedness. In particular, it is preserved only by the strong definition of dominance provided in Section 3.1.5 — which we call *strong dominance* in the following property. We suppose here that circular reasoning is sound.

Property 3.2.4 *Consider two contracts $\mathcal{C}_1$ and $\mathcal{C}'_1$ for the same interface $\mathcal{P}_1$ and a contract $\mathcal{C}_2$ for an interface $\mathcal{P}_2$ disjoint from $\mathcal{P}_1$. If $\mathcal{C}_1$ and $\mathcal{C}_2$ are compatible and $\mathcal{C}'_1$ strongly dominates $\mathcal{C}_1$, then $\mathcal{C}'_1$ and $\mathcal{C}_2$ are compatible.*

Proof. Suppose that $\mathcal{C}_1 = (A_1, gl_1, G_1)$ and $\mathcal{C}_2 = (A_2, gl_2, G_2)$ are compatible and that $\mathcal{C}'_1 = (A'_1, gl_1, G'_1)$ strongly dominates $\mathcal{C}_1$. Then there exists E, gl_{E_1} and gl_{E_2} such that:

1. $gl_{E_1}\{E, G_2\} \sqsubseteq_{G_1,gl_1} A_1$
2. $gl_{E_2}\{E, G_1\} \sqsubseteq_{G_2,gl_2} A_2$
3. $G'_1 \sqsubseteq_{A_1,gl_1} G_1$
4. $A_1 \sqsubseteq_{G'_1,gl_1} A'_1$

We want to show that $gl_{E_1}\{E, G_2\} \sqsubseteq_{G'_1,gl_1} A'_1$ and $gl_{E_2}\{E, G'_1\} \sqsubseteq_{G_2,gl_2} A_2$.
By circular reasoning (**CR**) on 1. and 3. we obtain:

5. $gl_{E_1}\{E, G_2\} \sqsubseteq_{G'_1,gl_1} A_1$

We can then conclude by transitivity of $\sqsubseteq_{G'_1,gl}$ on 5. and 4. the first part of our result.
By applying (**CR**) to 3. and 1. we get:

6. $G'_1 \sqsubseteq_{gl_{E_1}\{E,G_2\},gl_1} G_1$

Using preservation of refinement by composition on 6., we obtain:

7. $gl_{E_2}\{E, G'_1\} \sqsubseteq_{G_2,gl_2} gl_{E_2}\{E, G_1\}$

Finally, by transitivity of $\sqsubseteq_{G_2,gl_2}$, we get the second part of our result from 7. and 2. □

Like consistency, compatibility has a structural counterpart related to the glues that must be found to establish compatibility.

Definition 3.2.5 (Structural compatibility) *Contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ for disjoint interfaces $\mathcal{P}_1$ and $\mathcal{P}_2$ are* structurally compatible *if and only if there exists gl_{E_1} and gl_{E_2} such that $\mathcal{P}_2 \subseteq S_{gl_{E_1}}$, $\mathcal{P}_1 \subseteq S_{gl_{E_2}}$ and $gl_1 \circ gl_{E_1} = gl_2 \circ gl_{E_2}$.*

Note that if gl_E and gl are provided, then establishing structural compatibility of contracts boils down to prove that $gl_E \circ gl$ and gl_1 are compatible according to Definition 3.1.3, and $gl_E \circ gl$ and gl_2 also.

Example 3.2.6 *In the HRC L0 framework presented in Chapter 6, assumptions and guarantees are expressed on the same set of ports. That is, glues in such contracts are implicitly defined as sets of binary rendezvous. However, glues in the general framework may contain n-ary rendezvous. Such a situation is illustrated in Figure 3.5, where hierarchy allows building a rendezvous between the*

three components. In this context, finding gl_{E_1} and gl_{E_2} is quite straightforward, intuitively because a rendezvous involving three participants a, b and c can be decomposed into two rendezvous: the first between a and b, the second between the group $\{a, b\}$ and c.

3.2.3 Composition of contracts

Some contract frameworks, such as the HRC L0 framework presented in Chapter 6, define an algebra on contracts. Having a contract algebra is nice because a dominance problem then boils down to checking that $\mathcal{C}_1 \parallel \mathcal{C}_2 \parallel \ldots \parallel \mathcal{C}_n$ dominates $\mathcal{C}$ for a given $\parallel$ defined on contracts. In particular, this implies that it is sufficient to define dominance as a binary relation.

In our context, where glues are parametrized, dominance would still have to be defined with respect to a glue gl, and we would need for each glue gl the existence of an operator $\widetilde{gl}$ on contracts such that:

$$\begin{cases} K_1 \models \mathcal{C}_1 \\ K_2 \models \mathcal{C}_2 \end{cases} \implies gl\{K_1, K_2\} \models \widetilde{gl}\{\mathcal{C}_1, \mathcal{C}_2\}$$

We do not propose such an algebra for two reasons: first, this in not possible in general (see Example 3.2.7), for example in frameworks which do not define a least upper bound of components, and we do not want to restrict ourselves to frameworks for which composition of contracts is defined. Rather, our aim is to propose a generic notion of contract framework which imposes minimal constraints on the component framework it is associated with. Second, composing contracts is not always desirable, as this may lead to state explosion in many concrete contract frameworks. Our goal is to show that it is possible to reason efficiently about contracts without ever composing them.

Example 3.2.7 *Let us sketch informally a simple example where two contracts cannot be composed. Consider three interfaces as in Figure 3.7. Suppose that we have two contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ for respectively $\mathcal{P}_1$ and $\mathcal{P}_2$ such that $gl_1 = gl_2$ consists of a single rendezvous connector between one port of each $\mathcal{P}_i$ and two ports of $\mathcal{P}_E$ (denoted p_E and p'_E). Suppose also that both assumptions A_1 and A_2 express that this rendezvous should not take place. The assumption of a contract $\mathcal{C} = \widetilde{gl}\{\mathcal{C}_1, \mathcal{C}_2\}$ would also be that the environment should prevent this rendezvous from taking place. However, two ports of $\mathcal{P}_E$ may be used for this purpose. If the component framework does not permit to offer either an interaction on p_E or on p'_E without offering both, then there does not exist a least restrictive global assumption.*

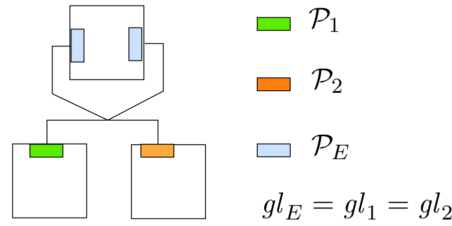


Figure 3.7 – Structure of a system for which contracts cannot be composed

3.2.4 Multiple contracts for components

In complex resource-constrained systems, a unique decomposition of the system under design is often not realistic. The approach developed here can handle independent development of multiple requirements. Let us consider attaching several contracts to the same component. It fact, this could lead to different interpretations. Indeed, a component K satisfying two contracts $\mathcal{C}_1 = (A_1, gl_1, G_1)$ and $\mathcal{C}_2 = (A_2, gl_2, G_2)$ could mean:

1. either that K is expected to be used in an environment satisfying *both* assumptions; $\mathcal{C}_1 = (A_1, gl_1, G_1)$ and $\mathcal{C}_2 = (A_2, gl_2, G_2)$ must be structurally compatible. This corresponds to using several contracts for the same component to describe different *viewpoints* of this component.

2. or K is expected to be used in an environment satisfying *at least one* of the assumptions. This corresponds to using several contracts for the same component to describe *use cases*.

If the first approach is taken, then all assumptions must be discharged (for well-formedness and for applying our sufficient condition for dominance). All guarantees may be used to discharge an assumption, but viewpoints can help choosing a subset of relevant guarantees. If the second approach is preferred, then at least one assumption per component must be discharged, and only the guarantees corresponding to discharged assumptions may be used. We have not addressed in this thesis the question of how to optimize the verification process to these methodologies involving multiple requirements.

3.3 Combining two refinement relations

We consider here using two relations of refinement under context instead of a single one. This idea comes from the comparison between the circular reasoning and assume-guarantee proof rules:

$$\frac{K \sqsubseteq_{A,gl} G \qquad E \sqsubseteq_{G,gl} A}{K \sqsubseteq_{E,gl} G}\ (\textbf{CR}) \qquad\qquad \frac{K \sqsubseteq_{A,gl} G \qquad E \sqsubseteq A}{K \sqsubseteq_{E,gl} G}\ (\textbf{AG})$$

As these rules are very similar, we wondered whether it was possible to unify them, and how useful this could be. The key to unification is to see refinement in any context as a very strong refinement under context — a relation so strong that if it holds, it actually holds for any context. This leads to the following rule, where *two* refinement relations, namely $\sqsubseteq^\alpha$ and $\sqsubseteq^\beta$, appear instead of just one:

$$\frac{K \sqsubseteq^\alpha_{A,gl} G \qquad E \sqsubseteq^\beta_{G,gl} A}{K \sqsubseteq^\alpha_{E,gl} G}\ (\textbf{PCR})$$

We call this rule pseudo-circular reasoning. Rule (**CR**) is obtained from (**PCR**) by taking $\sqsubseteq^\beta = \sqsubseteq^\alpha$. Rule (**AG**) is obtained by defining $\sqsubseteq^\beta$ as $\sqsubseteq^\alpha$ in any context, that is:

$$K_1 \sqsubseteq^\beta_{E,gl} K_2 \triangleq \forall \omega \in \Omega.\ K_1 \sqsubseteq^\alpha_\omega K_2$$

As for the interest of such a rule, which is detailed in the next two subsections, it is actually twofold. A first advantage is that it provides a compromise between a fully symmetric proof rule (circular reasoning) and a fully asymmetric one (assume-guarantee). Indeed, circular reasoning allows the environment E of the component K to rely on properties of K as much as K is allowed to rely on E. On the other hand, assume-guarantee denies E the possibility to rely on any property of K. In between, whenever circular reasoning is not sound, pseudo-circular reasoning reintroduces some asymmetry while still allowing E to rely on some properties of K.

A second, somewhat unexpected advantage of this rule is that it can also be applied for $\sqsubseteq^\beta$ *weaker* than $\sqsubseteq^\alpha$, thus relaxing rule (**CR**) when it is unnecessarily strong. Chapter 6 presents applications of both approaches in the context of the HRC L0 and L1 frameworks where two refinements under context are considered, one supporting circular reasoning and the other not.

3.3.1 Relaxing assume-guarantee reasoning

As already discussed, there are refinement relations for which circular reasoning is not sound. Then, one is left with the assume-guarantee proof rule, which is much more restrictive with respect to E. Now, suppose that $\sqsubseteq^\alpha$ does not allow circular reasoning, but that we have a second, *stronger* refinement under context $\sqsubseteq^\beta$ which is strong enough to guarantee pseudo-circular reasoning.

In this case, we can use rule (**PCR**) instead of (**AG**) for dominance checks. Note that the question of finding the actual proof remains, just like for assume-guarantee reasoning: which part of the system is supposed to refine its guarantee according to $\sqsubseteq^\alpha$ and which part according to $\sqsubseteq^\beta$? However, if $\sqsubseteq^\beta$ is weaker than refinement in any context — as should be the case — then (**PCR**) relaxes (**AG**) and thus allows establishing dominance more often than (**AG**).

3.3.2 Relaxing circular reasoning

Now, is there any interest in using (**PCR**) when $\sqsubseteq^\beta$ is *weaker* than $\sqsubseteq^\alpha$? Suppose that circular reasoning is sound for $\sqsubseteq^\alpha$ and that we have another, weaker, refinement under context relation $\sqsubseteq^\beta$ such that pseudo-circular reasoning is sound for $\sqsubseteq^\alpha$ and $\sqsubseteq^\beta$. In this case, rule (**PCR**) relaxes rule (**CR**) for circular reasoning. In particular we prove that pseudo-circular reasoning allows the following sufficient condition for dominance (when $\sqsubseteq^\alpha$ is stronger than $\sqsubseteq^\beta$).

Theorem 3.3.1 *If* $\forall i$, $\exists gl_{E_i}$, $gl \circ gl_I = gl_i \circ gl_{E_i}$ *and pseudo-circular reasoning is sound for* $\sqsubseteq^\alpha$ *and* $\sqsubseteq^\beta$*, then to prove that* $\{\mathcal{C}_i\}_{i=1}^n$ *dominates* $\mathcal{C}$ *w.r.t.* gl_I*, it is sufficient to prove:*

$$\begin{cases} gl_I\{G_1, \ldots, G_n\} \models^\alpha \mathcal{C} \\ \forall i, gl_{E_i}\{A, G_1, \ldots, G_{i-1}, G_{i+1}, \ldots, G_n\} \models^\beta \mathcal{C}_i^{-1} \end{cases}$$

This is useful for two reasons: one is that pseudo-circular reasoning allows establishing dominance more often than circular reasoning, as the condition that must be fulfilled by the environment of each component is weaker than the one for circular reasoning. In particular, if various methods or tools are used in the same framework, and some of them can only guarantee refinement with respect to $\sqsubseteq^\beta$, then they can still be used in conjunction with the pseudo-circular approach.

The other reason is that $\sqsubseteq^\beta$ may be not only weaker but also less costly to check than $\sqsubseteq^\alpha$ (this will be the case of $\sqsubseteq^{L0}$ compared to $\sqsubseteq^{L1}$ in Chapter 6). In this case, pseudo-circular reasoning increases the scalability of dominance checks, as only one satisfaction check with respect to $\sqsubseteq^\alpha$ has to be performed while n satisfaction checks are needed with respect to $\sqsubseteq^\beta$.

3.4 Proofs

NB: Remember that the following theorem is given for $\sqsubseteq^\alpha$ stronger than or equal to $\sqsubseteq^\beta$.

Theorem 3.3.1 *If* $\forall i,\ \exists gl_{E_i},\ gl \circ gl_I = gl_i \circ gl_{E_i}$ *and pseudo-circular reasoning is sound for* $\sqsubseteq^\alpha$ *and* $\sqsubseteq^\beta$*, then to prove that* $\{\mathcal{C}_i\}_{i=1}^n$ *dominates* $\mathcal{C}$ *w.r.t.* gl_I*, it is sufficient to prove that:*

$$\begin{cases} gl_I\{G_1,\ldots,G_n\} \models^\alpha \mathcal{C} \\ \forall i,\ gl_{E_i}\{A, G_1,\ldots,G_{i-1},G_{i+1},\ldots,G_n\} \models^\beta \mathcal{C}_i^{-1} \end{cases}$$

Proof. For every $i \in [1,n]$, let K_i be a component on $\mathcal{P}_i$. Suppose the following:

1. $\forall i,\ \exists gl_{E_i},\ gl \circ gl_I = gl_i \circ gl_{E_i}$
2. $gl_I\{G_1,\ldots,G_n\} \sqsubseteq^\alpha_{A,gl} G$
3. $\forall i,\ gl_{E_i}\{A, G_1,\ldots,G_{i-1},G_{i+1},\ldots,G_n\} \sqsubseteq^\beta_{G_i,gl_i} A_i$
4. $\forall i,\ K_i \sqsubseteq^\alpha_{A_i,gl_i} G_i$

We aim at proving that $gl_I\{K_1,\ldots,K_n\} \models^\alpha \mathcal{C}$, that is: $gl_I\{K_1,\ldots,K_n\} \sqsubseteq^\alpha_{A,gl} G$. For this, we show by induction that for any l in $[0,n]$, for any partition $\{J,K\}$ of $[1,n]$ such that $|J| = l$:

$$\begin{cases} gl_I\{\mathcal{K}^J \cup \mathcal{G}^K\} \sqsubseteq^\alpha_{A,gl} G \\ \forall i \in K,\ gl_{E_i}\{A, \mathcal{E}_i^{J,K}\} \sqsubseteq^\beta_{G_i,gl_i} A_i \end{cases}$$

with $\mathcal{K}^J = \{K_j\}_{j\in J}$, $\mathcal{G}^K = \{G_k\}_{k\in K}$ and with $\mathcal{E}_i^{J,K} = \mathcal{K}^J \cup (\mathcal{G}^K \backslash \{G_i\})$.

- $l = 0$. By (2) and (3) the property holds.
- $0 \leq l < n$. We suppose that our property holds for l. Let $\{J', K'\}$ be a partition of $[1,n]$ such that $|J'| = l+1$. Let q be an element of J'. We fix $J = J' \backslash \{q\}$ and $K = K' \cup \{q\}$.

Step 1 We first prove that $gl_I\{\mathcal{K}^{J'} \cup \mathcal{G}^{K'}\} \sqsubseteq^\alpha_{A,gl} G$.

$$\begin{cases} K_q \sqsubseteq^\alpha_{A_q,gl_q} G_q \text{ from (4)} \\ gl_{E_q}\{A,\ \mathcal{E}_q^{J,K}\} \sqsubseteq^\beta_{G_q,gl_q} A_q \end{cases}$$

The second property is our recurrence hypothesis, as $q \in K$. Thus, by circular reasoning (**CR**):

$$K_q \sqsubseteq^\alpha_{gl_{E_q}\{A,\mathcal{E}_q^{J,K}\},gl_q} G_q$$

As refinement under context is preserved by composition, we obtain by (**CMP**):

$$gl_I\{K_q, \mathcal{E}_q^{J,K}\} \sqsubseteq^{\alpha}_{A,gl} gl_I\{G_q, \mathcal{E}_q^{J,K}\}$$

This is equivalent to $gl_I\{\mathcal{K}^{J'} \cup \mathcal{G}^{K'}\} \sqsubseteq^{\alpha}_{A,gl} gl_I\{\mathcal{K}^J \cup \mathcal{G}^K\}$.
Finally, by using the recurrence hypothesis: $gl_I\{\mathcal{K}^{J'} \cup \mathcal{G}^{K'}\} \sqsubseteq^{\alpha}_{A,gl} G$.

Step 2 We now have to prove that:

$$\forall i \in K',\ gl_{E_i}\{A, \mathcal{E}_i^{J',K'}\} \sqsubseteq^{\beta}_{G_i,gl_i} A_i$$

We fix $i \in K'$. We have proved in step 1 that:

$$K_q \sqsubseteq^{\alpha}_{gl_{E_q}\{A,\mathcal{E}_q^{J,K}\},gl_q} G_q$$

As $\sqsubseteq^{\alpha}$ is stronger than $\sqsubseteq^{\beta}$, we also have:

$$K_q \sqsubseteq^{\beta}_{gl_{E_q}\{A,\mathcal{E}_q^{J,K}\},gl_q} G_q$$

$K = K' \cup \{q\}$, so $i \in K$. Thus, by compositionality (**CMP**), we obtain:

$$gl_{E_i}\{K_q, A, \mathcal{E}_q^{J,K\setminus\{i\}}\} \sqsubseteq^{\beta}_{G_i,gl_i} gl_{E_i}\{G_q, A, \mathcal{E}_q^{J,K\setminus\{i\}}\}$$

This boils down to $gl_{E_i}\{A, \mathcal{E}_i^{J',K'}\} \sqsubseteq^{\beta}_{G_i,gl_i} gl_{E_i}\{A, \mathcal{E}_i^{J,K}\}$.
Hence, using the recurrence hypothesis: $gl_{E_i}\{A, \mathcal{E}_i^{J',K'}\} \sqsubseteq^{\beta}_{G_i,gl_i} A_i$.

Conclusion By applying our property to $l = n$, we get:

$$gl_I\{K_1, \ldots, K_n\} \sqsubseteq^{\alpha}_{A,gl} G$$

□

Chapter 4

A contract framework for the BIP semantic level

From now on, Part I is devoted to providing and discussing meaningful contract frameworks illustrating all the concepts introduced in the previous chapters. More precisely, it is organized as follows:

– Chapter 4 defines a contract framework for the semantic level of BIP.
– Chapter 5 introduces two component frameworks based on BIP that structure interaction and allow encapsulation. These component frameworks form the basis of two contract frameworks in Chapter 6, respectively for I/O interface automata and for the SPEEDS framework L1.
– Chapter 6 presents applications of our methodology to I/O interface automata and then to the SPEEDS framework. contract frameworks based on the component frameworks of the previous chapter.
– Chapter 7 discusses in depth the differences between the various refinement under context introduced for LTS in the previous chapters. It also proposes contract frameworks for MTS and then LTS with priorities as well as MTS with priorities. Refinement in any context and structural consistency are also discussed.

In this short chapter, we define a first simple contract framework corresponding to the BIP semantic level. At the semantic level, a component can be seen as an LTS: that is, we do not consider composite components. Also, we do not deal with priorities in this variant. This simple framework presents a first solution to the issue of finding a refinement under context for which circular reasoning is sound.

We present the proofs in some detail for two reasons: first, to show that they are technical but quite simple; second, because the proofs for the other frameworks are similar. Before starting with the definitions, let us recall the definitions that are required and the proofs that we have to provide.

4.1 Necessary ingredients for a successful encoding

Here is a summary of the definitions required to build a contract framework. One needs:

1. a component framework consisting of:
   (a) a set of components $\mathcal{K}$
   (b) an equivalence relation $\cong$ on these components
   (c) a set of glue operators GL
   (d) a composition operator $\circ$ on these glues
2. a conformance relation $\preccurlyeq$
3. a set of refinement under context relations $\{\sqsubseteq\}_{\omega\in\Omega}$

Furthermore, there are some coherence conditions to establish between these various notions. In the following, items introduced by ▲ relate to conditions referring to the component framework while those preceded by ■ refer to the contract framework. These conditions are between, respectively:

- ▲ composition of glues and equivalence of components, that is, $\circ$ and $\cong$
- ■ conformance and refinement under context, i.e., $\preccurlyeq$ and $\{\sqsubseteq\}_{\omega\in\Omega}$
- ■ refinement under context and equivalence of components, i.e., $\{\sqsubseteq\}_{\omega\in\Omega}$ and $\cong$

Then, before using circular reasoning, the following questions must be answered:

- ■ Is refinement under context preserved by composition?
- ■ Is circular reasoning sound?
- ▲ Given a glue gl on $\mathcal{P}$ (i.e. $S_{gl} = \mathcal{P}$) and gl_1 on $\mathcal{P}_1 \subseteq \mathcal{P}$, can we determine whether there exists gl_2 such that $gl = gl_1 \circ gl_2$?

If the answer to the last question is negative, then such gl_2 must be provided by the user for each contract before applying the sufficient condition for dominance 2.3.5.

4.2 The BIP semantic contract framework

Let us now introduce formally the BIP semantic contract framework. The component framework that we define in this section is strongly related to the framework presented in the preliminaries of this thesis. However, there are two key differences:

– We do not consider priorities. How to enrich this framework with priorities will be explained in Chapter 7.
– We do not handle hierarchical components. This will also be discussed later.

These strong restrictions have many consequences. First, by getting rid of priorities, the compositional semantics and the closed one defined in the preliminaries become identical. In particular, the compositional semantics of a component can be expressed as an LTS. In other words, for any given composite (possibly hierarchical) component, there exists an atomic component that has the same compositional semantics. This implies that in a verification process, it is possible to consider only atomic components, and this is what we do in this chapter. As a result, a glue $\mathcal{I}$ as defined here does not build a composite component from a set of components $K_1 \dots K_n$; instead it associates with $\{K_i\}_{i=1}^n$ what would be defined in the preliminaries as the compositional semantics of $\mathcal{I}\{K_1, \dots, K_n\}$. This is the reason why we call this framework a semantic framework.

As before, we suppose that a set of all possible ports $Ports$ is given. The BIP semantic framework is as follows:

– A component K is an LTS $(Q, q^0, 2^{\mathcal{P}}, \longrightarrow)$, where $\mathcal{P}$ is called the *interface* of K.
– $\cong$ is syntactic equality, possibly after renaming of states [1].
– A glue $\mathcal{I}$ on a set of ports P is defined by a set of interactions (i.e. non empty sets of ports) in $\mathcal{P}$ and is such that $\mathcal{I}\{K_i\}_{i=1}^n$ is the n-ary product of LTS $K_1, \dots, K_n$ where transitions with labels in the same interaction are synchronized. This is formally defined below.
– $\mathcal{I}_1 \circ \mathcal{I}_2$ is the glue defined by the set of interactions which are allowed by both $\mathcal{I}_1$ and $\mathcal{I}_2$ according to the composition of interaction sets of Definition 1.2.9.

We adopt the convention that in a component $K = (Q, q^0, 2^{\mathcal{P}}, \longrightarrow)$, it holds that $\forall q \in Q : q \xrightarrow{\emptyset} q$. As already mentioned and unless otherwise stated, a component K_i is implicitly defined as a tuple $(Q_i, q_i^0, 2^{\mathcal{P}_i}, \longrightarrow_i)$.

Definition 4.2.1 (Interaction, Glue) *Consider a set of ports $P \subseteq Ports$. An* interaction *in P is a non-empty subset of P. A set of interactions $\mathcal{I}$ in P defines a* glue *$\mathcal{I}$ which associates, with every set of components $\{K_i\}_{i=1}^n$ with disjoint interfaces $\{\mathcal{P}_i\}_{i=1}^n$ such that $P = \bigcup_{i=1}^n \mathcal{P}_i$, a component*

1. Note that it could as well be defined as bisimulation.

$\mathcal{I}\{K_1, ..., K_n\}$ *on* P *defined as the LTS* $(Q, q^0, 2^P, \longrightarrow)$ *where:*

– $Q = Q_1 \times ... \times Q_n$

– $q^0 = (q_1^0, ..., q_n^0)$

– *given two states* $q = (q_1, \ldots, q_n)$ *and* $q' = (q_1', \ldots, q_n')$ *in* Q *and an interaction* $\alpha \in \mathcal{I}$, $q \xrightarrow{\alpha} q'$ *if and only if* $\forall i, q_i \xrightarrow{\alpha_i}_i q_i'$ *where* $\alpha_i = \alpha \cap \mathcal{P}_i$.

Now that we have fully defined the component framework that we are studying in this chapter, we can focus on finding appropriate refinement relations. The conformance that we choose is simulation (see Definition 1.1.11), as is very often the case for frameworks based on LTS. The reason for that is that simulation preserves safety properties, which are essential to the design of many systems.

Definition 4.2.2 (Conformance) $K_1 \preccurlyeq K_2 \triangleq K_1$ *simulates* K_2.

Even for this classical definition of conformance, how to define refinement under context is an interesting question. In Chapter 2, the following usual definition was proposed:

$$K_1 \sqsubseteq^{\preccurlyeq}_{E,\mathcal{I}} K_2 \triangleq \mathcal{I}\{K_1, E\} \preccurlyeq \mathcal{I}\{K_2, E\}$$

It was then illustrated in Section 2.3.2, when circular reasoning was presented, that this usual definition yields a refinement under context $\sqsubseteq^{\preccurlyeq}$ for which circular reasoning is not sound in presence of non-determinism or strong synchronization. Our framework here offers both, thus we need another relation if we want to apply the sufficient condition provided in Section 2.3.3. The definition below tackles these two problems by "breaking the symmetry" between component and environment within the definition: in a given state, a component make take into account only the past actions of its environment E, and not the interactions possible in the current state of E.

Definition 4.2.3 (Refinement under context) Refinement under context *is defined as follows:* $K_1 \sqsubseteq_{E,\mathcal{I}} K_2$ *if and only if there exists a relation* $\mathcal{R} \subseteq (Q_1 \times Q_E) \times Q_2$ *such that:*

– $(q_1^0, q_E^0)\, \mathcal{R}\, q_2^0$

– *if* $(q_1, q_E)\, \mathcal{R}\, q_2$, $q_1 \xrightarrow{\alpha_K}_1 q_1'$ *and* $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}$, *then there exists* q_2' *such that* $q_2 \xrightarrow{\alpha_K}_2 q_2'$ *and any* q_E' *such that* $q_E \xrightarrow{\alpha_E}_E q_E'$ *satisfies* $(q_1', q_E')\, \mathcal{R}\, q_2'$

We use the convention that $\forall q \xrightarrow{\emptyset} q$, so the above condition includes cases where only K_1 or only E move on. This refinement is in fact a stronger version of $\sqsubseteq^{\preccurlyeq}$, as will be proved in Chapter 7. Suppose that $(q_1, q_E)\, \mathcal{R}\, q_2$ and $q_1 \xrightarrow{\alpha_K}_1 q_1'$. Then the above definition can be rephrased as follows:

- If α_K is structurally forbidden, that is, there is no interaction α allowed by $\mathcal{I}$ that projects onto α_K, then it is not required that transition $q_1 \xrightarrow{\alpha_K}_1 q_1'$ have a counterpart in $\longrightarrow_2$.
- Otherwise, $q_1 \xrightarrow{\alpha_K}_1 q_1'$ must have a counterpart $q_2 \xrightarrow{\alpha_K}_2 q_2'$, even if the environment E does not offer any interaction that could match α_K.
- Furthermore, this counterpart must be the same for all possible moves of E that have the same label: any q_E' such that $q_E \xrightarrow{\alpha_E}_E q_E'$ satisfies $(q_1', q_E') \, \mathcal{R} \, q_2'$.
- Note that q_1' and q_2' have to be related by $\mathcal{R}$ only if there exists some q_E' such that $q_E \xrightarrow{\alpha_E}_E q_E'$. That is, two states q_1' and q_2' are related only if (q_1', q_E') and (q_2', q_E') are both reachable in respectively $\mathcal{I}\{K_1, E\}$ and $\mathcal{I}\{K_2, E\}$.

This choice of refinement under context is one among possible relations consistent with simulation and for which circular reasoning is sound. Other relations are discussed in Chapter 7.

We have defined a contract framework in which circular reasoning is sound: the next section contains all the proofs required to establish this result. Although a bit technical, these proofs are simple and short.

4.3 Coherence conditions

We first focus on the conditions required from our component framework, that is: 1. coherence between $\circ$ and $\cong$, then: 2. possibility to structure systems. After this, we will present the proofs related to conformance and refinement under context, namely: 3. consistency of $\preccurlyeq$ and $\{\sqsubseteq\}_{\omega \in \Omega}$, 4. preservation of refinement under context by composition and 5. soundness of circular reasoning. The condition that $\{\sqsubseteq\}_{\omega \in \Omega}$ and $\cong$ must be coherent is trivially true.

4.3.1 Composition of glues and equivalence of components

Proof. The first condition relates to composition of glues. We must show that for any glues $\mathcal{I}_1, \mathcal{I}_2$ and any sets of components $\mathcal{K}^i$ such that all terms are defined, $\mathcal{I}_1\{\mathcal{I}_2\{\mathcal{K}^1\}, \mathcal{K}^2\} \cong (\mathcal{I}_1 \circ \mathcal{I}_2)\{\mathcal{K}^1 \cup \mathcal{K}^2\}$.
- We write the proof for $\mathcal{K}^1 = \{K_1, K_2\}$ and $\mathcal{K}^2 = \{K_3\}$. The generalization should be clear. The syntactic differences between both components are bold and colored.

$\mathcal{I}_1\{\mathcal{I}_2\{\mathcal{K}^1\}, \mathcal{K}^2\} = (Q, q^0, \mathcal{P}, \longrightarrow)$, where:

- $Q = (Q_1 \times Q_2) \times Q_3$
- $q^0 = ((q_1^0, q_2^0), q_3^0)$
- $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3$
- for $q = ((q_1, q_2), q_3)$ and $q' = ((q_1', q_2'), q_3')$ in Q and $\alpha \in \boldsymbol{\mathcal{I}_1}$, $q \xrightarrow{\alpha} q'$ if and only if $\boldsymbol{\alpha \cap (\mathcal{P}_1 \cup \mathcal{P}_2) \in \mathcal{I}_2}$ and $q_i \xrightarrow{\alpha_i}_i q_i'$ for $i \in \{1, 2, 3\}$, where $\alpha_i = \alpha \cap \mathcal{P}_i$

$(\mathcal{I}_1 \circ \mathcal{I}_2)\{\mathcal{K}^1 \cup \mathcal{K}^2\} = (Q, q^0, \mathcal{P}, \longrightarrow)$, where:

- $Q = Q_1 \times Q_2 \times Q_3$
- $q^0 = (q_1^0, q_2^0, q_3^0)$
- $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3$
- given $q = (q_1, q_2, q_3)$ and $q' = (q_1', q_2', q_3')$ in Q and $\alpha \in \boldsymbol{\mathcal{I}_1 \circ \mathcal{I}_2}$, $q \xrightarrow{\alpha} q'$ if and only if $q_i \xrightarrow{\alpha_i}_i q_i'$ for $i \in \{1, 2, 3\}$, where $\alpha_i = \alpha \cap \mathcal{P}_i$

Thus, after renaming states of $\mathcal{I}_1\{\mathcal{I}_2\{\mathcal{K}^1\}, \mathcal{K}^2\}$ — a state of the form $((q_1, q_2), q_3)$ becomes (q_1, q_2, q_3) — there only remains to prove that interactions from both LTS are the same. That is, we must show that $\alpha \in \mathcal{I}_1 \circ \mathcal{I}_2$ if and only if $\alpha \in \mathcal{I}_1$ and $\alpha \cap (\mathcal{P}_1 \cup \mathcal{P}_2) \in \mathcal{I}_2$. As $\mathcal{I}_1 \circ \mathcal{I}_2$ and $\mathcal{I}_1$ are both defined on $\mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3$ while $\mathcal{I}_2$ is defined on $\mathcal{P}_1 \cup \mathcal{P}_2$, this is easily obtained: by definition $\alpha \in \mathcal{I}_1 \circ \mathcal{I}_2$ if and only if $\alpha \cap (\mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3) \in \mathcal{I}_1$ and $\alpha \cap (\mathcal{P}_1 \cup \mathcal{P}_2) \in \mathcal{I}_2$. □

4.3.2 Structuring systems

The second coherence condition with respect to the component framework is about structuring a composition according to any given partition.

Definition 4.3.1 (Decomposition of interaction sets) *A interaction set $\mathcal{I}$ on $\mathcal{P}$ can always be put under the form $\mathcal{I}' \circ \mathcal{I}_1 \circ \mathcal{I}_2$, where $\mathcal{I}'$ is defined on $\mathcal{P}$ and the $\mathcal{I}_i$ are defined on a partition $\{\mathcal{P}_1, \mathcal{P}_2\}$ of $\mathcal{P}$. A trivial decomposition is to choose $\mathcal{I}' = \mathcal{I}$ and $\mathcal{I}_i = 2^{\mathcal{P}_i}$. A more interesting decomposition is the following: $\mathcal{I}' = \mathcal{I}$ and $\mathcal{I}_i = \{\alpha \cap \mathcal{P}_i \mid \alpha \in \mathcal{I}\}$.*

As a consequence, if we suppose given $\mathcal{I}$ on $\mathcal{P}$ and $\mathcal{I}_1$ on $\mathcal{P}_1 \subseteq \mathcal{P}$, there exists $\mathcal{I}_2$ such that $\mathcal{I} = \mathcal{I}_1 \circ \mathcal{I}_2$ if and only if any interaction $\alpha \in \mathcal{I}$ is such that $\alpha \cap \mathcal{P}_1 \in \mathcal{I}_1$. If so, then $\mathcal{I}_2 = \mathcal{I}$ is already a solution, but more interestingly, one can choose $\mathcal{I}_2 = \mathcal{I} \backslash \mathcal{I}_1$.

4.3.3 Consistency between $\sqsubseteq$ and $\preccurlyeq$

Proof. Suppose that $K_1 \sqsubseteq_{E,\mathcal{I}} K_2$. Let us show that $\mathcal{I}\{K_1, E\}$ simulates $\mathcal{I}\{K_2, E\}$.
- Let $\mathcal{R}' \subseteq (Q_1 \times Q_E) \times (Q_2 \times Q_E)$ be defined as:

$$\boldsymbol{(q_1, q_E)\, \mathcal{R}'\, (q_2, q_E') \triangleq q_E' = q_E \wedge (q_1, q_E)\, \mathcal{R}\, q_2}$$

- We show that $\mathcal{R}'$ is a simulation relating states of $\mathcal{I}\{K_1, E\}$ and $\mathcal{I}\{K_2, E\}$
- The first condition, namely that $(q_1^0, q_E^0)\, \mathcal{R}'\, (q_2^0, q_E^0)$, holds by definition.
- Now, suppose $(q_1, q_E)\, \mathcal{R}'\, (q_2, q_E)$ and $(q_1, q_E) \xrightarrow{\alpha} (q_1', q_E')$. We have to prove that:

$$\exists q_2' \text{ s.t. } (q_2, q_E) \xrightarrow{\alpha} (q_2', q_E') \text{ and } (q_1', q_E')\, \mathcal{R}'\, (q_2', q_E')$$

- We decompose α as $\alpha_K \cup \alpha_E$, which implies that $q_1 \xrightarrow{\alpha_K}_1 q_1'$ and $q_E \xrightarrow{\alpha_E}_E q_E'$.
- From $(q_1, q_E)\,\mathcal{R}\,q_2$ and $q_1 \xrightarrow{\alpha_K}_1 q_1'$, we conclude that there exists q_2' such that $q_2 \xrightarrow{\alpha_K}_2 q_2'$ and $\forall q_E'',\ q_E \xrightarrow{\alpha_E}_E q_E'' \implies (q_1', q_E'')\,\mathcal{R}\,q_2'$.
- In particular, we have $(q_1', q_E')\,\mathcal{R}\,q_2'$. This implies that $(q_1', q_E')\mathcal{R}'(q_2', q_E')$.
- Besides, $q_2 \xrightarrow{\alpha_K}_2 q_2'$ and $q_E \xrightarrow{\alpha_E}_E q_E'$, so $(q_2, q_E) \xrightarrow{\alpha} (q_2', q_E')$.
- Hence the result. □

4.3.4 Preservation of refinement by composition

Proof. Define a context $(E, \mathcal{I})$ for an interface $\mathcal{P}$ and $\mathcal{I}_E,\ E_1, E_2$ such that $E = \mathcal{I}_E\{E_1, E_2\}$. Suppose that $K_1 \sqsubseteq_{E,\mathcal{I}} K_2$. Let us show that $\mathcal{I}_1\{K_1, E_1\} \sqsubseteq_{E_2,\mathcal{I}_2} \mathcal{I}_1\{K_2, E_1\}$.
- As $Q_E = Q_{E_1} \times Q_{E_2}$, there exists a relation $\mathcal{R} \subseteq (Q_1 \times (Q_{E_1} \times Q_{E_2})) \times Q_2$ with the properties stated in Definition 4.2.3.
- We define $\mathcal{R}' \subseteq ((Q_1 \times Q_{E_1}) \times Q_{E_2}) \times (Q_2 \times Q_{E_1})$ by:

$$\boldsymbol{((q_1, q_{E_1}), q_{E_2})\,\mathcal{R}'(q_2, q_{E_1}) \triangleq (q_1, (q_{E_1}, q_{E_2}))\,\mathcal{R}\,q_2}$$

- Let us prove that $\mathcal{R}'$ has the expected properties:
- First, $((q_1^0, q_{E_1}^0), q_{E_2}^0)\,\mathcal{R}'(q_2^0, q_{E_1}^0)$ holds by definition.
- Suppose now that $((q_1, q_{E_1}), q_{E_2})\,\mathcal{R}'(q_2, q_{E_1})$ and $(q_1, q_{E_1}) \xrightarrow{\alpha'} (q_1', q_{E_1}')$ with $\alpha' = \alpha_K \cup \alpha_{E_1}$. Suppose $\alpha = \alpha' \cup \alpha_{E_2}$ is in $\mathcal{I}_2$. We must prove that there exists q_2' such $(q_2, q_{E_1}) \xrightarrow{\alpha'} (q_2', q_{E_1}')$ and any q_{E_2} such that $q_{E_2} \xrightarrow{\alpha_{E_2}} q_{E_2}'$ satisfies $((q_1', q_{E_1}'), q_{E_2}')\,\mathcal{R}'(q_2', q_{E_1}')$.
- Note that $(q_1, q_{E_1}) \xrightarrow{\alpha'} (q_1', q_{E_1}')$ implies that $q_1 \xrightarrow{\alpha_K} q_1'$ and $q_{E_1} \xrightarrow{\alpha_{E_1}} q_{E_1}'$.
- As $(q_1, (q_{E_1}, q_{E_2}))\,\mathcal{R}\,q_2$ and $q_1 \xrightarrow{\alpha_K} q_1'$, we know that there exists q_2' such that $q_2 \xrightarrow{\alpha_K} q_2'$ and any (q_{E_1}, q_{E_2}) such that $(q_{E_1}, q_{E_2}) \xrightarrow{\alpha_{E_1} \cup \alpha_{E_2}} (q_{E_1}', q_{E_2}')$ satisfies $(q_1', (q_{E_1}', q_{E_2}'))\,\mathcal{R}\,q_2'$.
- Let us show that this q_2' has the expected properties.
- First, as $q_{E_1} \xrightarrow{\alpha_{E_1}} q_{E_1}'$ and $q_2 \xrightarrow{\alpha_K} q_2'$, it holds that $(q_2, q_{E_1}) \xrightarrow{\alpha'} (q_2', q_{E_1}')$.
- Second, let q_{E_2} be such that $q_{E_2} \xrightarrow{\alpha_{E_2}} q_{E_2}'$. Then $(q_{E_1}, q_{E_2}) \xrightarrow{\alpha_{E_1} \cup \alpha_{E_2}} (q_{E_1}', q_{E_2}')$, which implies that $(q_1', (q_{E_1}', q_{E_2}'))\,\mathcal{R}\,q_2'$. This, by definition, implies that $((q_1', q_{E_1}'), q_{E_2}')\,\mathcal{R}'(q_2', q_{E_1}')$. □

4.3.5 Soundness of circular reasoning

Proof. Suppose that $K \sqsubseteq_{A,\mathcal{I}} G$ and $E \sqsubseteq_{G,\mathcal{I}} A$. Let us show that $K \sqsubseteq_{E,\mathcal{I}} G$.
- As $K \sqsubseteq_{A,\mathcal{I}} G$ and $E \sqsubseteq_{G,\mathcal{I}} A$, there exist two relations $\mathcal{R}_1$ and $\mathcal{R}_2$ which are defined respectively

on $(Q_K \times Q_A) \times Q_G$ and $(Q_E \times Q_G) \times Q_A$ as in Definition 4.2.3.
- We define $\mathcal{R} \subseteq (Q_K \times Q_E) \times Q_G$ as follows:

$$\boldsymbol{(q_K, q_E)\, \mathcal{R}\, q_G \triangleq \exists q_A \in Q_A.\, (q_K, q_A)\, \mathcal{R}_1\, q_G \wedge (q_E, q_G)\, \mathcal{R}_2\, q_A}$$

Let us prove that $\mathcal{R}$ is indeed a refinement under context.
- The first condition is obvious: $(q_K^0, q_E^0)\, \mathcal{R}\, q_G^0$.
- Now consider q_K, q_E and q_G such that $(q_K, q_E)\, \mathcal{R}\, q_G$. Let q_A be such that $(q_K, q_A)\, \mathcal{R}_1\, q_G$ and $(q_E, q_G)\, \mathcal{R}_2\, q_A$. Suppose $q_K \xrightarrow{\alpha_K}_K q'_K$ and $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}$. We have to prove that there exists p'_G such that: (a) $q_G \xrightarrow{\alpha_K}_G q'_G$ and (b) $\forall q''_E,\ q_E \xrightarrow{\alpha_E}_E q''_E \implies (q'_K, q''_E)\, \mathcal{R}\, q'_G$.
- As $(q_K, q_A)\, \mathcal{R}_1\, q_G$, $q_K \xrightarrow{\alpha_K}_K q'_K$ and $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}$, there exists q'_G such that (a$_1$) $q_G \xrightarrow{\alpha_K}_G q'_G$ and (b$_1$) $\forall q''_A,\ q_A \xrightarrow{\alpha_E}_A q''_A \implies (q'_K, q''_A)\, \mathcal{R}_1\, q'_G$. We show that this q'_G satisfies (a) and (b).
- Condition (a) is exactly the same as (a$_1$), we focus on condition (b): let q'_E be such that $q_E \xrightarrow{\alpha_E}_E q'_E$. We show that $(q'_K, q'_E)\, \mathcal{R}\, q'_G$.
- As $(q_E, q_G)\, \mathcal{R}_2\, q_A$, $q_E \xrightarrow{\alpha_E}_E q'_E$ and $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}$, there exists q'_A such that (a$_2$) $q_A \xrightarrow{\alpha_E}_A q'_A$ and (b$_2$) $\forall q''_G,\ q_G \xrightarrow{\alpha_K}_G q''_G \implies (q'_E, q''_G)\, \mathcal{R}_2\, q'_A$.
- Thus, applying (b$_1$) to this q'_A, we obtain that $(q'_K, q'_A)\, \mathcal{R}_1\, q'_G$.
- Besides, as $q_G \xrightarrow{\alpha_K}_G q'_G$, by applying (b$_2$) to this q'_G we obtain $(q'_E, q'_G)\, \mathcal{R}_2\, q'_A$.
- Finally, according to the definition of $\mathcal{R}$, we can conclude that $(q'_K, q'_E)\, \mathcal{R}\, q'_G$. $\square$

Chapter 5

Two component frameworks for BIP

In this chapter, we present two variants of the BIP framework which we use in the sequel to build contract frameworks, and which had never been formalized before. Both variants structure interactions using connectors and provide mechanisms for encapsulation: that is, new ports are associated with connectors and the interface of a composite component is the set of ports corresponding to its internal connectors. Encapsulation enables to abstract the behavior of a component in a *black-box* manner, by describing only which connector is triggered but not exactly which interaction takes place. This makes it much more complicated to provide a compositional semantics than in the white-box BIP framework presented in the preliminaries. There exists two variants of the BIP framework handling encapsulation: one is the formalism defining the BIP2 tool-chain, which does not allow flattening; another is [BJS09], which handles only a specific class of interaction models (sets of rendezvous connectors progress) and no priority. In fact, it is not possible to combine the full expressivity of BIP with flattening and encapsulation. As a result, the two variants proposed here are restrictions:

1. The first variant, which we call BIP with maximal progress, has the following characteristics: behaviors are LTS, interactions provide encapsulation, priorities are limited to maximal progress and we use two one-shot semantics, one that is compositional and black-box, and another for closed systems that is white-box. This combination of features is close to the actual implementation in the BIP2 tool-chain, but has never been formalized like this before.
2. The second variant, which we call multi-shot BIP, differs from the first one mainly in that its compositional semantics is multi-shot. Thus, this variant is powerful enough to encompass synchronous systems. As a side effect, it cannot handle priorities.

Table 5.1 presents a summary of the differences between the variants of BIP presented in this thesis. In more detail, these differences are related to each other as follows:

	white-box	first variant	second variant
structured interaction	no	yes	yes
encapsulation	no	yes	yes
priorities	yes	only maximal progress	no
transitions labeled by	interactions	ports	interactions
interaction model on	set of ports	set of disjoint interfaces	set of ports
restrictions	none	each interaction in at most one connector	none
restrictions	none	0 or 1 port of a component per connector	none

Table 5.1 – Summary of the differences between the variants of BIP presented in this thesis

– As already stated, the main difference between the two variants and the white-box framework presented in Chapter 5 is that these variants provide a mechanism for encapsulation: they structure interactions into connectors, and only these connectors appear at the interface of a composite component. That is, the environment of such a component cannot distinguish interactions from the same connector. Note that in [GS05], interactions are also structured into connectors although no encapsulation is possible. We have chosen to present connectors only as a means for encapsulation. Also, note that our closed semantics are white-box, as encapsulation is neither necessary nor useful for a system that cannot be composed anymore.
– Providing semantics for a framework encompassing encapsulation and flattening is a challenging task. We propose here two different solutions. The first one preserves a subset of the possible priority orders, namely maximal progress, because they can be dealt with in a compositional way if the following restriction is made: an interaction cannot be part of several connectors. To ensure this property in hierarchical components, we have to require furthermore that connector are used only to connect ports of different components and never ports of the same component. This in turn implies that interaction models have to be defined on sets of interfaces so that it is possible to distinguish which port belongs to which component when defining connectors. Finally, this makes labeling transitions by interactions useless, as a component cannot activate more than one port at the same time.

Note that there are other possibilities to build a BIP framework encompassing encapsulation and flattening. For example, one could define a compositional semantics as in the white-box framework, that is, as a pair made of an LTS and a priority order. However, even in this case, not all priorities can be handled as they have to be expressible at the interface of the component.

This chapter is organized as follows: we present BIP with maximal progress, then multi-shot BIP. Finally, we show how hiding of ports can be encoded in an elegant manner. We leave it to the next

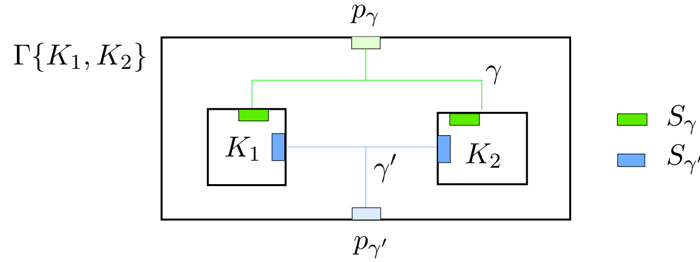


Figure 5.1 – The role of connectors in a composition

chapter to prove that these two frameworks are indeed component frameworks as defined in Chapter 2.

5.1 A first variant: BIP with maximal progress

In this variant, as in the next one, interactions are structured using *connectors*. Priority is restricted to maximal progress within a connector. Even with this strong restriction, we need, in order to provide a compositional semantics, to require furthermore that at most one port of each component (atomic or composite) can be triggered at a time, and to restrict interaction models to sets of connectors with disjoint interaction sets. In this constrained setting, transitions are labeled by ports, and not by interactions. An interaction corresponds to exactly one connector, which simplifies both the conditions related to maximal progress and the proof of consistency between the semantics of a hierarchical component and its flattened form.

Definition 5.1.1 (Atomic component) *An* atomic component *on a interface $\mathcal{P}$ is defined by an LTS $K = (Q, q^0, \mathcal{P}, \longrightarrow)$.*

As in the semantic framework, an *interaction* is represented as a non-empty set of ports, but now we can structure them into connectors, as in the following definition.

Definition 5.1.2 (Connector) *A* connector *γ is defined by a set of ports S_γ called the* support set *of γ, a port p_γ called its* exported port *and a set $\mathcal{I}(\gamma)$ of interactions in S_γ.*

The intuition behind support set and exported port is illustrated in Figure 5.1, where connectors relate in a composition a set of inner ports (of the subcomponents) to an outer port (of the composite component). One should keep in mind that a connector γ, and thus the exported port p_γ, represents a set of interactions rather than a single interaction.

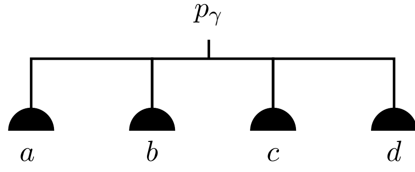


Figure 5.2 – A rendezvous connector

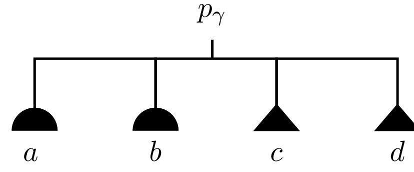


Figure 5.3 – A multiple broadcast

Typical connectors represent rendezvous (only one interaction, equal to the support set), broadcast (all the interactions containing a specific port called *trigger*) and also mutual exclusion (some interactions but not their union). When the set of interactions of a connector is closed by union (no mutual exclusion), we use the following convention for graphically representing connectors: a trigger port (a port p such that $\{p\}$ is an interaction) is associated with a triangle while the other ports, called *synchrons*, are associated with semi-circles. For example, the connector of Figure 5.2 has no trigger and thus only one interaction, namely $\{a, b, c, d\}$, while that of Figure 5.3 has as interaction set all the interactions which contain at least c or d.

When combined with hierarchy of connectors, the representation of interaction sets using triggers and synchrons is quite expressive, and allows many interesting transformations [BS08a]. When this representation is not sufficient (the interaction set of some connector is not closed under union), or when we do not want to represent explicitly the interaction sets of connectors, we only draw the connections between support set and exported port, as in Figure 5.1. We now define *interaction models*, which define sets of connectors that can be used together in order to compose components. As we do not allow connecting two ports of the same component in one connector, an interaction model has to be defined for a given set of interfaces.

Definition 5.1.3 (Interaction model) *An* interaction model Γ *defined on a set of disjoint interfaces* $\{\mathcal{P}_i\}_{i=1}^n$ *is a set of connectors with distinct exported ports and disjoint interaction sets, and such that every connector has a support set included in* $\bigcup_{i=1}^n \mathcal{P}_i$ *and containing at most one port in every* $\mathcal{P}_i$.

In this definition, an interaction corresponds to at most one connector, i.e., if $i \in \mathcal{I}(\gamma)$ and $i \in \mathcal{I}(\gamma')$, then $\gamma = \gamma'$. However, connectors in an interaction model are not required to have pairwise disjoint support sets and a port may be connected through several connectors.

An interaction model Γ defines an interface, denoted $\mathcal{P}_\Gamma$, namely $\{p_\gamma \mid \gamma \in \Gamma\}$. $S_\Gamma = \bigcup_{i=1}^n \mathcal{P}_i$ is called the *support set* of Γ. Besides, $\mathcal{I}(\Gamma)$ denotes the set of all interactions of the connectors in Γ, i.e.: $\mathcal{I}(\Gamma) = \bigcup_{\gamma \in \Gamma} \mathcal{I}(\gamma)$. In Figure 5.1, Γ is composed of connectors γ and γ'.

In this section, the only priority order that we consider is maximal progress, first, because it is not possible in the general case to provide a compositional semantics for priority orders in presence of encapsulation, and second because we only need maximal progress in this thesis.

Definition 5.1.4 (Maximal progress) *Given an interaction model Γ, maximal progress $\prec_{MP}$ is defined by $\forall i, j \in \mathcal{I}(\Gamma)$:*

$$i \prec_{MP} j \triangleq i \subseteq j \wedge \exists \gamma \in \Gamma,\, i \in \mathcal{I}(\gamma) \wedge j \in \mathcal{I}(\gamma)$$

Maximal progress is a priority order that relates interactions of the same connector γ, and gives higher priority to the larger interaction. It applies only to interactions in the same connector for two reasons: one is that otherwise it is not possible to provide a compositional semantics in the form of an LTS; the second reason is that maximal progress makes more sense applied only within a connector.

Definition 5.1.5 (Component, Composite component) *A* component *is either an atomic component or it is inductively defined as the composition of a set of components $\{K_i\}_{i=1}^n$ with disjoint interfaces $\{\mathcal{P}_i\}_{i=1}^n$ using an interaction model Γ on $\{\mathcal{P}_i\}_{i=1}^n$. Such a composition is called a* composite component *on $\mathcal{P}_\Gamma$ and it is denoted $\Gamma\{K_i\}_{i=1}^n$.*

We now provide two semantics enforcing maximal progress — called respectively compositional and closed semantics — which are intended for different purposes. The compositional, black-box semantics must be preserved by composition, and will be used to express the behavior of a component when it is part of a larger system, thus it refers to exported ports. The closed semantics will be used for expressing the behavior of a closed system, that is, a system that has no interaction with an environment. In particular, the closed semantics is white-box in the sense that it reflects the inner interactions taking place and not only the corresponding exported ports at the interface of the system. These differences are illustrated in Figure 5.4, where a simple composite component is given with its two corresponding semantics, compositional (top-right) and closed (down-right). Maximal progress ensures that, although they could be fired separately, b and e are both fired in a single transition.

Consider a composite component K defined by a set of components $\{K_i\}_{i=1}^n$ and an interaction model Γ on $\{\mathcal{P}_i\}_{i=1}^n$.

Definition 5.1.6 (Compositional semantics) *The* compositional semantics *of K is denoted $|K|$ and is defined as $(Q, q^0, \mathcal{P}_\Gamma, \rightsquigarrow)$, where $Q = \prod_{i=1}^n Q_i$, $q^0 = (q_1^0, \ldots, q_n^0)$ and $\rightsquigarrow$ is defined as follows. Given two states $q^1 = (q_1^1, \ldots, q_n^1)$ and $q^2 = (q_1^2, \ldots, q_n^2)$ in Q and $\gamma \in \Gamma$, $q^1 \overset{p_\gamma}{\rightsquigarrow} q^2$ if and only if there exists $\alpha \in \mathcal{I}(\gamma)$ such that:*

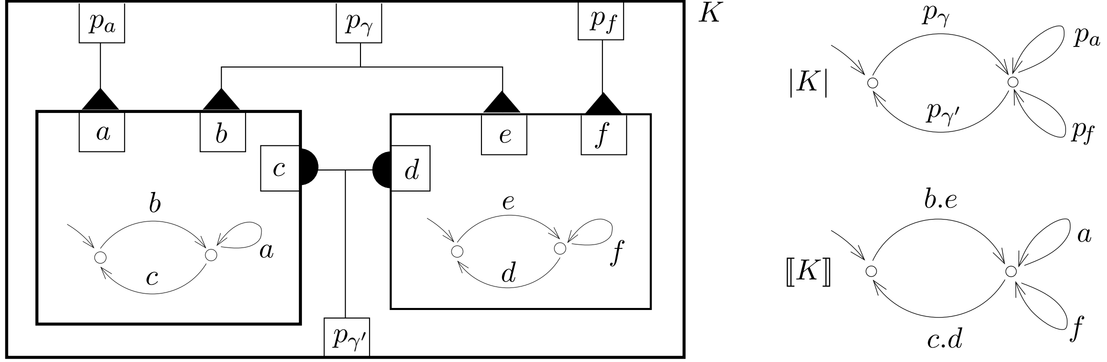


Figure 5.4 – A composite (left) and its compositional (top-right) and closed (bottom-right) semantics

1. *$\forall i$ such that $\alpha \cap \mathcal{P}_i = \emptyset$: $q_i^1 = q_i^2$*
2. *$\forall i$ such that $\alpha \cap \mathcal{P}_i = \{p_i\}$ for some $p_i \in \mathcal{P}_i$: $q_i^1 \overset{p_i}{\rightsquigarrow}_i q_i^2$*
3. *$\nexists \alpha' \in \mathcal{I}(\gamma)$ such that $\alpha \prec_{MP} \alpha'$ and satisfying conditions 1. and 2.*

Components not involved in the interaction do not move. Note also that the priorities resulting from maximal progress can be applied locally. Indeed, filtering is done only between interactions in the same connector, which are indistinguishable at the interface of K. The closed semantics only differs from the compositional semantics in that it is white-box, so labels are interactions in $\mathcal{I}(\Gamma)$ rather than ports in $\mathcal{P}_\Gamma$.

Definition 5.1.7 (Closed semantics) *The* closed semantics *of K is denoted $[\![K]\!]$ and is defined as $(Q, q^0, \mathcal{I}(\Gamma), \longrightarrow)$, where $Q = \prod_{i=1}^n Q_i$, $q^0 = (q_1^0, \ldots, q_n^0)$ and $\longrightarrow$ is defined as follows. Given two states $q^1 = (q_1^1, \ldots, q_n^1)$ and $q^2 = (q_1^2, \ldots, q_n^2)$ in Q and an interaction $\alpha \in \mathcal{I}(\Gamma)$, $q^1 \xrightarrow{\alpha} q^2$ iff:*

1. *$\forall i$ such that $\alpha \cap \mathcal{P}_i = \emptyset$: $q_i^1 = q_i^2$*
2. *$\forall i$ such that $\alpha \cap \mathcal{P}_i = \{p_i\}$ for some $p_i \in \mathcal{P}_i$: $q_i^1 \overset{p_i}{\rightsquigarrow}_i q_i^2$*
3. *$\nexists \alpha' \in \mathcal{I}(\Gamma)$ such that $\alpha \prec_{MP} \alpha'$ and satisfying conditions 1. and 2.*

In this semantics, only interactions that are locally enabled in all concerned components, and furthermore not inhibited by any larger interaction, may be fired.

We still have to describe how interaction models can be composed. The difficulty of this composition lies mainly in handling hierarchical connectors. If $p_{\gamma'} \in S_\gamma$, then γ and γ' can be composed to form a hierarchical connector denoted $\gamma * \gamma'$ (see Figure 5.5) with support set $S_\gamma \cup S_{\gamma'} \backslash \{p_{\gamma'}\}$, with exported port p_γ and whose interaction set is computed from $\mathcal{I}(\gamma)$ as follows: each interaction α in

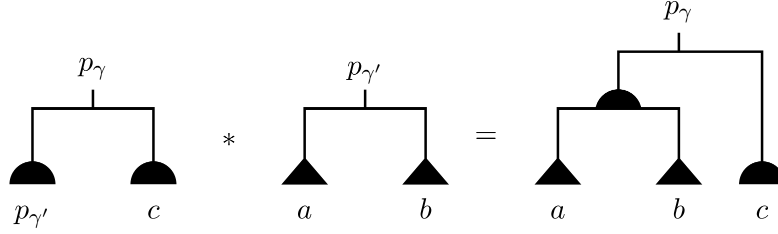


Figure 5.5 – A hierarchical connector

which $p_{\gamma'}$ occurs is replaced by a set of interactions identical to α except that the occurrence of $p_{\gamma'}$ is replaced by an interaction of $\mathcal{I}(\gamma')$.

Definition 5.1.8 (Hierarchical connector) *Let γ and γ' be two connectors such that $p_{\gamma'} \in S_\gamma$. The* hierarchical *connector $\gamma * \gamma'$ resulting from their composition is defined as follows:*

- $S_{\gamma*\gamma'} \triangleq S_\gamma \cup S_{\gamma'} \backslash \{p_{\gamma'}\}$
- $p_{\gamma*\gamma'} \triangleq p_\gamma$
- $\mathcal{I}(\gamma * \gamma') \triangleq \{\alpha \in \mathcal{I}(\gamma) \mid p_{\gamma'} \notin \alpha\} \cup \{\alpha.\alpha' \mid \alpha.p_{\gamma'} \in \mathcal{I}(\gamma) \wedge \alpha' \in \mathcal{I}(\gamma')\}$

Example 5.1.9 *Consider, as in Figure 5.5, γ such that $S_\gamma = \{p_{\gamma'}, c\}$ and $\mathcal{I}(\gamma) = \{\{p_{\gamma'}, c\}\}$; consider also γ' with $S_{\gamma'} = \{a, b\}$ and $\mathcal{I}(\gamma') = \{\{a\}, \{b\}, \{a, b\}\}$. Then $\gamma * \gamma'$ has as support set $\{a, b, c\}$, as exported port p_γ and as interaction set $\{\{a, c\}, \{b, c\}, \{a, b, c\}\}$.*

Connectors whose exported ports and support sets are not related are called *disjoint*. They need not be composed. Note that according to our definition of connector, a connector γ may have as exported port p_γ a port that is also in its support set S_γ. We use this possibility in Chapter 6 to export ports while preserving their name in the I/O contract framework. As a result, $*$ may not be commutative. However, it is always associative.

Definition 5.1.10 (Composition of interaction models) *The operator $*$ is extended to interaction models as follows. The composition $\Gamma_1 * \Gamma_2$ of two interaction models Γ_1 and Γ_2 is obtained from $\Gamma_1 \cup \Gamma_2$ by inductively composing all connectors which are not disjoint.*

Flattening of components, that is, representing them using only atomic components and a single interaction model, is defined as in the semantic framework.

Definition 5.1.11 (Flat component) *A component is called* flat *if it is atomic or of the form $\Gamma\{K_1, \ldots, K_n\}$, where all K_i are atomic components. A component that is not flat is called* hierarchical.

A hierarchical component K is of the form $\Gamma\{K_1, \ldots, K_n\}$ such that at least one K_i is composite. Thus, such a K can be represented as $\Gamma^2\{\Gamma^1\{\mathcal{K}^1\}, \mathcal{K}^2\}$, where $\mathcal{K}^1$ and $\mathcal{K}^2$ are sets of components.

Definition 5.1.12 (Flattening of components) *The* flattened form *of a component K is defined inductively as follows:*

- *if K is a flat component, then its flattened form is equal to K.*
- *otherwise, K is of the form $\Gamma^2\{\Gamma^1\{\mathcal{K}^1\}, \mathcal{K}^2\}$, and then its flattened form is the flattened form of $(\Gamma^2 * \Gamma^1)\{\mathcal{K}^1 \cup \mathcal{K}^2\}$.*

Finally, as in the semantic framework, the following theorem relates the semantics of a composite component and its flattened form.

Theorem 5.1.13 $[\![\Gamma\{\Gamma'\{\mathcal{K}^1\}, \mathcal{K}^2\}]\!]$ *and* $[\![(\Gamma * \Gamma')\{\mathcal{K}^1 \cup \mathcal{K}^2\}]\!]$ *are equivalent in the following sense: there exists a renaming of labels in* $(\Gamma * \Gamma')\{\mathcal{K}^1 \cup \mathcal{K}^2\}$ *as below that makes them bisimilar.*

Let $B_h = [\![\Gamma\{\Gamma'\{\mathcal{K}^1\}, \mathcal{K}^2\}]\!]$ and $B_f = [\![(\Gamma * \Gamma')\{\mathcal{K}^1 \cup \mathcal{K}^2\}]\!]$ be the behavior of respectively the hierarchical component and its flattened form. Formally, the renaming of B_f is an LTS $B_{f'} = (Q_f, q_f^0, \mathcal{I}(\Gamma), \longrightarrow_{f'})$, that is, only the set of labels and the transition relation are modified: the renaming consists in replacing every interaction in $\mathcal{I}(\Gamma')$ by its corresponding exported port — remember that there is only one such port.

The transition relation of $|\Gamma'\{\mathcal{K}^1\}|$ is denoted $\rightsquigarrow_1$. For the sake of clarity, we also use the following notations for $i = 1$ and $i = 2$:

- $\mathcal{K}^i = \{K_k\}_{k=m_i}^{n_i}$ with $m_1 = 1$ and $m_2 = n_1 + 1$
- $Q^i = Q_{m_i} \times \ldots \times Q_{n_i}$
- $\mathcal{P}^i = \bigcup_{k=m_i}^{n_i} \mathcal{P}_k$
- $\mathcal{Q}_i$ denotes $q_{m_i}, \ldots, q_{n_i}$ where $q_k \in Q_k$ for $m_i \leq k \leq n_i$
- $\mathcal{Q}_i \overset{\alpha_i}{\dashrightarrow}_i \mathcal{Q}_i'$ denotes that for every K_k in $\mathcal{K}^i$, either K_k is not involved in α_i (i.e., $\alpha_i \cap \mathcal{P}_k = \emptyset$) and then $q_k = q_k'$, or it is involved in α_i (i.e., $\alpha_i \cap \mathcal{P}_k = \{p_k\}$) and then $q_k \overset{p_k}{\rightsquigarrow}_k q_k'$
- α_i is said to be enabled in $\mathcal{Q}_i$ if there exists $\mathcal{Q}_i'$ such that $\mathcal{Q}_i \overset{\alpha_i}{\dashrightarrow}_i \mathcal{Q}_i'$

Proof. We define $\mathcal{R} \subseteq ((Q^1) \times Q^2) \times (Q^1 \times Q^2)$ as:

$$((\mathcal{Q}_1), \mathcal{Q}_2)\, \mathcal{R}\, (\mathcal{Q}_1', \mathcal{Q}_2') \triangleq \mathcal{Q}_1 = \mathcal{Q}_1' \wedge \mathcal{Q}_2 = \mathcal{Q}_2'$$

We show that this relation is a bisimulation. The initial states are trivially related.

Let us suppose that $((\mathcal{Q}_1), \mathcal{Q}_2) \xrightarrow{\alpha_h}_h ((\mathcal{Q}'_1), \mathcal{Q}'_2)$ and then show that $(\mathcal{Q}_1, \mathcal{Q}_2) \xrightarrow{\alpha_h}_{f'} (\mathcal{Q}'_1, \mathcal{Q}'_2)$. According to the definition of closed semantics, there is no interaction larger than α_h that is enabled in $((\mathcal{Q}_1), \mathcal{Q}_2)$. Besides, α_h contains at most one port of $\mathcal{P}_{\Gamma'}$.

If there is no port of $\mathcal{P}_{\Gamma'}$ in α_k, then $\alpha_f = \alpha_h$ and $(\mathcal{Q}_1, \mathcal{Q}_2) \xrightarrow{\alpha_h}_{f'} (\mathcal{Q}'_1, \mathcal{Q}'_2)$.

Now, suppose that there is one port of $\mathcal{P}_{\Gamma'}$ in α_k. We decompose α_h into $\{p_\gamma\} \cup \alpha_2$, where $\gamma \in \Gamma'$ and $\alpha_2 \in 2^{\mathcal{P}^2}$. Thus, $(\mathcal{Q}_1) \overset{p_\gamma}{\rightsquigarrow}_1 (\mathcal{Q}'_1)$ and $\mathcal{Q}_2 \overset{\alpha_2}{\dashrightarrow}_2 \mathcal{Q}'_2$. This in turn implies, according to the definition of compositional semantics, that there exists $\alpha_1 \in \mathcal{I}(\Gamma')$ such that $\mathcal{Q}_1 \overset{\alpha_1}{\dashrightarrow}_1 \mathcal{Q}'_1$ and $\nexists \alpha'_1 \in \mathcal{I}(\Gamma')$ such that $\alpha_1 \prec_{MP} \alpha'_1$ and α'_1 is enabled in $\mathcal{Q}_1$.

Let us prove that $(\mathcal{Q}_1, \mathcal{Q}_2) \xrightarrow{\alpha_f}_f (\mathcal{Q}'_1, \mathcal{Q}'_2)$ for $\alpha_f = \alpha_1 \cup \alpha_2$. $\mathcal{Q}_1 \overset{\alpha_1}{\dashrightarrow}_1 \mathcal{Q}'_1$ and $\mathcal{Q}_2 \overset{\alpha_2}{\dashrightarrow}_2 \mathcal{Q}'_2$, so we only have to prove that there is no interaction larger than α_f that is enabled in $(\mathcal{Q}_1, \mathcal{Q}_2)$. Suppose that there is an interaction $\alpha' \in \mathcal{I}(\Gamma \circ \Gamma')$ larger than α_f and enabled in $(\mathcal{Q}_1, \mathcal{Q}_2)$. Then α' can be decomposed into $\alpha'_1 \cup \alpha'_2$ with $\alpha'_1 \in \mathcal{I}(\Gamma')$, $\alpha'_2 \in 2^{\mathcal{P}_2}$, so $\alpha_1 \subseteq \alpha'_1$ and $\alpha_2 \subseteq \alpha'_2$. Furthermore, $\mathcal{Q}_1 \overset{\alpha'_1}{\dashrightarrow}_1 \mathcal{Q}'_1$ and $\mathcal{Q}_2 \overset{\alpha'_2}{\dashrightarrow}_2 \mathcal{Q}'_2$.

Maximal progress within $\mathcal{I}(\Gamma')$ implies that α_1 is maximal among the interactions in $\mathcal{I}(\Gamma')$ enabled in $\mathcal{Q}_1$, so $\alpha_1 = \alpha'_1$. Besides, maximal progress within $\mathcal{I}(\Gamma)$ imposes that $\{p_\gamma\} \cup \alpha_2$ is maximal among the interactions in $\mathcal{I}(\Gamma)$ enabled in $(\mathcal{Q}_1), \mathcal{Q}_2$. As $\{p_\gamma\} \cup \alpha'_2$ is in $\mathcal{I}(\Gamma)$ and enabled in $(\mathcal{Q}_1), \mathcal{Q}_2$, we have $\alpha_2 = \alpha'_2$. Thus, $(\mathcal{Q}_1, \mathcal{Q}_2) \xrightarrow{\alpha_f}_f (\mathcal{Q}'_1, \mathcal{Q}'_2)$ which implies, according to the definition of renaming, that $(\mathcal{Q}_1, \mathcal{Q}_2) \xrightarrow{\alpha_h}_{f'} (\mathcal{Q}'_1, \mathcal{Q}'_2)$.

Symmetrically, suppose that $(\mathcal{Q}_1, \mathcal{Q}_2) \xrightarrow{\alpha_h}_{f'} (\mathcal{Q}'_1, \mathcal{Q}'_2)$. By definition of renaming, this implies that α_h can be decomposed into $\alpha_h = \{p_\gamma\} \cup \alpha_2$, where $\gamma \in \Gamma'$ and $\alpha_2 \in 2^{\mathcal{P}^2}$. Furthermore, there exists $\alpha_1 \in \mathcal{I}(\gamma)$ such that $\mathcal{Q}_1 \overset{\alpha_1}{\dashrightarrow}_1 \mathcal{Q}'_1$, $\mathcal{Q}_2 \overset{\alpha_2}{\dashrightarrow}_2 q'_2$ and $\alpha_f = \alpha_1 \cup \alpha_2$ is maximal among the interactions in $\mathcal{I}(\Gamma \circ \Gamma')$ enabled in $(\mathcal{Q}_1, \mathcal{Q}_2)$.

$(\mathcal{Q}_1) \overset{p_\gamma}{\rightsquigarrow}_1 (\mathcal{Q}'_1)$, and $\mathcal{Q}_2 \overset{\alpha_2}{\dashrightarrow}_2 q'_2$, so we only have left to show that there is no interaction larger than α_h that is enabled in $((\mathcal{Q}_1), \mathcal{Q}_2)$. Suppose that there exists such an interaction $\alpha' = \{p_\gamma\} \cup \alpha'_2$. This means that $\alpha_1 \cup \alpha'_2$ is enabled in $(\mathcal{Q}_1, \mathcal{Q}_2)$ and larger than α_f. Because of maximal progress within $\mathcal{I}(\Gamma \circ \Gamma')$, this implies that $\alpha'_2 = \alpha_2$, and thus $\alpha' = \alpha_h$. □

5.2 A second variant: multi-shot BIP

We now propose a second variant that also provides structured interaction and encapsulation but relaxes the conditions of the first variant with respect to composition — thus encompassing multishot semantics — at the cost of not expressing any priority, not even maximal progress. As most of the definitions are common, we write in bold the differences introduced in this setting.

We do not require here that at most one of the ports of a component can be activated at the same time. This implies that components are labeled in this variant with interactions rather than ports, and furthermore we need to use multi-shot for defining our compositional semantics. We make this choice for three reasons: first, we want the possibility to decompose a composite component $\Gamma\{K_1, K_2, K_3\}$ into $\Gamma_2\{\Gamma_1\{K_1, K_2\}, K_3\}$. This is not possible in general if ports of the same component cannot be connected higher in the hierarchy. For example, a connector γ with support set $S_\gamma = \{a, b, c\}$ and interaction set $\mathcal{I}(\gamma) = \{\{a, b\}, \{a, c\}, \{b, c\}\}$ cannot be represented as a hierarchical component without connecting two ports of the same component. A second motivation for this choice is to define interaction models on a set of ports without mentioning the partition of this set according to the interfaces of the components to be composed. Finally, this formalism illustrates how a synchronous semantics could be enforced in this framework.

Definition 5.2.1 (Atomic component) *An* atomic component *on a interface $\mathcal{P}$ is defined by an LTS $K = (Q, q^0, \mathbf{2}^{\mathcal{P}}, \longrightarrow)$.*

Note that here atomic components are labeled by sets of ports rather than ports, because the environment of a component may trigger several of its ports at the same time. The definitions of interaction and connector are those of the first variant. However, the definition of interaction model can be simplified. Indeed, as it is now allowed to connect two ports of the same component, an interaction model can be defined on a set of ports rather than a specific partition of this set. Besides, we also drop the condition that an interaction must be part of at most one connector, because this situation may happen anyway due to hierarchical connectors, as illustrated in Figure 5.6.

Definition 5.2.2 (Interaction model) *An* interaction model Γ *on a support set $\boldsymbol{S_\Gamma}$ is a set of connectors with disjoint exported ports and with support sets included in S_Γ.*

Definition 5.2.3 (Component, Composite component) *A* component *is either an atomic component or it is inductively defined as the composition of a set of components $\{K_i\}_{i=1}^n$ with disjoint interfaces $\{\mathcal{P}_i\}_{i=1}^n$ using an interaction model Γ on $\mathcal{P} = \bigcup_{i=1}^n \mathcal{P}_i$. Such a composition is called a* composite component *on $\mathcal{P}_\Gamma$ and it is denoted $\Gamma\{K_i\}_{i=1}^n$.*

We now focus on semantics. As previously, the compositional semantics refers to exported ports (it is black-box) while the closed semantics distinguishes the different interactions in each connector (white-box). Besides, here the compositional semantics now allows several ports to be fired at the same time. The reason is that these ports may be part of the same connector at a higher level of

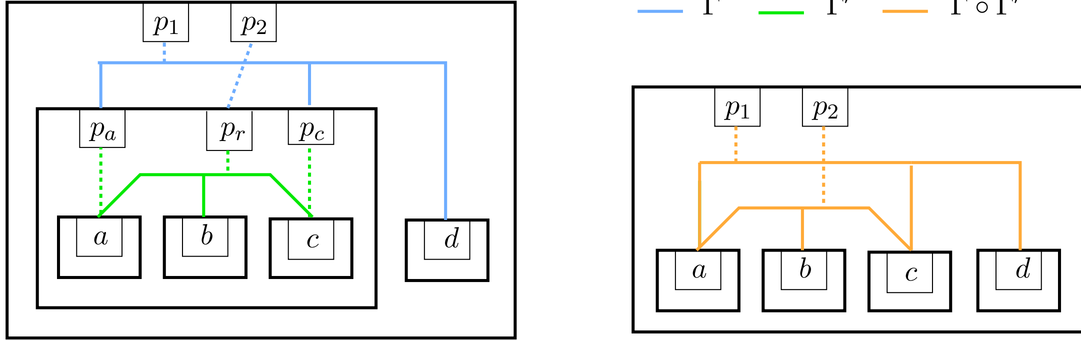


Figure 5.6 – Interaction $\{a, c\}$ may be part of several hierarchical connectors

hierarchy. On the contrary, the closed semantics that we defined allows firing only one connector at a time. Figure 5.7 shows the compositional (top-right) and closed (down-right) semantics of the composite component represented on the left.

Consider a composite component K defined by a set of components $\{K_i\}_{i=1}^n$ and an interaction model Γ with support set $S_\Gamma = \bigcup_{i=1}^n \mathcal{P}_i$. For defining our compositional semantics, we now need to define what a multi-shot interaction is, and in fact we need two such notions, one for representing black-box interactions and one for white-box interactions.

Definition 5.2.4 (Multi-shot interaction) *Given an interaction model* Γ*, a* black-box multi-shot *interaction is of the form* $\{p_{\gamma_1}, \ldots, p_{\gamma_k}\}$*, where all connectors* γ_i *have pairwise disjoint support sets. Each such interaction* $m = \{p_{\gamma_1}, \ldots, p_{\gamma_k}\}$ *is associated with a set of* white-box multi-shot *interactions denoted* $wb(m)$ *and defined as:* $wb(m) = \{\alpha_1 \cup \ldots \cup \alpha_k \mid \forall i \in [1, k],\ \alpha_i \in \mathcal{I}(\gamma_i)\}$.

The set of legal black-box, respectively white-box, multi-shot interactions of Γ is denoted $\mathcal{M}_{bb}(\Gamma)$, respectively $\mathcal{M}_{wb}(\Gamma)$. Multi-shot interactions allow concurrency, as interactions from non-conflicting connectors may be fired simultaneously (unless stated otherwise by the components' behaviors).

Definition 5.2.5 (Compositional semantics) *The* compositional semantics *of* K *is denoted* $|K|$ *and is defined as* $(Q, q^0, \boldsymbol{\mathcal{M}_{bb}(\Gamma)}, \rightsquigarrow)$*, where* $Q = \prod_{i=1}^n Q_i$*,* $q^0 = (q_1^0, \ldots, q_n^0)$ *and given two states* $q^1 = (q_1^1, \ldots, q_n^1)$ *and* $q^2 = (q_1^2, \ldots, q_n^2)$ *in* Q *and a multi-shot interaction* $m \in \mathcal{M}_{bb}(\Gamma)$*,* $q^1 \overset{m}{\rightsquigarrow} q^2$ *if and only if there exists* $\alpha \in wb(m)$ *such that* $\boldsymbol{\forall i, q_i^1 \overset{\alpha_i}{\rightsquigarrow}_i q_i^2}$*, where* $\boldsymbol{\alpha_i = \alpha \cap \mathcal{P}_i}$.

Again, components not involved in the interaction do not move, but this can expressed simply here by using the convention that $\forall q,\ q \xrightarrow{\emptyset} q$. The closed semantics is similar to that of the first variant, except that the condition related to maximal progress has been removed.

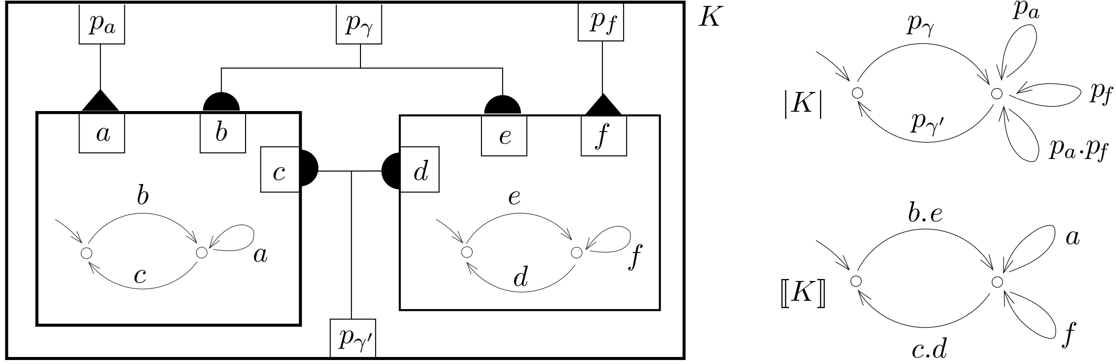


Figure 5.7 – A composite (left) and its compositional (top-right) and closed (bottom-right) semantics

Definition 5.2.6 (Closed semantics) *The* closed semantics *of K is denoted $[\![K]\!]$ and is defined as $(Q, q^0, \mathcal{I}(\Gamma), \longrightarrow)$, where $Q = \prod_{i=1}^{n} Q_i$, $q^0 = (q_1^0, \ldots, q_n^0)$ and $\longrightarrow$ is defined as follows. Given two states $q^1 = (q_1^1, \ldots, q_n^1)$ and $q^2 = (q_1^2, \ldots, q_n^2)$ in Q and an interaction $\alpha \in \mathcal{I}(\Gamma)$, $q^1 \xrightarrow{\alpha} q^2$ if and only if $\boldsymbol{\forall i, q_i^1 \overset{\alpha_i}{\rightsquigarrow}_i q_i^2}$, where $\boldsymbol{\alpha_i = \alpha \cap \mathcal{P}_i}$.*

Composition of interaction models is the same as in the first variant. Finally, we have again the theorem relating the semantics of a composite component and its flattened form.

Theorem 5.2.7 $[\![\Gamma\{\Gamma'\{\mathcal{K}^1\}, \mathcal{K}^2\}]\!]$ *and* $[\![(\Gamma * \Gamma')\{\mathcal{K}^1 \cup \mathcal{K}^2\}]\!]$ *are equivalent in the following sense: there exists a renaming of labels in* $(\Gamma * \Gamma')\{\mathcal{K}^1 \cup \mathcal{K}^2\}$ *as below that makes them bisimilar.*

For simplifying notation, we suppose that $\mathcal{K}^1 = \{K_1, K_2\}$ and $\mathcal{K}^2 = \{K_3\}$ and we denote B_h the behavior of the hierarchical component, i.e., $[\![\Gamma\{\Gamma'\{K_1, K_2\}, K_3\}]\!]$ and B_f the behavior of its flattened form $[\![(\Gamma * \Gamma')\{K_1, K_2, K_3\}]\!]$. The transition relation of $|\Gamma'\{K_1, K_2\}|$ is denoted $\rightsquigarrow_{1,2}$.

As before, the renaming consists in replacing interactions of $\mathcal{I}(\Gamma')$ by the corresponding exported ports. However, due to the fact that an interaction may be part of several connectors, possibly through hierarchical connectors, some problems may arise in this renaming. Figure 5.6 illustrates this: depending on the context, $\alpha_f = \{a, c\}$ of the flattened component may be renamed into either $\alpha_h = \{p_a, p_c\}$ or $\alpha'_h = \{p_r\}$.

Formally, the renaming of B_f is an LTS $B_{f'} = (Q_f, q_f^0, \mathcal{I}(\Gamma), \longrightarrow_{f'})$, that is, only the set of labels and the transition relation are modified. The new transition relation is defined as follows. $(q_1, q_2, q_3) \xrightarrow{\alpha_h}_{f'} (q_1, q_2, q_3)$ if and only if:

1. $(q_1, q_2, q_3) \xrightarrow{\alpha_f}_{f} (q_1, q_2, q_3)$; we decompose α_f into $\alpha_f = \alpha_1 \cup \alpha_2 \cup \alpha_3$ with $\alpha_i \in \mathcal{P}_i$
2. α_h can be decomposed into $m \cup \alpha_3$ such that $m \in \mathcal{M}_{bb}(\Gamma')$ and $\alpha_1 \cup \alpha_2 \in wb(m)$

Proof. We define $\mathcal{R} \subseteq ((Q_1 \times Q_2) \times Q_3) \times (Q_1 \times Q_2 \times Q_3)$ as:

$$((q_1, q_2), q_3)\, \mathcal{R}\, (q'_1, q'_2, q'_3) \triangleq q_1 = q'_1 \wedge q_2 = q'_2 \wedge q_3 = q'_3$$

We show that this relation is a bisimulation. The initial states are trivially related.

Suppose that $((q_1, q_2), q_3) \xrightarrow{\alpha_h}_h ((q'_1, q'_2), q'_3)$. Let us show that $(q_1, q_2, q_3) \xrightarrow{\alpha_h}_{f'} (q'_1, q'_2, q'_3)$. According to the definition of closed semantics, α_h can be decomposed as $\alpha_h = m \cup \alpha_3$ where $m \in \mathcal{M}_{bb}(\Gamma')$, $\alpha_3 \in \mathcal{P}_3$, and furthermore $(q_1, q_2) \overset{m}{\rightsquigarrow}_{1,2} (q'_1, q'_2)$ and $q_3 \overset{\alpha_3}{\rightsquigarrow}_3 q'_3$. This in turn implies, according to the definition of compositional semantics, that there exists $\alpha_1 \in \mathcal{P}_1$ and $\alpha_2 \in \mathcal{P}_2$ such that $\alpha_1 \cup \alpha_2 \in wb(m)$, $q_1 \overset{\alpha_1}{\rightsquigarrow}_1 q'_1$ and $q_2 \overset{\alpha_2}{\rightsquigarrow}_2 q'_2$.

As $q_i \overset{\alpha_i}{\rightsquigarrow}_i q'_i$ for $i \in [1, 3]$, we have $(q_1, q_2, q_3) \xrightarrow{\alpha_f}_f (q'_1, q'_2, q'_3)$ for $\alpha_f = \alpha_1 \cup \alpha_2 \cup \alpha_3$. Thus, according to the definition of renaming and of α_f, this implies that $(q_1, q_2, q_3) \xrightarrow{\alpha_h}_{f'} (q'_1, q'_2, q'_3)$.

Symmetrically, suppose that $(q_1, q_2, q_3) \xrightarrow{\alpha_h}_{f'} (q'_1, q'_2, q'_3)$. By definition of renaming, this implies that α_h can be decomposed into $m \cup \alpha_3$ with $m \in \mathcal{M}_{bb}(\Gamma')$ and $\alpha_3 \in \mathcal{P}_3$ and furthermore that there exists $\alpha_1 \in \mathcal{P}_1$ and $\alpha_2 \in \mathcal{P}_2$ such that $\alpha_1 \cup \alpha_2 \in wb(m)$ and $(q_1, q_2, q_3) \xrightarrow{\alpha_f}_f (q'_1, q'_2, q'_3)$ for $\alpha_f = \alpha_1 \cup \alpha_2 \cup \alpha_3$, which implies that $q_i \overset{\alpha_i}{\rightsquigarrow}_i q'_i$ for $i \in [1, 3]$. We obtain from this that $(q_1, q_2) \overset{m}{\rightsquigarrow}_{1,2} (q'_1, q'_2)$ and then $((q_1, q_2), q_3) \xrightarrow{\alpha_h}_h ((q'_1, q'_2), q'_3)$. □

Data transformation will not be dealt with in this thesis, although we have also studied frameworks handling data. The interested reader is referred to [BHQG10b, BHQG10c] for a framework inspired by BIP an handling data transformation.

5.3 Projection

We discuss here a simple way of handling hiding of ports, by using the convention that $\forall q,\ q \xrightarrow{\emptyset} q$. Indeed, such a convention makes sense in a context without variables, as a transition labeled by $\emptyset$ from q to q is really equivalent to an absence of progress. Note that as a result, an LTS is in deadlock in a state q if its only transition enabled in q is $q \xrightarrow{\emptyset} q$.

If we adopt this convention, then we can modify the semantics provided in the previous sections as follows: $\emptyset$ can be defined as an interaction, and components that do not take part in an interaction may still perform an internal computation labeled by $\emptyset$. Then, hiding a connector is achieved simply by defining its exported port as $\emptyset$. Hiding a port p is achieved by defining a connector with support set $\{p\}$, interaction set $\{\{p\}\}$ and exported port $\emptyset$. Hence the following definition of projection.

Definition 5.3.1 (Projection) *The* projection *of a component K defined on an interface $\mathcal{P}$ onto a subset of its ports $\mathcal{P}' \subseteq \mathcal{P}$, denoted $\Pi_{\mathcal{P}'}(K)$ is $|gl_{\Pi}\{\mathcal{K}\}|$ where:*

$$
\begin{aligned}
gl_{\Pi} = \{\gamma_p \mid p \in \mathcal{P}' \wedge S_{\gamma_p} = \{p\}, p_{\gamma_p} = \{p\}, \mathcal{I}(\gamma_p) = \{\{p\}\}\} \\
\cup \{\gamma_p \mid p \in \mathcal{P} \backslash \mathcal{P}' \wedge S_{\gamma_p} = \{p\}, p_{\gamma_p} = \{\emptyset\}, \mathcal{I}(\gamma_p) = \{\{p\}\}\}
\end{aligned}
$$

It is also useful in some contexts (for example in the L0 framework) to define an inverse projection that augments the set of ports of a component. It is also possible to define it as a glue

Definition 5.3.2 (Inverse projection) *The* inverse projection *of a component K defined on an interface $\mathcal{P}$ onto a superset of its ports $\mathcal{P}'$, denoted $\Pi^{-1}_{\mathcal{P}'}(K)$ is $|gl^{-1}_{\Pi}\{\mathcal{K}\}|$ where:*

$$
\begin{aligned}
gl^{-1}_{\Pi} = \{\gamma_p \mid p \in \mathcal{P} \wedge S_{\gamma_p} = \{p\}, p_{\gamma_p} = \{p\}, \mathcal{I}(\gamma_p) = \{\{p\}\}\} \\
\cup \{\gamma_p \mid p \in \mathcal{P}' \backslash \mathcal{P} \wedge S_{\gamma_p} = \{\emptyset\}, p_{\gamma_p} = \{p\}, \mathcal{I}(\gamma_p) = \{\emptyset\}\}
\end{aligned}
$$

Chapter 6

Application to I/O automata and to the SPEEDS project

In this chapter, we present two distinct applications involving the variants of BIP presented in the previous chapter. The first application is the representation of I/O interface automata [LNW06] as a contract framework using BIP. The second application is the definition of two contract frameworks in the context of the SPEEDS project, called L0 and L1, for which we provide a methodology making it possible to use them jointly.

6.1 Encoding of interface I/O automata

Input/Output (I/O) automata [Lyn96] are a formalism of choice to describe and analyze distributed systems. I/O automata interact with each other by synchronizing an output action in one I/O automaton with a corresponding input action in another interacting I/O automaton. Outputs are typically used to represent method calls, data transmission, return of method calls, exceptions raised during execution etc. Symmetrically, inputs usually model method invocations, receiving of data, and return location of a function call.

One difficulty when dealing with I/O automata is the following: in general, in the concrete system represented by an I/O automaton, whenever an output is possible, the data are sent whether the corresponding input is possible or not. In the latter case, the data are lost. However, composition of I/O automata is by strong synchronization. This means that in the abstract setting, an output may take place *only* if the corresponding input is also possible. To solve this discrepancy, one usually assumes input-completeness (i.e., all inputs are always possible).

Larsen et al., inspired by [dAH01a], have proposed in [LNW06] an interface theory dealing with this issue, based on the use of *I/O interface automata*. Their approach consists in defining an interface as a pair assumption/guarantee where both assumptions and guarantees are I/O automata. A syntactic notion of *illegal* state is introduced to capture situations when an output is offered without the corresponding input being possible.

In this section, we show that I/O interface automata [LNW06] can be nicely represented as a contract framework using BIP. We build on the variant BIP with maximal progress but restrict ourselves to a subset of the BIP glues defined in Section 5.1. In particular, the distinction between inputs and outputs is used only here to determine which component triggers an interaction (the one proposing the output), and behaviors are represented by LTS instead of I/O automata. We are then able to verify that no output is lost like any other safety property, by checking that no output occurs without the corresponding inputs also being present. We then provide alternative and simpler proofs for two theorems presented at the beginning of [LNW06], which are only based on abstract concepts such as transitivity and reflexivity of refinement under context, soundness of circular reasoning etc.

6.1.1 The I/O contract framework

The I/O component framework that we define is based on the first variant BIP framework presented in Chapter 5, namely BIP with maximal progress. Note that it is essential here to ensure maximal progress. Indeed, we do not define connectors as rendezvous between an output and the corresponding input, but as broadcasts where the input is not necessary for the interaction to take place. In this context, if two interactions are possible, e.g., a single output $\{p!\}$ or the same output along with its corresponding input $\{p!, p?\}$, then the latter should be preferred.

In the context of I/O automata, the set of ports $Ports$ is partitioned into *input* ports $Ports^?$ and *output* ports $Ports^!$ such that each port appears in both sets. In this section, ! (respectively ?) is used to denote that a port is an output (respectively an input) or that a set of ports contains only outputs (resp. inputs). Inputs and outputs are used only to define the connectors. We use the notations and definitions introduced for BIP with maximal progress in Section 5.1.

Definition 6.1.1 (Component framework) *The I/O component framework is defined as follows:*

- $\mathcal{K}$ *is a set of (possibly composite) components.*
- $K_1 \cong K_2 \triangleq |K_1|^{det} = |K_2|^{det}$ [1]. *It is obviously an equivalence relation.*

1. Remember that A^{det} denotes the determinization of A for any LTS A; see Definition 1.1.6.

- *Given a set of ports $\mathcal{P} = \mathcal{P}^? \cup \mathcal{P}^!$, we define an interaction model on $\mathcal{P}$ (and the corresponding glue) by the following set of connectors Γ. For each $p \in \mathcal{P}$:*
  - *if $p?$ and $p!$ are both in $\mathcal{P}$, then we define γ_p with support set $\{p!, p?\}$, exported port $\emptyset$ and interaction set $\{\{p!\}, \{p?, p!\}\}$.*
  - *if $p?$ is in $\mathcal{P}$ but not $p!$, then we define γ_p with support set $\{p?\}$, exported port $p?$ and interaction set $\{\{p?\}\}$.*
  - *similarly, if only $p!$ is in $\mathcal{P}$, then we define γ_p with support set $\{p!\}$, exported port $p!$ and interaction set $\{\{p!\}\}$.*
- *$\circ$ is the composition of such interaction models as defined in Section 5.1. Note that if Γ and Γ' are of the form described in the previous item, then so is $\Gamma \circ \Gamma'$.*

Note that there is exactly one interaction model per support set. Note also that unlike in the more general framework BIP with maximal progress, it is not necessary here to define the support set of a glue as a partition.

Definition 6.1.2 (Conformance) $K_1 \preccurlyeq K_2 \triangleq Tr(\llbracket K_1 \rrbracket) \subseteq Tr(\llbracket K_2 \rrbracket)$

For I/O interface automata, refinement under context is the usual one. As exactly one component has control over an interaction (the component whose port is the output), this definition is sufficient to ensure soundness of circular reasoning.

Definition 6.1.3 (Refinement under context) *Refinement under context is defined as follows:* $K_1 \sqsubseteq_{E,\Gamma} K_2 \triangleq Tr(\llbracket \Gamma\{K_1, E\} \rrbracket) \subseteq Tr(\llbracket \Gamma\{K_2, E\} \rrbracket)$

Interestingly, we know something more about an interaction model Γ that is used to define a closed system: it must be such that all inputs have a corresponding output and vice versa. That is, it consists only of connectors of the form $\{\{p!\}, \{p?, p!\}\}$.

6.1.2 Coherence conditions

Here is a brief summary of the conditions to be checked:

- ▲ composition of glues and equivalence of components, that is, $\circ$ and $\cong$
  Formally: $\Gamma_1\{\Gamma_2\{\mathcal{K}^1\}, \mathcal{K}^2\} \cong (\Gamma_1 \circ \Gamma_2)\{\mathcal{K}^1 \cup \mathcal{K}^2\}$
- ■ conformance and refinement under context, i.e., $\preccurlyeq$ and $\{\sqsubseteq\}_{\omega\in\Omega}$
  Formally: if $K_1 \sqsubseteq_{E,\Gamma} K_2$, then $\Gamma\{K_1, E\} \preccurlyeq \Gamma\{K_2, E\}$
  This is trivially obtained from the definitions of $\sqsubseteq$ and $\preccurlyeq$.

- refinement under context and equivalence of components, i.e., $\{\sqsubseteq\}_{\omega\in\Omega}$ and $\cong$
  Formally: if $K_1 \cong K_1'$ then $K_1 \sqsubseteq_{E,\Gamma} K_2$ if and only if $K_1' \sqsubseteq_{E,\Gamma} K_2$
  This comes directly from the fact that $|K_1| = |K_1'|$ implies that $[\![\Gamma\{K_1, E\}]\!] = [\![\Gamma\{K_2, E\}]\!]$.
- Is refinement under context preserved by composition?
- Is circular reasoning sound?
- ▲ Given a glue Γ on $\mathcal{P}$ (i.e. $S_\Gamma = \mathcal{P}$) and Γ_1 on $\mathcal{P}_1 \subseteq \mathcal{P}$, can we determine whether there exists Γ_2 such that $\Gamma = \Gamma_1 \circ \Gamma_2$?

Coherence between $\circ$ and $\cong$

Proof. We want to show that $|\Gamma_1\{\Gamma_2\{\mathcal{K}^1\}, \mathcal{K}^2\}|^{det} = |(\Gamma_1 \circ \Gamma_2)\{\mathcal{K}^1 \cup \mathcal{K}^2\}|^{det}$. Theorem 5.1.13 says that $[\![\Gamma_1\{\Gamma_2\{\mathcal{K}^1\}, \mathcal{K}^2\}]\!]$ and $[\![(\Gamma_1 \circ \Gamma_2)\{\mathcal{K}^1 \cup \mathcal{K}^2\}]\!]$ are bisimilar after hiding data transfer within $\mathcal{K}_1$. By definition of closed and open semantics, this implies that $|\Gamma_1\{\Gamma_2\{\mathcal{K}^1\}, \mathcal{K}^2\}|$ and $|(\Gamma_1 \circ \Gamma_2)\{\mathcal{K}^1 \cup \mathcal{K}^2\}|$ are bisimilar. Hence the result. □

Preservation of refinement by composition

Proof. Refinement under context is preserved by composition.
- Let (E, Γ) be a context for an interface $\mathcal{P}$ and Γ_E, E_1, E_2 such that $E = \Gamma_E\{E_1, E_2\}$ and $\Gamma \circ \Gamma_E = \Gamma_2 \circ \Gamma_1$, where Γ_1 is defined on $\mathcal{P}_{K_1} \cup \mathcal{P}_{E_1}$ and Γ_2 is defined on $\mathcal{P}_{\Gamma_1} \cup \mathcal{P}_{E_2}$.
- By definition, $K_1 \sqsubseteq_{E,\Gamma} K_2 \triangleq \mathit{Tr}([\![\Gamma\{K_1, E\}]\!]) \subseteq \mathit{Tr}([\![\Gamma\{K_2, E\}]\!])$.
- Besides, according to the definition of E etc., we have:

$$\begin{aligned}\Gamma\{K_1, E\} &= \Gamma\{K_1, \Gamma_E\{E_1, E_2\}\} \\ &= (\Gamma \circ \Gamma_E)\{K_1, E_1, E_2\} \\ &= (\Gamma_2 \circ \Gamma_1)\{K_1, E_1, E_2\} \\ &= \Gamma_2\{\Gamma_1\{K_1, E_1\}, E_2\}\end{aligned}$$

- Similarly, we obtain that $\Gamma\{K_2, E\} = \Gamma_2\{\Gamma_1\{K_2, E_1\}, E_2\}$.
- Hence: $K_1 \sqsubseteq_{E,\Gamma} K_2 \iff \mathit{Tr}([\![\Gamma_2\{\Gamma_1\{K_1, E_1\}, E_2\}]\!]) \subseteq \mathit{Tr}([\![\Gamma_2\{\Gamma_1\{K_2, E_1\}, E_2\}]\!])$.
- The right part of this equivalence, by definition of refinement under context, is equivalent to: $\Gamma_1\{K_1, E_1\} \sqsubseteq_{E_2,\Gamma_2} \Gamma_1\{K_2, E_1\}$. □

This proof is a direct consequence of properties of composition of BIP glues. We now prove and discuss soundness of circular reasoning.

Soundness of circular reasoning

Proof. Let us suppose that $K \sqsubseteq_{A,\Gamma} G$ and $E \sqsubseteq_{G,\Gamma} A$ and show that $K \sqsubseteq_{E,\Gamma} G$.
- That is, suppose (1) $Tr([\![\Gamma\{K,A\}]\!]) \subseteq Tr([\![\Gamma\{G,A\}]\!])$ and (2) $Tr([\![\Gamma\{G,E\}]\!]) \subseteq Tr([\![\Gamma\{G,A\}]\!])$. We have to prove that $Tr([\![\Gamma\{K,E\}]\!]) \subseteq Tr([\![\Gamma\{G,E\}]\!])$.
- As $\{\sqsubseteq\}_{\omega\in\Omega}$ is consistent with $\cong$, we may safely suppose that K, A, G and E are deterministic, and as a consequence so are $[\![\Gamma\{K,A\}]\!]$, $[\![\Gamma\{G,A\}]\!]$, $[\![\Gamma\{G,E\}]\!]$ and $[\![\Gamma\{K,E\}]\!]$.
- To simplify notations, we omit in the rest of the proof the brackets $[\![]\!]$.
- We prove by induction on the length of the traces that any trace of $\Gamma\{K,E\}$ is a trace of $\Gamma\{K,A\}$, of $\Gamma\{G,A\}$ and of $\Gamma\{G,E\}$.
- The property trivially holds for traces of length 0.
- Suppose now that this property holds for traces of length l. Let σ be a trace of $\Gamma\{K,E\}$ of length $l+1$. We decompose σ as $\sigma'.\,\alpha$.
- Denote (q_K,q_E) the state of $\Gamma\{K,E\}$ reached after firing σ' from the initial state. As $\Gamma\{K,E\}$ is deterministic, this state is unique (see Property 1.1.5).
- As σ' is a trace of $\Gamma\{K,E\}$ of length l, it is also a trace of $\Gamma\{K,A\}$, $\Gamma\{G,A\}$ and $\Gamma\{G,E\}$. Denote respectively (q_K,q_A), (q_G,q_A) and (q_G,q_E) the states reached after firing σ' from the initial state.
- As already mentioned, Γ defines a closed system and as a result it is specific: all inputs have a corresponding output and vice versa. That is, Γ consists only of connectors of the form $\{\{p!\},\{p?,p!\}\}$ with $p? \in \mathcal{P}_K$ and $p! \in \mathcal{P}_E$ — we say that p is *controlled* by E — or $p? \in \mathcal{P}_E$ and $p! \in \mathcal{P}_K$ — where p is controlled by K.
- Suppose that p is controlled by K in $\Gamma\{K,E\}$ — and thus also in $\Gamma\{K,A\}$. Thus, α is either $\{p!\}$ or $\{p?,p!\}$ with $p! \in \mathcal{P}_K$ and $p? \in \mathcal{P}_E$. This implies that $p!$ is enabled in q_K.
- Suppose first that $\alpha = \{p?,p!\}$.
This implies that $p?$ is enabled in q_E.
Because of (1), $p!$ is enabled in q_G (whether $p?$ is enabled in q_A or not does not matter here). Hence, α is enabled in (q_G,q_E), so σ is a trace of $\Gamma\{G,E\}$.
As σ is a trace of $\Gamma\{G,E\}$, we get from (2) that α is enabled in (q_G,q_A). This in turn implies that $p?$ is enabled in q_A and thus α is also enabled in (q_K,q_A).
Thus, if p is controlled by K in $\Gamma\{K,E\}$ and if $\alpha = \{p?,p!\}$, then σ is a trace of $\Gamma\{G,E\}$, $\Gamma\{K,A\}$ and $\Gamma\{G,A\}$.
- Suppose now that $\alpha = \{p!\}$ (p is still controlled by K).
Because of maximal progress, this implies that $p?$ is *not* enabled in q_E.
As before, because of (1), $p!$ is enabled in q_G (whether $p?$ is enabled in q_A or not does not matter

here). Hence, α is enabled in (q_G, q_E), so σ is a trace of $\Gamma\{G, E\}$.
As σ is a trace of $\Gamma\{G, E\}$, we get from (2) that α is enabled in (q_G, q_A). This in turn implies that $p?$ is *not* enabled in q_A and thus α is also enabled in (q_K, q_A).
- Thus, if p is controlled by K in $\Gamma\{K, E\}$ (whether $\alpha = \{p!\}$ or $\alpha = \{p?, p!\}$), then σ is a also trace of $\Gamma\{G, E\}$, $\Gamma\{K, A\}$ and $\Gamma\{G, A\}$.
- Suppose now that p is controlled by E in $\Gamma\{K, E\}$ and then also in $\Gamma\{G, E\}$. This implies that $p!$ is enabled in q_E.
- Again, suppose first that $\alpha = \{p?, p!\}$.
This implies that $p?$ is enabled in q_K.
Because of (2), $p!$ is enabled in q_A (whether $p?$ is enabled in q_G or not does not matter here). Hence, α is enabled in (q_K, q_A), so σ is a trace of $\Gamma\{K, A\}$.
As σ is a trace of $\Gamma\{K, A\}$, we get from (1) that α is enabled in (q_G, q_A). This in turn implies that $p?$ is enabled in q_G and thus α is also enabled in (q_G, q_E).
Suppose now that $\alpha = \{p!\}$ (p is still controlled by E).
Because of maximal progress, this implies that $p?$ is *not* enabled in q_K.
As before, because of (2), $p!$ is enabled in q_A (whether $p?$ is enabled in q_G or not does not matter here). Hence, α is enabled in (q_K, q_A), so σ is a trace of $\Gamma\{K, A\}$.
As σ is a trace of $\Gamma\{K, A\}$, we get from (1) that α is enabled in (q_G, q_A). This in turn implies that $p?$ is *not* enabled in q_G and thus α is also enabled in (q_G, q_E).
- Thus, if p is controlled by E in $\Gamma\{K, E\}$ (whether $\alpha = \{p!\}$ or $\alpha = \{p?, p!\}$), then σ is a also trace of $\Gamma\{G, E\}$, $\Gamma\{K, A\}$ and $\Gamma\{G, A\}$.
- Hence the result. □

Soundness of circular reasoning in this case is ensured by the asymmetry in the interaction: exactly one component has control over an interaction, which implies that knowing what happens in the global system is sufficient to determine what may be possible for each component. In particular, if an interaction is forbidden, then only the trigger of this interaction must forbid it, because the synchrons will anyway be blocked by the absence of the trigger.

6.1.3 Using the I/O contract framework

To illustrate the usefulness of reasoning at this level of abstraction, we present simple proofs for two theorems of [LNW06] which are applications of circular reasoning. As they are completely independent of the I/O framework and only rely on compositionality and soundness of circular reasoning, they hold in any contract-based verification framework ensuring these two properties. These theorems

are presented in the context of a dominance problem where two low-level contracts (A_1, Γ_1, G_1) and (A_2, Γ_2, G_2) must dominate a top-level contract (A, Γ, G) with $G = \Gamma_{1,2}\{G_1, G_2\}$.

The first theorem states that it is sufficient to discharge assumptions A_i in the abstract context to be sure that these assumptions are indeed discharged by any actual context satisfying its own contract.

Theorem 6.1.4

$$\Gamma_{A,2}\{A, G_2\} \sqsubseteq_{G_1,\Gamma_1} A_1 \wedge \Gamma_{A,1}\{A, G_1\} \sqsubseteq_{G_2,\Gamma_2} A_2$$

is equivalent to

$$\forall K_1, K_2, K_1 \sqsubseteq_{A_1,\Gamma_1} G_1 \wedge K_2 \sqsubseteq_{A_2,\Gamma_2} G_2 \implies \Gamma_{A,2}\{A, K_2\} \sqsubseteq_{K_1,\Gamma_1} A_1 \wedge \Gamma_{A,1}\{A, K_1\} \sqsubseteq_{K_2,\Gamma_2} A_2$$

Proof. The right-to-left implication is trivial since $G_1 \sqsubseteq_{A_1,\Gamma_1} G_1$ and $G_2 \sqsubseteq_{A_2,\Gamma_2} G_2$ (because for any A and Γ the relation $\sqsubseteq_{A,\Gamma}$ is reflexive). Now let us fix K_1 and K_2 and suppose that the following properties hold.

$$\begin{cases} \Gamma_{A,2}\{A, G_2\} \sqsubseteq_{G_1,\Gamma_1} A_1 & (1) \\ \Gamma_{A,1}\{A, G_1\} \sqsubseteq_{G_2,\Gamma_2} A_2 & (2) \\ K_1 \sqsubseteq_{A_1,\Gamma_1} G_1 & (3) \\ K_2 \sqsubseteq_{A_2,\Gamma_2} G_2 & (4) \end{cases}$$

Our goal is to prove that $\Gamma_{A,2}\{A, K_2\} \sqsubseteq_{K_1,\Gamma_1} A_1$ and $\Gamma_{A,1}\{A, K_1\} \sqsubseteq_{K_2,\Gamma_2} A_2$. By applying circular reasoning (**CR**) to (3) and (1), and to (4) and (2), we get the following:

$$\begin{cases} K_1 \sqsubseteq_{\Gamma_{A,2}\{A,G_2\},\Gamma_1} G_1 & (5) \\ K_2 \sqsubseteq_{\Gamma_{A,1}\{A,G_1\},\Gamma_2} G_2 & (6) \end{cases}$$

Then, by preservation of refinement by composition (**CMP**), we get from (5) and (6):

$$\begin{cases} \Gamma_{A,1}\{A, K_1\} \sqsubseteq_{G_2,\Gamma_2} \Gamma_{A,1}\{A, G_1\} & (7) \\ \Gamma_{A,2}\{A, K_2\} \sqsubseteq_{G_1,\Gamma_1} \Gamma_{A,2}\{A, G_2\} & (8) \end{cases}$$

We now apply transitivity of $\sqsubseteq_{G_2,\Gamma_2}$ (resp. $\sqsubseteq_{G_1,\Gamma_1}$) to (7) and (2) (resp. (8) and (1)):

$$\begin{cases} \Gamma_{A,1}\{A, K_1\} \sqsubseteq_{G_2,\Gamma_2} A_2 & (9) \\ \Gamma_{A,2}\{A, K_2\} \sqsubseteq_{G_1,\Gamma_1} A_1 & (10) \end{cases}$$

Finally, by applying circular reasoning (**CR**) to (9) and (4), and to (10) and (3), we get the result:

$$\begin{cases} \Gamma_{A,1}\{A, K_1\} \sqsubseteq_{K_2,\Gamma_2} A_2 \\ \Gamma_{A,2}\{A, K_2\} \sqsubseteq_{K_1,\Gamma_1} A_1 \end{cases} \qquad \square$$

The second theorem is a special case of the sufficient condition for dominance expressed in Theorem 2.3.5. We give here the proof for this particular case to emphasize the relevant properties.

Theorem 6.1.5

$$\Gamma_{A,2}\{A, G_2\} \sqsubseteq_{G_1,\Gamma_1} A_1 \wedge \Gamma_{A,1}\{A, G_1\} \sqsubseteq_{G_2,\Gamma_2} A_2$$

implies

$$\forall K_1, K_2, K_1 \sqsubseteq_{A_1,\Gamma_1} G_1 \wedge K_2 \sqsubseteq_{A_2,\Gamma_2} G_2 \implies \Gamma_{1,2}\{K_1, K_2\} \sqsubseteq_{A,\Gamma} \Gamma_{1,2}\{G_1, G_2\}$$

Proof. let us fix K_1 and K_2 and suppose that the following properties hold.

$$\begin{cases} \Gamma_{A,2}\{A, G_2\} \sqsubseteq_{G_1,\Gamma_1} A_1 & (1) \\ \Gamma_{A,1}\{A, G_1\} \sqsubseteq_{G_2,\Gamma_2} A_2 & (2) \\ K_1 \sqsubseteq_{A_1,\Gamma_1} G_1 & (3) \\ K_2 \sqsubseteq_{A_2,\Gamma_2} G_2 & (4) \end{cases}$$

Our goal is to prove that $\Gamma_{1,2}\{K_1, K_2\} \sqsubseteq_{A,\Gamma} \Gamma_{1,2}\{G_1, G_2\}$.
The first two steps of this proof already appeared in the previous proof.
By applying circular reasoning (**CR**) to (3) and (1), we get:

$$K_1 \sqsubseteq_{\Gamma_{A,2}\{A,G_2\},\Gamma_1} G_1 \qquad (5)$$

Then, by preservation of refinement by composition (**CMP**), we obtain:

$$\Gamma_{A,1}\{A, K_1\} \sqsubseteq_{G_2,\Gamma_2} \Gamma_{A,1}\{A, G_1\} \qquad (6)$$

Next, we apply transitivity of $\sqsubseteq_{G_2,\Gamma_2}$ to (6) and (2):

$$\Gamma_{A,1}\{A, K_1\} \sqsubseteq_{G_2,\Gamma_2} A_2 \qquad (7)$$

We now apply (**CR**) to (4) and (7):

$$K_2 \sqsubseteq_{\Gamma_{A,1}\{A,K_1\},\Gamma_2} G_2 \qquad (8)$$

We apply (**CMP**) to (8) and to (5):

$$\begin{cases} \Gamma_{1,2}\{K_1, K_2\} \sqsubseteq_{A,\Gamma} \Gamma_{1,2}\{K_1, G_2\} & (9) \\ \Gamma_{1,2}\{K_1, G_2\} \sqsubseteq_{A,\Gamma} \Gamma_{1,2}\{G_1, G_2\} & (10) \end{cases}$$

Finally, by transitivity of $\sqsubseteq_{A,\Gamma}$ applied to (9) and (10), we obtain the result:

$$\Gamma_{1,2}\{K_1, K_2\} \sqsubseteq_{A,\Gamma} \Gamma_{1,2}\{G_1, G_2\} \qquad \square$$

Note that this theorem expresses the sufficient condition for dominance provided in Theorem 2.3.5 in the case where there are only two components with contracts (A_1, Γ_1, G_1) and (A_2, Γ_2, G_2), and where the global contract is $(A, \Gamma, \Gamma_{1,2}\{G_1, G_2\})$. The left-hand side in the theorem above and $\Gamma_{1,2}\{G_1, G_2\} \sqsubseteq_{A,\Gamma} \Gamma_{1,2}\{G_1, G_2\}$ (implied by reflexivity of $\sqsubseteq_{A,\Gamma}$) correspond exactly to the sufficient condition of 2.3.5. Hence, $\{(A_1, \Gamma_1, G_1), (A_2, \Gamma_2, G_2)\}$ dominates $(A, \Gamma, \Gamma_{1,2}\{G_1, G_2\})$, which is exactly the right-hand side of Theorem 6.1.5.

The proofs presented here do not depend on the formalism of I/O automata, which makes them shorter and easier to understand than in [LNW06].

6.2 The SPEEDS project

We show now how the general methodology presented in Chapter 2 has been applied in the context of the SPEEDS project [SPE]. In this project, a modeling framework called HRC — standing for Heterogeneous Rich Components — has been defined to provide system designers with an environment for contract-based design. HRC offers the possibility to define hierarchical components where interactions and data exchange are defined by explicit connectors between ports which define the component's interface. Components are associated with behaviors (implementations) and contracts are represented as a pair of behaviors (that is, properties) expressing an assumption on the environment and a guarantee, that is a property that the component's implementation must realize in any environment obeying the constraint imposed by the assumption.

In HRC, we choose to represent behaviors on an interface (that is represented as a set of ports $\mathcal{P}$) as labeled transition systems, which are sufficiently enriched to represent hybrid and stochastic behaviors as well as implementations realizing complex data transformations. To simplify the presentation, we restrict ourselves here to abstract transition systems without data nor any other extension.

Definition 6.2.1 (LTS as behaviors) *A* behavior on a set of ports $\mathcal{P}$ *is a labeled transition system as*

in Definition 1.1.1 and such that its set of labels is $2^{\mathcal{P}}$.

For hierarchical model composition, there exist several models defining different levels of abstraction: they range from low-level semantic composition (called L0) to user level composition which generalizes the composition primitives existing in commercial design tools. We discuss here two layers: L0 which we already mentioned, and L1 which offers a richer set of glues.

In L0, connectors are simple name matchings which connect an output port to one (or more) input ports, possibly in a hierarchical fashion. The composition semantics is synchronous, that is, a transition of a composed system involves all components in a globally maximal interaction. L1-connectors are more complex; they are inspired by BIP connectors [GS05]. In L1, a connector defines a set of possible interactions on a set of ports, which may be output, input or event ports. For compatibility reasons, connectors on inputs and outputs must obey the same constraints as L0 connectors, that is, connect exactly one output to a set of inputs and define exactly one interaction involving all connected ports. The L1 execution model is more asynchronous as executions of different connectors are interleaved. In fact, it corresponds to the second variant of BIP presented in the previous chapter called multishot BIP.

6.2.1 The L0 contract framework

At the semantic level, the L0 contract framework of the SPEEDS HRC model is based on a simple trace-based representation, and uses set operations for the definition of the operators. In other words, LTS are used at the L0 level as recognizers for the trace representation. This is convenient, as the synchronous composition semantics translates into simple intersection of trace sets. In this context, we are not concerned with the specific form of a trace (more details can be found elsewhere [BFM+08]). Instead, we simply assume that for any set of ports $\mathcal{P}$ there exists a set of corresponding traces G over those ports which we call *behaviors* or *runs* over $\mathcal{P}$. As discussed in Section 5.3, the set of behaviors is equipped with a projection operator $\Pi_{\mathcal{P}_1}(G)$which restricts the behaviors to ports in $\mathcal{P}_1 \subseteq \mathcal{P}_2$, and a corresponding inverse projection $\Pi^{-1}_{\mathcal{P}_2}(G)$ to extend the behaviors to a larger set of ports $\mathcal{P}_2$.

A component K with interface $\mathcal{P}_K$ at level L0 is defined as a set of behaviors over $\mathcal{P}_K$. The behaviors correspond to the history of ports that are visited when traversing the transitions of the LTS. Composition, in the original formulation, is defined as a new LTS obtained by the Cartesian product of the transition systems, and by retaining only the pairs of transitions whose labels of ports match, given the correspondence induced by the connectors. If the matching ports of the two components had the same names, composition at the level of trace sets would boil down to a simple intersection of the sets of behaviors. Because this is not true in general, and is forbidden by the definitions of

our framework (components must have disjoint sets of ports under composition), we must introduce the connectors as explicit components that establish a synchronous relation between the histories of connected ports. The collection of these simple connectors forms the glues Γ of our framework at the L0 level. In addition to that, to make intersection work, we must also equalize the ports of all trace sets using inverse projection, to have a coherent representation of the composite. In particular, if $\mathcal{K} = \{K_1, \ldots, K_n\}$ is a set of components such that $\mathcal{P}_1, \ldots, \mathcal{P}_n$ are pairwise disjoint, then a composition operator Γ for $\mathcal{K}$ is a component K_Γ defined on the ports $\mathcal{P} = \mathcal{P}_\Gamma \cup \bigcup_{i=1}^{n} \mathcal{P}_i$, and:

$$\begin{aligned} K &= \Gamma\{K_1, \ldots, K_n\} \\ &= \Pi_{\mathcal{P}_\Gamma}(K_\Gamma \cap \Pi_{\mathcal{P}}^{-1} K_1 \cap \ldots \cap \Pi_{\mathcal{P}}^{-1} K_n) \end{aligned}$$

Component K_Γ is always taken as an identity operator, and is used exclusively to rename ports in the composition and to construct the new interface $\mathcal{P}_\Gamma$. In the following, at the semantic level, we implicitly assume the appropriate connector components are used whenever a composition is required, and instead use components with equal sets of ports for convenience.

The definition of $\circ$ is straightforward. Since glues are themselves components, their composition follows the same principle as component composition. Finally, the $\cong$ relation on $\mathcal{K}$ is taken as equality of sets of traces.

It is easy to define a notion of conformance $\preccurlyeq$ for the L0 model. This notion is equivalent to the traditional notion of refinement, and is defined as trace containment. More formally, if K_1 and K_2 are components over the same set of ports $\mathcal{P}$, then $K_1 \preccurlyeq K_2$ if and only if $K_1 \subseteq K_2$.

L0 contracts

Contracts are defined in L0 as pairs (A, G) of components over the same set of ports. In particular, A represents the *assumptions* of the contract, or, equivalently, the behaviors that are considered acceptable by the contract. Likewise, G expresses the guarantees, or those behaviors that are possible under the contract, provided the assumptions are satisfied. The glue Γ is implied by port name matching.

The definition of *refinement* under context $\sqsubseteq_{E,\Gamma}$ is derived from the definition of composition and conformance as in Example 2.2.5. $K_1 \sqsubseteq_{E,\Gamma} K_2$ if and only if $\Gamma\{K_1, E\} \preccurlyeq \Gamma\{K_2, E\}$. The relation so defined is a preorder, and satisfies by definition the conditions required by our framework.

In the L0 model, contract satisfaction is defined as refinement under the context of the assump-

tions. Formally, a component K satisfies a contract $\mathcal{C} = (A, G)$ if and only if

$$K \cap A \subseteq G.$$

Observe that the above is equivalent to $K \cap A \subseteq G \cap A$. In our framework, this translates into $\Gamma\{K, A\} \preccurlyeq \Gamma\{G, A\}$, where Γ is the appropriate composition operator that computes the identity relation. By definition, this is the same as $K \sqsubseteq_{A,\Gamma} G$. Thus, the definition of satisfaction in HRC is consistent with the more general definition of satisfaction of our framework.

In the L0 model there exists a unique maximal component satisfying a contract $\mathcal{C}$, namely:

$$M_{\mathcal{C}} = G \cup \neg A, \tag{6.1}$$

where $\neg$ denotes the operation of complementation on the set of all behaviors over ports $\mathcal{P}_A$. The operation of computing a canonical form is well defined, since the maximal implementation is unique, and it is idempotent. It is easy to show that $K \models \mathcal{C}$ if and only if $K \subseteq M_{\mathcal{C}}$. We say that a contract $\mathcal{C} = (A, G)$ is in *canonical form* when $G = M_{\mathcal{C}}$. Every contract has an equivalent contract in canonical form, which is obtained by replacing G with $M_{\mathcal{C}}$. In the following, we focus on contracts in canonical form, since several expressions can be simplified. The limitation is that complementation may not be effective in certain models (such as timed models). Besides, in our methodology, assumptions are meant to be used top-down by implementations, not to help discharging guarantees in a bottom-up fashion. Thus, $M_{\mathcal{C}}$ is likely to be more expressive than needed. In such cases, the use of canonical forms is precluded, and the more generic L1 theory is required.

Parallel composition of contracts in L0

Contract composition formalizes how contracts related to different components should be combined to specify a single, compound, component. Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be contracts. First, composing these two contracts amounts to composing their promises. Regarding assumptions, however, the situation is more subtle. Suppose first that these two contracts possess disjoint sets of ports and variables. At a first sight, the assumptions of the composite should intuitively be simply the conjunction of the assumptions of the rich components, since the environment should satisfy all the assumptions. In general, however, part of the assumptions A_1 will be already satisfied by composing $\mathcal{C}_1$ with $\mathcal{C}_2$ acting as a partial environment for $\mathcal{C}_1$. Therefore, G_2 can contribute to relaxing assumption A_1, and vice versa. Whence the following definition:

Definition 6.2.2 (Composition of contracts) *The parallel composition* $\mathcal{C}_1 \,||\, \mathcal{C}_2$ *is defined as the contract* $\mathcal{C} = (A, G)$ *such that:*

$$\begin{aligned} A &= (A_1 \cap A_2) \cup \neg(G_1 \cap G_2), \\ G &= G_1 \cap G_2. \end{aligned}$$

Note that the so defined contract is in canonical form.

The following result, which is not true in the general case, expresses the strong compositional properties of the L0 contract theory.

Lemma 6.2.3 *If* $K \models \mathcal{C}$ *and* $K \models \mathcal{C}'$*, then* $K \cap K' \models \mathcal{C} \,||\, \mathcal{C}'$.

Dominance in L0

Dominance is defined in L0 as a contravariant relation between assumptions and guarantees. The relation between guarantees is required for general dominance, whereas the second condition for assumptions is intended for preserving well-formedness.

Definition 6.2.4 (L0-Dominance) *A contract* $\mathcal{C} = (A, G)$ *dominates a contract* $\mathcal{C}' = (A', G')$ *if and only if* $A \supseteq A'$ *and* $G \subseteq G'$.

Dominance amounts to relaxing assumptions and reinforcing promises. Note that if $\mathcal{C}$ dominates $\mathcal{C}'$ and $\mathcal{C}'$ dominates $\mathcal{C}$, then $\mathcal{C} = \mathcal{C}'$. Furthermore, if $\mathcal{C}$ dominates $\mathcal{C}'$ then $M_{\mathcal{C}} \models \mathcal{C}'$. This property implies the following result, which relates the definition of dominance in L0 to the more general definition of our contract framework:

Lemma 6.2.5 *If* $K \models \mathcal{C}$ *and* $\mathcal{C}$ *dominates* $\mathcal{C}'$*, then* $K \models \mathcal{C}'$.

As a partial order, dominance admits both greatest lower bounds and least upper bounds, which we call conjunction and disjunction of contracts, respectively. Let $\mathcal{C}_1 = (A_1, G_1)$ and $\mathcal{C}_2 = (A_2, G_2)$ be contracts. The greatest lower bound of $\mathcal{C}_1$ and $\mathcal{C}_2$, written $\mathcal{C} = \mathcal{C}_1 \sqcap \mathcal{C}_2$, is given by $\mathcal{C} = (A, G)$ where $A = A_1 \cup A_2$ and $G = G_1 \cap G_2$. Similarly, the least upper bound of $\mathcal{C}_1$ and $\mathcal{C}_2$, written $\mathcal{C} = \mathcal{C}_1 \sqcup \mathcal{C}_2$, is given by $\mathcal{C}' = (A', G')$ where $A = A_1 \cap A_2$ and $G = G_1 \cup G_2$. Note that the result of these operations are contracts in canonical form. Minimal and maximal contracts can also be defined, as well as complementation, making L0 contracts a boolean algebra.

Discussion

The L0 contract framework has strong compositional properties, which derive from its simple definition and operators. The theory, however, depends on the effectiveness of certain operators, complementation in particular, which are necessary for the computation of canonical forms. While the complete theory can be formulated without the use of canonical forms, complementation remains fundamental in the definition of contract composition, which is at the basis of system construction.

Circular reasoning is sound for a contract framework based on canonical forms. This is because any behavior that is not allowed to the environment, is instead allowed by the guarantees. This is no longer the case for contracts which are *not* in canonical form. This is a limitation of the L0 framework, since working with canonical forms could prove computationally hard.

In the following section we analyze a higher level model, for which there is no obvious composition of contracts. We will show that the high level framework, called L1, can be used consistently with L0 by proving that properties in L1 are preserved in L0 by soundness results.

6.2.2 The L1 contract framework

The HRC L1 framework (1) is able to express more complex glues and (2) decouples the notions of conformance and refinement in a context in order to enable circular reasoning even for contracts that are not in normal form, and thus avoid the limitations of the L0 contract framework.

For defining the L1 contract framework, we follow the general methodology exhibited in Section 2.1. The L1 component framework corresponds to the multishot BIP component framework of Chapter 5.

L1 contracts

Let $\mathcal{P}$ be an interface and Γ a glue with $\mathcal{P}$ in its support set, which implicitly defines an interface $\mathcal{P}_E$ of the environment. An L1-contract for $\mathcal{P}$ is then of the form (A, Γ, G) where A is an LTS on $\mathcal{P}_E$ and G an LTS on $\mathcal{P}$. Note that we have provided a compositional semantics which associates with every component an atomic one that is equivalent, so it is sufficient to consider only atomic components, that is, LTS. It now remains to define the two refinement relations, *conformance* and *refinement under context*. We choose L1-conformance to be *simulation*, that is, the structural counterpart of L0-conformance (trace inclusion).

Definition 6.2.6 (L1-conformance) $K_1 \preccurlyeq^{L1} K_2 \triangleq [\![K_1]\!]$ *simulates* $[\![K_2]\!]$.

Thus, L1-conformance is identical to L0-conformance for behaviors without non-observable non-determinism, and otherwise it is stronger. Note that in verification tools, in order to check trace inclusion efficiently, one will generally check simulation anyway.

For refinement under context, which defines contract satisfaction, we choose a stronger relation than for L0, in order to be able to use circular reasoning for dominance checks.

Definition 6.2.7 (L1-satisfaction) *Given an LTS K on an interface $\mathcal{P}$ and a contract (A, Γ, G) for $\mathcal{P}$, we define satisfaction as:*

$$K \sqsubseteq^{L1}_{A,\Gamma} G \triangleq \begin{cases} \Gamma\{K, A_{det}\} \preccurlyeq^{L1} \Gamma\{G, A_{det}\} \\ (q_K, q_A)\, \mathcal{R}\, (q_G, q'_A) \wedge q_K \xrightarrow{\alpha}_K \implies q_G \xrightarrow{\alpha}_G \end{cases}$$

where A_{det} is the determinization of A (see Definition 1.1.6) and $\mathcal{R}$ is the relation proving that $\Gamma\{K, A_{det}\} \preccurlyeq^{L1} \Gamma\{G, A_{det}\}$.

That is, $\sqsubseteq^{L1}$ strengthens the usual notion of refinement under context defined in Example 2.2.5 and used in the L0 framework by determinizing A and adding a condition stating that every transition of a refining state must correspond to a transition in each corresponding abstract state — but the target states must be related only if the environment allows this transition. As a consequence, $\sqsubseteq^{L1}$ allows circular reasoning. Note that in frameworks with data, one usually requires preservation of certain predicates from the concrete to the abstract transition system; here we require preservation of transition enabledness, independently of data.

Note that we are interested in preserving well-formedness in this framework. As a result, we use strong dominance here as in Section 3.1.5.

6.2.3 Consistency between L0 and L1

In the previous sections, we have introduced two different notions of refinement under context: $\sqsubseteq^{L0}$ and $\sqsubseteq^{L1}$ where the second is strictly stronger than the first one. As already stated, circular reasoning is sound for $\sqsubseteq^{L1}$ but not for $\sqsubseteq^{L0}$.

Now, based on the results of Section 3.3, we provide a way of relaxing the dominance checking for L1. Symmetrically, we propose a verification condition for L0-satisfaction that does not require the actual environment to refine the abstraction of the environment in any context. As explained in Section 3.3, these two improvements are based respectively on the following two theorems.

Theorem 6.2.8 *If $K \sqsubseteq^{L1}_{A,\Gamma} G$ and $E \sqsubseteq^{L0}_{G,\Gamma} A$, then $K \sqsubseteq^{L1}_{E,\Gamma} G$.*

This allows us to relax our sufficient condition for dominance in L1. This is particularly interesting because checking L0-satisfaction is obviously easier than checking L1-satisfaction. Thus, checking that a set of contracts $\{\mathcal{C}_i\}_{i=1}^n$ L1-dominates a contract $\mathcal{C}$ requires one L1-satisfaction check and n L0-satisfaction checks. Moreover, this relaxed rule allows establishing dominance more often. Finally, L1- and L0-satisfaction are very similar. As a result, if L1-satisfaction cannot be established, the relation under construction can be reused to check L0-dominance.

Theorem 6.2.9 *If $K \sqsubseteq^{L0}_{A,\Gamma} G$ and $E \sqsubseteq^{L1}_{G,\Gamma} A$, then $K \sqsubseteq^{L0}_{E,\Gamma} G$.*

This second theorem allows in the SPEEDS project building a complete tool chain based on a set of tools checking either L0-satisfaction or L1-dominance (implemented by a set of L1-satisfaction checks). We can check a complete contract hierarchy requiring dominance checks using the existing L1 dominance checker, and at the very end check satisfaction mostly with the more scalable L0 satisfaction checker.

Based on these theorems, it is possible to combine in several ways the results from the L0-satisfaction and L1-dominance/satisfaction checker that have been developed in the SPEEDS project.

6.2.4 Implementation issues

We have developed a dominance and satisfaction checker for the L1 framework. This prototype is based on the Maude rewrite engine [Mau]. We do not present it in detail here but we describe a specific problem that is related to the question of structuring of systems discussed in Section 3.1.1. The situation is illustrated in Figure 6.1: given a dominance problem, that is, a top-level contract $\mathcal{C}$ for an interface $\mathcal{P}$; a decomposition Γ_I of this interface; a set of low-level contracts $\{\mathcal{C}_i\}_{i=1}^n$, how do we find the glues Γ_{E_1} and Γ_{E_2} that are necessary to apply our sufficient condition for dominance? The L0-dominance problem is more specific than the general dominance problem described in Chapter 2, because connectors are rendezvous, and rendezvous can be easily restructured. However, in the dominance checker that we have implemented, the exported port of a connector is defined by a function $*$ of its support set: if γ is such that $S_\gamma = \{p_1, p_2\}$, then $p_\gamma = *\{p_1, p_2\}$. Thus, we are left with a problem of unification of names, because Γ_{E_1} and Γ_{E_2} are obvious, but their exported interfaces do not necessarily match $\mathcal{P}_{A_1}$ and $\mathcal{P}_{A_2}$. Rather than introducing explicitly renaming of ports in the system, our implementation computes a renaming of ports in $\mathcal{P}_{A_1}$, $\mathcal{P}_{A_2}$ and $\mathcal{P}$ such that all interfaces match.

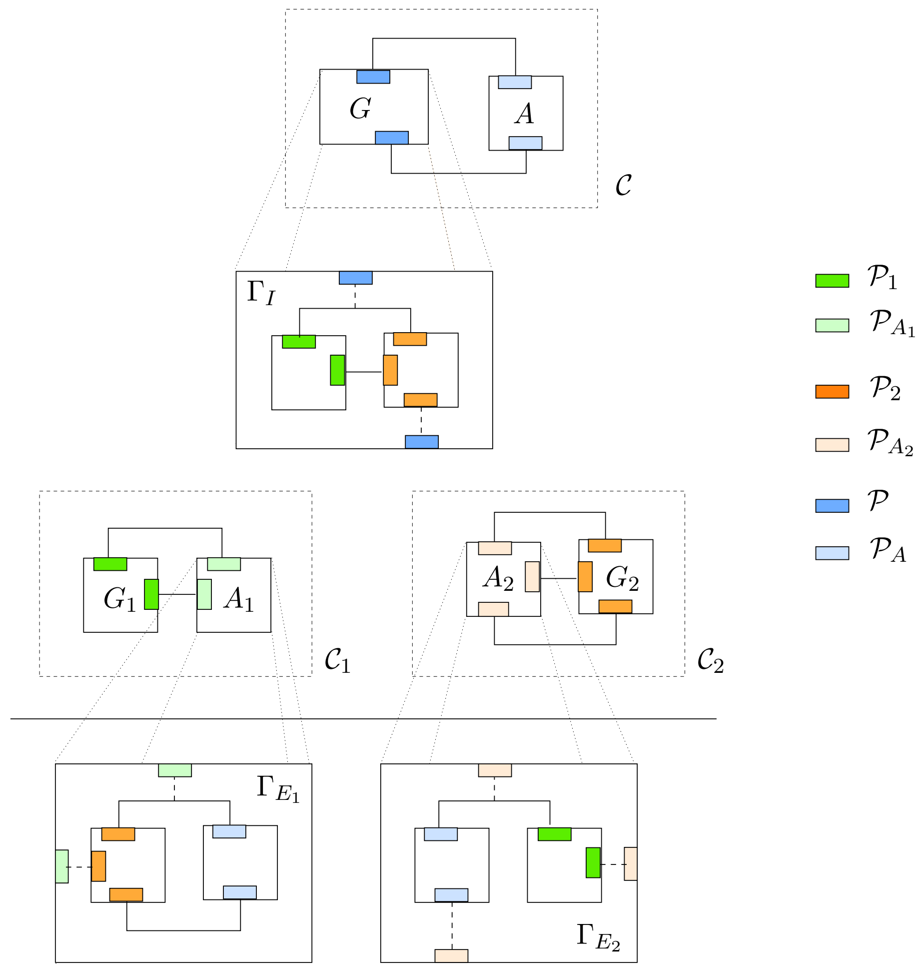


Figure 6.1 – Given $\mathcal{C}$, $\mathcal{C}_1$, $\mathcal{C}_2$ and Γ_I, how to find Γ_{E_1} and Γ_{E_2}?

6.2.5 Proofs

We prove here the following theorems:

1. L1-satisfaction implies L0-satisfaction
2. Soundness of circular circular reasoning for $\sqsubseteq^{L1}$
3. Soundness of pseudo-circular reasoning for $\sqsubseteq^{L1}$ and $\sqsubseteq^{L0}$
4. Soundness of pseudo-circular reasoning for $\sqsubseteq^{L0}$ and $\sqsubseteq^{L1}$

We also provide characterizations of $\sqsubseteq^{L1}$ and $\sqsubseteq^{L0}$ which are used in the proofs.

Lemma 6.2.10 *$[\![\Gamma\{K_1, E\}]\!]$ simulates $[\![\Gamma\{K_2, E\}]\!]$ if and only if there exists $\mathcal{R} \subseteq (Q_1 \times Q_E) \times Q_2$ such that:*

- $(q_1^0, q_E^0)\,\mathcal{R}\,q_2^0$
- *If $(q_1, q_E)\,\mathcal{R}\,q_2$ and $(q_1, q_E) \xrightarrow{\alpha} (q_1', q_E')$ for $\alpha \in \mathcal{I}(\Gamma)$ such that $\alpha = \alpha_K \cup \alpha_E$, then $\exists q_2'$ s.t. $q_2 \xrightarrow{\alpha_K} q_2'$ and $(q_1', q_E')\,\mathcal{R}\,q_2'$.*

Characterization of L1-satisfaction

Lemma 6.2.11 *$K_1 \sqsubseteq^{L1}_{E,\Gamma} K_2$ if and only if there exists a relation $\mathcal{R} \subseteq (Q_1 \times 2^{Q_E}) \times Q_2$ such that:*

- $(q_1^0, \{q_E^0\})\,\mathcal{R}\,q_2^0$
- *If $(q_1, \mathcal{Q}_E)\,\mathcal{R}\,q_2$ and $q_1 \xrightarrow{\alpha_K}_1 q_1'$, there exists q_2' such that:*
  1. $q_2 \xrightarrow{\alpha_K}_2 q_2'$
  2. *if there exists α_E, $q_E \in \mathcal{Q}_E$ and $q_E' \in Q_E$ such that $q_E \xrightarrow{\alpha_E}_E q_E'$ and $\alpha_K \cup \alpha_E \in \mathcal{I}(\Gamma)$, then $(q_1', \mathcal{Q}_E')\,\mathcal{R}\,q_2'$ where $\mathcal{Q}_E'$ is $\{q_E' \mid \exists q_E \in \mathcal{Q}_E \text{ s.t. } q_E \xrightarrow{\alpha_E}_E q_E'\}$.*

We use the convention that $\forall q \xrightarrow{\emptyset} q$, so the above condition includes cases where only K_1 or only E move on.

Characterization of L0-satisfaction

Lemma 6.2.12 *$K_1 \sqsubseteq^{L0}_{E,\Gamma} K_2$ if and only if there exists a relation $\mathcal{R} \subseteq (2^{Q_1} \times Q_E) \times 2^{Q_2}$ such that:*

- $(\{q_1^0\}, q_E^0)\,\mathcal{R}\,\{q_2^0\}$
- *If $(\mathcal{Q}_1, q_E)\,\mathcal{R}\,\mathcal{Q}_2$ and $(q_1, q_E) \xrightarrow{\alpha} (q_1', q_E')$ for $q_1 \in \mathcal{Q}_1$ with $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}(\Gamma)$, then there exists $q_2 \in \mathcal{Q}_2$ and $q_2' \in Q_2$ such that $q_2 \xrightarrow{\alpha_K} q_2'$ and $(\mathcal{Q}_1', q_E')\,\mathcal{R}\,\mathcal{Q}_2'$ for $\mathcal{Q}_1'$ defined as $\{q_1' \mid \exists q_1 \in \mathcal{Q}_1 \text{ s.t. } q_1 \xrightarrow{\alpha_K}_1 q_1'\}$ and $\mathcal{Q}_2'$ defined as $\{q_2' \mid \exists q_2 \in \mathcal{Q}_2 \text{ s.t. } q_2 \xrightarrow{\alpha_K}_2 q_2'\}$.*

We use the convention that $\forall q \xrightarrow{\emptyset} q$, so the above condition includes cases where only K_1 or only E moves on.

L1-satisfaction implies L0-satisfaction

Proof. We suppose that $K_1 \sqsubseteq^{L1}_{E,\Gamma} K_2$, and then we prove that $K_1 \sqsubseteq^{L0}_{E,\Gamma} K_2$.
- As $K_1 \sqsubseteq^{L1}_{E,\Gamma} K_2$, there exists a relation $\mathcal{R} \subseteq (Q_1 \times 2^{Q_E}) \times Q_2$ as in Lemma 6.2.11.
- We define $\mathcal{R}' \subseteq (2^{Q_1} \times Q_E) \times 2^{Q_2}$ as follows: we define $(\{q_1\}, q_E)\, \mathcal{R}'\, \{q_2\}$ iff there exists $\mathcal{Q}_E \subseteq Q_E$ such that $q_E \in \mathcal{Q}_E$ and $(q_1, \mathcal{Q}_E)\, \mathcal{R}\, q_2$.
- Obviously $(\{q_1^0\}, q_E^0)\, \mathcal{R}'\, \{q_2^0\}$.
- Now suppose $(\{q_1\}, q_E)\, \mathcal{R}'\, \{q_2\}$ and $(q_1, q_E) \xrightarrow{\alpha} (q_1', q_E')$ with $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}(\Gamma)$. We must show that there exists $q_2' \in Q_2$ such that $q_2 \xrightarrow{\alpha_K} q_2'$ and $(\{q_1'\}, q_E')\, \mathcal{R}'\, \{q_2'\}$.
- Let $\mathcal{Q}_E \subseteq Q_E$ be such that $q_E \in \mathcal{Q}_E$ and $(q_1, \mathcal{Q}_E)\, \mathcal{R}\, q_2$. We have $q_1 \xrightarrow{\alpha_K}_1 q_1'$. Thus, there exists q_2' such that $q_2 \xrightarrow{\alpha_K}_2 q_2'$.
- Besides, as $q_E \xrightarrow{\alpha_E}_E q_E'$, we have $(q_1', \mathcal{Q}_E')\, \mathcal{R}\, q_2'$ where $\mathcal{Q}_E'$ is $\{q_E' \mid \exists q_E \in \mathcal{Q}_E\ s.t.\ q_E \xrightarrow{\alpha_E}_E q_E'\}$.
- Hence, by definition of$\mathcal{R}'$: $(\{q_1'\}, q_E')\, \mathcal{R}'\, \{q_2'\}$. □

Soundness of circular reasoning for $\sqsubseteq^{L1}$

Proof. Let K be a component on $\mathcal{P}$, (E, Γ) a context for $\mathcal{P}$ and $\mathcal{C} = (A, \Gamma, G)$ a contract for $\mathcal{P}$.
- Suppose that $K \sqsubseteq^{L1}_{A,\Gamma} G \wedge E \sqsubseteq^{L1}_{G,\Gamma} A$. We have to prove that $K \sqsubseteq^{L1}_{E,\Gamma} G$.
- As $K \sqsubseteq^{L1}_{A,\Gamma} G$ and $E \sqsubseteq^{L1}_{G,\Gamma} A$, there exist two relations $\mathcal{R}_1$ and $\mathcal{R}_2$ on respectively $(Q_K \times 2^{Q_A}) \times Q_G$ and $(Q_E \times 2^{Q_G}) \times Q_A$ as in Lemma 6.2.11.
- We define $\mathcal{R} \subseteq (Q_K \times 2^{Q_E}) \times Q_G$ as follows: for any $q_K \in Q_K$, $\mathcal{Q}_E \subseteq Q_E$ and $q_G \in Q_G$, $(q_K, \mathcal{Q}_E)\, \mathcal{R}\, q_G$ iff there exists $\mathcal{Q}_A \subseteq Q_A$, $\mathcal{Q}_G \subseteq Q_G$, $q_E \in \mathcal{Q}_E$ and $q_A \in \mathcal{Q}_A$ such that $q_G \in \mathcal{Q}_G$, $(q_K, \mathcal{Q}_A)\, \mathcal{R}_1\, q_G$ and $(q_E, \mathcal{Q}_G)\, \mathcal{R}_2\, q_A$.
- We have to prove that $\mathcal{R}$ ensures the conditions of Lemma 6.2.11. Obviously, $(q_K^0, \{q_E^0\})\, \mathcal{R}\, q_G^0$.
- Let $q_K \in Q_K$, $\mathcal{Q}_E \subseteq Q_E$ and $q_G \in Q_G$ be such that $(q_K, \mathcal{Q}_E)\, \mathcal{R}\, q_G$. Let $\mathcal{Q}_A \subseteq Q_A$, $\mathcal{Q}_G \subseteq Q_G$, $q_E \in \mathcal{Q}_E$ and $q_A \in \mathcal{Q}_A$ be such that $q_G \in \mathcal{Q}_G$, $(q_K, \mathcal{Q}_A)\, \mathcal{R}_1\, q_G$ and $(q_E, \mathcal{Q}_G)\, \mathcal{R}_2\, q_A$.
- Now suppose $q_K \xrightarrow{\alpha_K}_K q_K'$.
- We have to prove that there exists q_G' such that:

1. $q_G \xrightarrow{\alpha_K}_G q_G'$
2. if there exists α_E, $q_E \in \mathcal{Q}_E$ and $q_E' \in Q_E$ such that $q_E \xrightarrow{\alpha_E}_E q_E'$ and $\alpha_K \cup \alpha_E \in \mathcal{I}(\Gamma)$, then $(q_K', \mathcal{Q}_E')\, \mathcal{R}\, q_G'$ where $\mathcal{Q}_E'$ is $\{q_E' \mid \exists q_E \in \mathcal{Q}_E\ s.t.\ q_E \xrightarrow{\alpha_E}_E q_E'\}$.

- Because $(q_K, \mathcal{Q}_A)\, \mathcal{R}_1\, q_G$ and $q_K \xrightarrow{\alpha_K}_K q_K'$, we know that there exists q_G' such that:
  - $q_G \xrightarrow{\alpha_K}_G q_G'$

– if there exists α_A, $q_A \in \mathcal{Q}_A$ and $q'_A \in Q_A$ such that $q_A \xrightarrow{\alpha_A}_A q'_A$ and $\alpha_K \cup \alpha_A \in \mathcal{I}(\Gamma)$, then $(q'_K, \mathcal{Q}'_A)\,\mathcal{R}\,q'_G$ with $\mathcal{Q}'_A$ defined as $\{q'_A \mid \exists q_A \in \mathcal{Q}_A \; s.t. \; q_A \xrightarrow{\alpha_A}_A q'_A\}$.

- We show that this q'_G satisfies the two conditions required above from $\mathcal{R}$. Condition 1. is exactly the same as for $\mathcal{R}_1$.

- Let us show that the second condition holds. Suppose that there exists α_E, $q_E \in \mathcal{Q}_E$ and $q'_E \in Q_E$ such that $q_E \xrightarrow{\alpha_E}_E q'_E$ and $\alpha_K \cup \alpha_E \in \mathcal{I}(\Gamma)$. Let $\mathcal{Q}'_E$ be defined as above by $\mathcal{Q}'_E = \{q'_E \mid \exists q_E \in \mathcal{Q}_E \; s.t. \; q_E \xrightarrow{\alpha_E}_E q'_E\}$. We have to show that $(q'_K, \mathcal{Q}'_E)\,\mathcal{R}\,q'_G$.

- As $(q_E, \mathcal{Q}_G)\,\mathcal{R}_2\,q_A$ and $q_E \xrightarrow{\alpha_E}_E q'_E$, we know that there exists q'_A such that:

– $q_A \xrightarrow{\alpha_E}_A q'_A$

– if there exists α_G, $q_G \in \mathcal{Q}_G$ and $q'_G \in Q_G$ such that $q_G \xrightarrow{\alpha_G}_G q'_G$ and $\alpha_G \cup \alpha_E \in \mathcal{I}(\Gamma)$, then $(q'_E, \mathcal{Q}'_G)\,\mathcal{R}\,q'_A$ with $\mathcal{Q}'_G$ defined as $\{q'_G \mid \exists q_G \in \mathcal{Q}_G \; s.t. \; q_G \xrightarrow{\alpha_G}_G q'_G\}$.

- Thus, applying the second property offered by $\mathcal{R}_1$ to this α_E and q'_A, we obtain that $(q'_K, \mathcal{Q}'_A)\,\mathcal{R}_1\,q'_G$ where $\mathcal{Q}'_A$ is defined as $\{q'_A \mid \exists q_A \in \mathcal{Q}_A \; s.t. \; q_A \xrightarrow{\alpha_E}_A q'_A\}$.

- Besides, as there exist indeed $q_G \in \mathcal{Q}_G$ and $q'_G \in Q_G$ such that $q_G \xrightarrow{\alpha_K}_G q'_G$, then applying the second property offered by $\mathcal{R}_2$, we obtain $(q'_E, \mathcal{Q}'_G)\,\mathcal{R}_2\,q'_A$ for $\mathcal{Q}'_G$ defined as $\{q'_G \mid \exists q_G \in \mathcal{Q}_G \; s.t. \; q_G \xrightarrow{\alpha_K}_G q'_G\}$.

- Finally, according to the definition of $\mathcal{R}$, we can conclude that $(q'_K, \mathcal{Q}'_E)\,\mathcal{R}\,q'_G$. □

Pseudo-circular reasoning for $\sqsubseteq^{L1}$ and $\sqsubseteq^{L0}$

Proof. Let K be a component on an interface $\mathcal{P}$, (E, Γ) a context for $\mathcal{P}$ and $\mathcal{C} = (A, \Gamma, G)$ a contract for $\mathcal{P}$. We suppose that $K \sqsubseteq^{L1}_{A,\Gamma} G$ and $E \sqsubseteq^{L0}_{G,\Gamma} A$, and then we prove that $K \sqsubseteq^{L1}_{E,\Gamma} G$.
- As $K \sqsubseteq^{L1}_{A,\Gamma} G$, there exists a relation $\mathcal{R}_1$ on $(Q_K \times 2^{Q_A}) \times Q_G$ as in Lemma 6.2.11.
- As $E \sqsubseteq^{L0}_{G,\Gamma} A$, there exists a relation $\mathcal{R}_2$ on $(2^{Q_E} \times Q_G) \times 2^{Q_A}$ as in Lemma 6.2.12.
- We define $\mathcal{R} \subseteq (Q_K \times 2^{Q_E}) \times Q_G$ as follows: for any $q_K \in Q_K$, $\mathcal{Q}_E \subseteq Q_E$ and $q_G \in Q_G$, we define $(q_K, \mathcal{Q}_E)\,\mathcal{R}\,q_G$ iff there exists $\mathcal{Q}_A \subseteq Q_A$ such that $(q_K, \mathcal{Q}_A)\,\mathcal{R}_1\,q_G$ and $(\mathcal{Q}_E, q_G)\,\mathcal{R}_2\,\mathcal{Q}_A$.
- We have to prove that $\mathcal{R}$ ensures the conditions of Lemma 6.2.11. Obviously, $(q^0_K, \{q^0_E\})\,\mathcal{R}\,q^0_G$.
- Let $q_K \in Q_K, \mathcal{Q}_E \subseteq Q_E, q_G \in Q_G$ be such that $(q_K, \mathcal{Q}_E)\,\mathcal{R}\,q_G$. Let $\mathcal{Q}_A$ be such that $(q_K, \mathcal{Q}_A)\,\mathcal{R}_1\,q_G$ and $(\mathcal{Q}_E, q_G)\,\mathcal{R}_2\,\mathcal{Q}_A$.
- Now suppose $q_K \xrightarrow{\alpha_K}_K q'_K$.
- We have to prove that there exists p'_G such that:

1. $q_G \xrightarrow{\alpha_K}_G q'_G$
2. if there exists α_E, $q_E \in \mathcal{Q}_E$ and $q'_E \in Q_E$ such that $q_E \xrightarrow{\alpha_E}_E q'_E$ and $\alpha_K \cup \alpha_E \in \mathcal{I}(\Gamma)$, then $(q'_K, \mathcal{Q}'_E)\,\mathcal{R}\,q'_G$ where $\mathcal{Q}'_E$ is $\{q'_E \mid \exists q_E \in \mathcal{Q}_E\ s.t.\ q_E \xrightarrow{\alpha_E}_E q'_E\}$.

- Because $(q_K, \mathcal{Q}_A)\,\mathcal{R}_1\,q_G$ and $q_K \xrightarrow{\alpha_K}_K q'_K$, we know that there exists q'_G such that:
  - $q_G \xrightarrow{\alpha_K}_G q'_G$
  - if there exists α_A, $q_A \in \mathcal{Q}_A$ and $q'_A \in Q_A$ such that $q_A \xrightarrow{\alpha_A}_A q'_A$ and $\alpha_K \cup \alpha_A \in \mathcal{I}(\Gamma)$, then $(q'_K, \mathcal{Q}'_A)\,\mathcal{R}\,q'_G$ with $\mathcal{Q}'_A$ defined as $\{q'_A \mid \exists q_A \in \mathcal{Q}_A\ s.t.\ q_A \xrightarrow{\alpha_A}_A q'_A\}$.
- We show that this q'_G satisfies the two conditions required above from $\mathcal{R}$. Condition 1. is exactly the same as for $\mathcal{R}_1$.
- Let us show that the second condition holds. Suppose there exist α_E, $q_E \in \mathcal{Q}_E$ and $q'_E \in Q_E$ such that $q_E \xrightarrow{\alpha_E}_E q'_E$ and $\alpha_K \cup \alpha_E \in \mathcal{I}(\Gamma)$. Let $\mathcal{Q}'_E$ be defined as above by $\mathcal{Q}'_E = \{q'_E \mid \exists q_E \in \mathcal{Q}_E\ s.t.\ q_E \xrightarrow{\alpha_E}_E q'_E\}$. We have to show that $(q'_K, \mathcal{Q}'_E)\,\mathcal{R}\,q'_G$.
- As $q_E \xrightarrow{\alpha_E}_E q'_E$ and $q_G \xrightarrow{\alpha_K}_G q'_G$, we know that $(q_E, q_G) \xrightarrow{\alpha} (q'_E, q'_G)$. Thus, from $(\mathcal{Q}_E, q_G)\,\mathcal{R}_2\,\mathcal{Q}_A$ and because $q_E \in \mathcal{Q}_E$, we can conclude that there exist $q_A \in \mathcal{Q}_A$ and $q'_A \in Q_A$ such that $q_A \xrightarrow{\alpha_E} q'_A$ and $(\mathcal{Q}'_E, q'_G)\,\mathcal{R}_2\,\mathcal{Q}'_A$ for $\mathcal{Q}'_E$ defined as above and $\mathcal{Q}'_A$ defined as $\{q'_A \mid \exists q_A \in \mathcal{Q}_A\ s.t.\ q_A \xrightarrow{\alpha_E}_A q'_A\}$.
- Now, applying the second property offered by $\mathcal{R}_1$ to this α_E, q_A and q'_A, we obtain that $(q'_K, \mathcal{Q}'_A)\,\mathcal{R}_1\,q'_G$ with $\mathcal{Q}_A$ as defined above.
- Finally, according to the definition of $\mathcal{R}$, we can conclude that $(q'_K, \mathcal{Q}'_E)\,\mathcal{R}\,q'_G$. □

Pseudo-circular reasoning for $\sqsubseteq^{L0}$ and $\sqsubseteq^{L1}$

Proof. Let K be a component on an interface $\mathcal{P}$, (E, Γ) a context for $\mathcal{P}$ and $\mathcal{C} = (A, \Gamma, G)$ a contract for $\mathcal{P}$. We suppose that $K \sqsubseteq^{L0}_{A,\Gamma} G$ and $E \sqsubseteq^{L1}_{G,\Gamma} A$, and then we prove that $K \sqsubseteq^{L0}_{E,\Gamma} G$.
- As $K \sqsubseteq^{L0}_{A,\Gamma} G$, there exists a relation $\mathcal{R}_1$ on $(2^{Q_K} \times Q_A) \times 2^{Q_G}$ as in Lemma 6.2.12.
- As $E \sqsubseteq^{L1}_{G,\Gamma} A$, there exists a relation $\mathcal{R}_2$ on $(Q_E \times 2^{Q_G}) \times Q_A$ as in Lemma 6.2.11.
- We define $\mathcal{R} \subseteq (2^{Q_K} \times Q_E) \times 2^{Q_G}$ as follows: for any $\mathcal{Q}_K \subseteq Q_K$, $q_E \in Q_E$ and $\mathcal{Q}_G \subseteq Q_G$, we define $(\mathcal{Q}_K, q_E)\, \mathcal{R}\, \mathcal{Q}_G$ iff there exists $q_A \in Q_A$ such that $(\mathcal{Q}_K, q_A)\, \mathcal{R}_1\, \mathcal{Q}_G$ and $(q_E, \mathcal{Q}_G)\, \mathcal{R}_2\, q_A$.
- We have to prove that $\mathcal{R}$ ensures the conditions of Lemma 6.2.12. Obviously, $(\{q^0_K\}, q^0_E)\, \mathcal{R}\, \{q^0_G\}$.
- Let $\mathcal{Q}_K \subseteq Q_K$, $q_E \in Q_E$ and $\mathcal{Q}_G \subseteq Q_G$ be such that $(\mathcal{Q}_K, q_E)\, \mathcal{R}\, \mathcal{Q}_G$. Let q_A be such that $(\mathcal{Q}_K, q_A)\, \mathcal{R}_1\, \mathcal{Q}_G$ and $(q_E, \mathcal{Q}_G)\, \mathcal{R}_2\, q_A$.
- Now suppose $(q_K, q_E) \xrightarrow{\alpha} (q'_K, q'_E)$ for $q_K \in \mathcal{Q}_K$ with $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}(\Gamma)$. Let $\mathcal{Q}'_K$ be defined as $\{q'_K \mid \exists q_K \in \mathcal{Q}_K\ s.t.\ q_K \xrightarrow{\alpha_K}_K q'_K\}$ and $\mathcal{Q}'_G$ as $\{q'_G \mid \exists q_G \in \mathcal{Q}_G\ s.t.\ q_G \xrightarrow{\alpha_K}_G q'_G\}$. We have to prove that there exists $q_G \in \mathcal{Q}_G$ and $q'_G \in Q_G$ such that $q_G \xrightarrow{\alpha_K} q'_G$ and that $(\mathcal{Q}'_K, q'_E)\, \mathcal{R}\, \mathcal{Q}'_G$.
- As $(q_K, q_E) \xrightarrow{\alpha} (q'_K, q'_E)$, we know that $q_E \xrightarrow{\alpha_E}_E q'_E$. Then, because $(q_E, \mathcal{Q}_G)\, \mathcal{R}_2\, q_A$ and $q_E \xrightarrow{\alpha_E}_E q'_E$, there exists q'_A such that:
  – $q_A \xrightarrow{\alpha_E}_A q'_A$
  – if there exists α_G, $q_G \in \mathcal{Q}_G$ and $q'_G \in Q_G$ such that $q_G \xrightarrow{\alpha_G}_G q'_G$ and $\alpha_G \cup \alpha_E \in \mathcal{I}(\Gamma)$, then $(q'_E, \mathcal{Q}'_G)\, \mathcal{R}\, q'_A$ with $\mathcal{Q}'_G$ defined as above.
- This implies in particular that $(q_K, q_A) \xrightarrow{\alpha} (q'_K, q'_A)$. Thus, from $(q_K, \mathcal{Q}_A)\, \mathcal{R}_1\, q_G$ and because $q_A \in \mathcal{Q}_A$, we can conclude that there exists $q_G \in \mathcal{Q}_G$ and $q'_G \in Q_G$ such that $q_G \xrightarrow{\alpha_K} q'_G$ and $(\mathcal{Q}'_K, q'_A)\, \mathcal{R}_1\, \mathcal{Q}'_G$ for $\mathcal{Q}'_K$ and $\mathcal{Q}'_G$ defined as above.
- This gives us the q_G and q'_G we were looking for. There only remains to prove that $(\mathcal{Q}'_K, q'_E)\, \mathcal{R}\, \mathcal{Q}'_G$.
- Now, applying the second property offered by $\mathcal{R}_2$ to this α_K, q_G and q'_G, we obtain that $(q'_E, \mathcal{Q}'_G)\, \mathcal{R}_2\, q'_A$.
- Hence, according to the definition of $\mathcal{R}$, the conclusion that $(q'_K, \mathcal{Q}'_E)\, \mathcal{R}\, q'_G$. □

Chapter 7

Contract frameworks for transition systems

In this chapter, we explore more thoroughly various refinement under context relations that ensure soundness of circular reasoning within contract frameworks based on transition systems. We introduce a contract framework for modal transition systems (MTS), which is directly inspired by the BIP semantic framework of Chapter 4. When using MTS, structural consistency as discussed in Section 3.2.1 is a useful notion which allows detecting whether a contract can possibly be satisfied by a composition using a given glue. We provide a necessary and sufficient condition for structural consistency. We then focus on refinement in any context, which is the same as conformance for frameworks without priorities, as is usually the case. This is no longer true when priorities are introduced. The contract framework for MTS with priorities shows the interest of combining those two concepts: while the MTS contract framework without priorities is directly related to the framework for LTS, it is no longer the case in presence of priorities.

7.1 Labeled transition systems

7.1.1 Definitions

When dealing with LTS, there are two obvious conformance relations to consider: inclusion of traces and simulation. In this chapter, we focus on the various refinement relations consistent with simulation. Of course, a similar approach is possible for refinements with respect to inclusion of traces.

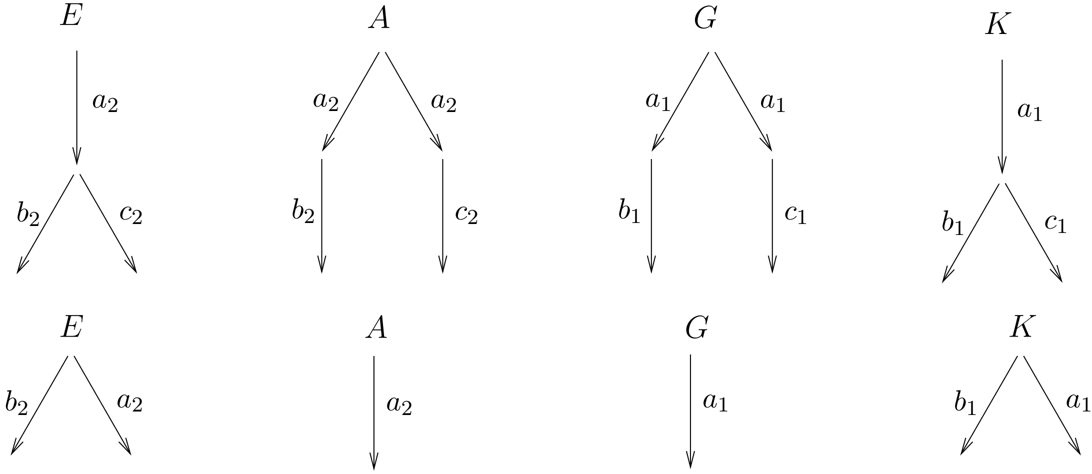


Figure 7.1 – $K \sqsubseteq^2_{A,\mathcal{I}} G$ and $E \sqsubseteq^2_{G,\mathcal{I}} A$ but $K \not\sqsubseteq^2_{E,\mathcal{I}} G$.

Definition 7.1.1 (Conformance) $K_1 \preccurlyeq K_2 \triangleq K_1$ *simulates* K_2.

Let us start with a quick summary of the various refinement under context relations that have been proposed until now in this thesis. Two relations are natural candidates:

1. $K_1 \sqsubseteq^1_{E,\mathcal{I}} K_2 \triangleq K_1$ simulates K_2
2. $K_1 \sqsubseteq^2_{E,\mathcal{I}} K_2 \triangleq \mathcal{I}\{K_1, E\}$ simulates $\mathcal{I}\{K_2, E\}$

However, none of them is satisfactory. The first one is too strong as it does not take at all the environment into account — it actually corresponds to refinement in any context. On the contrary, the second relation (which we have called the usual refinement under context in Example 2.2.5) is too weak to ensure soundness of circular reasoning. Let us recall the two reasons for that: one is related to non-determinism, and so does not interfere when conformance is inclusion of traces; the other is related to rendezvous interactions — as illustrated in Figure 7.1. In Chapter 4, we have proposed a relation for which circular reasoning is sound (Definition 4.2.3):

3. $K_1 \sqsubseteq^3_{E,\mathcal{I}} K_2 \triangleq$ There exists a relation $\mathcal{R} \subseteq (Q_1 \times Q_E) \times Q_2$ such that:
   - $(q_1^0, q_E^0)\, \mathcal{R}\, q_2^0$
   - if $(q_1, q_E)\, \mathcal{R}\, q_2$, $q_1 \xrightarrow{\alpha_K}_1 q_1'$ and $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}$, then there exists q_2' such that $q_2 \xrightarrow{\alpha_K}_2 q_2'$ and any q_E' such that $q_E \xrightarrow{\alpha_E}_E q_E'$ satisfies $(q_1', q_E')\, \mathcal{R}\, q_2'$

A slightly different relation is suggested in Section 6.2.2 about the L1 framework (Definition 6.2.7):

4. $K_1 \sqsubseteq^4_{E,\mathcal{I}} K_2 \triangleq \begin{cases} \text{There exists a simulation } \mathcal{R} \text{ between } \mathcal{I}\{K_1, E_{det}\} \text{ and } \mathcal{I}\{K_2, E_{det}\} \\ \text{such that } (q_1, q_E)\, \mathcal{R}\, (q_2, q_E') \wedge q_1 \xrightarrow{\alpha}_1 \implies q_2 \xrightarrow{\alpha}_2 \end{cases}$

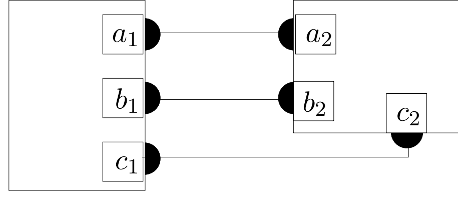


Figure 7.2 – Structure of the system used to build the counterexamples of this section.

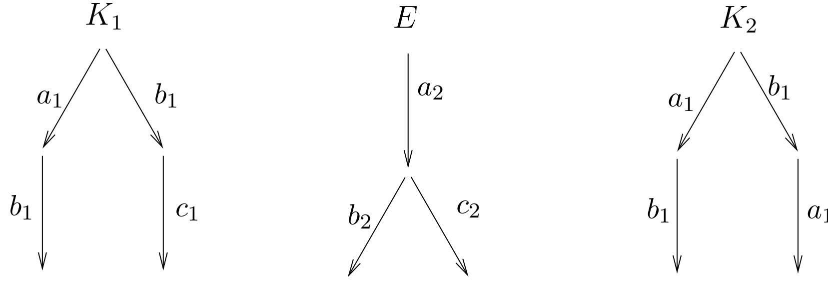


Figure 7.3 – $K_1 \sqsubseteq^4_{E,\mathcal{I}} K_2$ but $K_1 \not\sqsubseteq^1_{E,\mathcal{I}} K_2$.

Those relations are defined in a similar way. They are in fact related by the following theorem.

Theorem 7.1.2 *Consider K_1 and K_2 two components for an interface $\mathcal{P}$ and $(E, \mathcal{I})$ a context for $\mathcal{P}$.*

$$K_1 \sqsubseteq^1_{E,\mathcal{I}} K_2 \implies K_1 \sqsubseteq^4_{E,\mathcal{I}} K_2 \implies K_1 \sqsubseteq^3_{E,\mathcal{I}} K_2 \implies K_1 \sqsubseteq^2_{E,\mathcal{I}} K_2$$

None of the converse implications is true. Figure 7.3 shows a situation $K_1 \sqsubseteq^4_{E,\mathcal{I}} K_2$ but $K_1 \not\sqsubseteq^1_{E,\mathcal{I}} K_2$. Besides, in Figure 7.1, $K \sqsubseteq^2_{A,\mathcal{I}} G$ but $K \not\sqsubseteq^3_{A,\mathcal{I}} G$. These counterexamples are given for: K_1 and K_2 defined on $\mathcal{P} = \{a_1, b_1, c_1\}$; E defined on $\mathcal{P}_E = \{a_2, b_2, c_2\}$; $\mathcal{I} = \{\{a_1, a_2\}, \{b_1, b_2\}, \{c_1, c_2\}\}$, as shown in Figure 7.2 — we use a representation with connectors only for illustration purposes. Note that these counterexamples are relevant for any framework encompassing rendezvous (strong synchronization). Finally, $\sqsubseteq^4$ is strictly stronger than $\sqsubseteq^3$ because it does not take into account interactions which are structurally forbidden.

Now that we have classified these relations, we focus on pseudo-circular reasoning. The following theorem states that it is always sound except for the case of circular reasoning for $\sqsubseteq^2$.

Theorem 7.1.3 *For $i \in [1, 4]$, $K \sqsubseteq^i_{A,\mathcal{I}} G$ and $E \sqsubseteq^j_{G,\mathcal{I}} A$ implies $K \sqsubseteq^i_{E,\mathcal{I}} G$ if and only if i or j is not* 2.

Does there exist a refinement weaker than $\sqsubseteq^3$ but still strong enough to offer sound circular reasoning? In fact, there is: the condition that any transition $q_1 \xrightarrow{\alpha_K}_1 q_1'$ that is not structurally forbidden should have a counterpart $q_2 \xrightarrow{\alpha_2}_2 q_2'$ can be relaxed.

Definition 7.1.4 (Controlled interaction) *Consider a component K on an interface $\mathcal{P}$ and an interaction model $\mathcal{I}$ such that $\mathcal{P} \subseteq S_{\mathcal{I}}$. An interaction $\alpha_K \in 2^{\mathcal{P}}$ is* controlled *by K if and only if $\alpha_E \in \mathcal{I}$.*

If K controls α_K, then it cannot rely on its environment to prevent α_K from taking place. However, if α_K is always part of interactions $\alpha = \alpha_K \cup \alpha_E$ such that α_E is controlled by the environment, and if the environment offers none of these α_E, then it does not matter whether α_K is enabled in q_2 or not, because this has no effect on the composition. Thus, we can relax our definition:

$$K_1 \sqsubseteq^{3'}_{E,\mathcal{I}} K_2 \triangleq \text{there exists a relation } \mathcal{R} \subseteq (Q_1 \times Q_E) \times Q_2 \text{ such that:}$$

– $(q_1^0, q_E^0) \,\mathcal{R}\, q_2^0$

– if $(q_1, q_E) \,\mathcal{R}\, q_2$, $q_1 \xrightarrow{\alpha_K}_1 q_1'$ and $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}$, then:

$$\begin{cases} \text{there exists } q_2' \text{ such that } q_2 \xrightarrow{\alpha_K}_2 q_2' \text{ and any } q_E' \text{ such that } q_E \xrightarrow{\alpha_E}_E q_E' \text{ satisfies } (q_1', q_E') \,\mathcal{R}\, q_2' \\ \text{or} \\ \forall \alpha_E \in \mathcal{I}_E \text{ s.t. } \alpha_K \cup \alpha_E \in \mathcal{I} : \alpha_E \in \mathcal{I} \wedge \nexists q_E' : q_E \xrightarrow{\alpha_E}_E q_E' \end{cases}$$

Note that in a framework without rendez-vous and non-determinism, this definition boils down to usual refinement.

7.1.2 Refinement in any context

Composition à la BIP of LTS without priorities only reduces the set of possible behaviors. Thus, the "least helpful" environment is the one that accepts everything. This implies that refinement in any context is equivalent to refinement in the empty context, i.e., conformance — in other words, simulation. As a consequence, refinement in any context is *the same* for the four relations defined in this chapter:

Theorem 7.1.5 *For $i \in \{1, 4\}$, given two components K_1 and K_2 on the same interface, $K_1 \sqsubseteq^i K_2$ if and only if K_1 simulates K_2.*

7.1.3 Structural consistency

Consistency issues are almost irrelevant when dealing with LTS because the LTS that does nothing trivially refines any LTS, and it can be obtained by any composition of "idle" LTS. As a result, structural consistency is always ensured. This will no longer be the case for MTS.

7.2 Modal transition systems

LTS allow reasoning about safety properties only. This is why we now present a contract framework for *Modal Transition Systems* (MTS), in which we can reason about not only safety properties and also some progress properties, such as absence of interlocks or specification of some "minimal behavior".

MTS [LX90] are labeled transition systems where transitions have in addition a modality, either *must* or *may*. MTS used as specifications are very useful because they contain both an over- and under-approximation of their possible implementations, thus allowing verification of both safety and some progress properties. We now present a contract framework based on MTS and equipped with a refinement under context such that circular reasoning is sound. This framework is tightly related to the one introduced for LTS.

7.2.1 Definitions

We suppose given a set of ports $Ports$.

- $\mathcal{K}$ is the set of all possible MTS with labels in 2^{Ports}
- $\cong$ is syntactic equality, possibly after renaming of states
- glues are defined by interaction sets denoted $\mathcal{I}$ in $\mathcal{P}$ and $\mathcal{I}\{K_1, \ldots, K_n\}$ is the composition of MTS defined below
- $\circ$ is defined as for LTS

Definition 7.2.1 (Composition of MTS) *The composition of a set of MTS $\{K_i\}_{i=1}^n$ on disjoint interfaces $\{\mathcal{P}_i\}_{i=1}^n$ by a glue $\mathcal{I}$ on $P = \bigcup_{i=1}^n \mathcal{P}_i$, which is denoted $\mathcal{I}\{K_1, \ldots, K_n\}$, is an MTS $(Q, q^0, 2^P, \dashrightarrow, \longrightarrow)$ such that $Q = \prod_{i=1}^n Q_i$, $q^0 = (q_1^0, \ldots, q_n^0)$ and $\dashrightarrow$ and $\longrightarrow$ are defined by: $\forall \alpha \in \mathcal{I}$, $\forall q^1 = (q_1^1, \ldots, q_n^1), q^2 = (q_1^2, \ldots, q_n^2) \in Q$,*

- *$q^1 \overset{\alpha}{\dashrightarrow} q^2$ if and only if $\forall i : \alpha \cap \mathcal{P}_i = \emptyset$ or $q_i^1 \overset{\alpha_i}{\dashrightarrow} q_i^2$ where $\alpha_i = \alpha \cap \mathcal{P}_i$*
- *$q^1 \overset{\alpha}{\longrightarrow} q^2$ if and only if $\forall i : \alpha \cap \mathcal{P}_i = \emptyset$ or $q_i^1 \overset{\alpha_i}{\longrightarrow} q_i^2$ where $\alpha_i = \alpha \cap \mathcal{P}_i$*

Interestingly, the convention that $\forall q : q \overset{\emptyset}{\longrightarrow} q'$ cannot be safely applied to *must*-transitions for MTS. Note that this definition preserves modal consistency.

The counterpart of simulation for MTS is modal refinement as defined in Section 1.1.2. This is the reason why we choose here to define conformance as modal refinement.

Definition 7.2.2 (Conformance) $K_1 \preccurlyeq K_2 \triangleq K_1 \preceq K_2$*, where $\preceq$ is modal refinement.*

Definition 7.2.3 (Refinement under context) *Let K_1 and K_2 be two MTS on the same interface $\mathcal{P}$, and $(E, \mathcal{I})$ a context for $\mathcal{P}$. $K_1 \sqsubseteq_{E,\mathcal{I}} K_2 \triangleq$ there exists a relation $\mathcal{R} \subseteq (Q_1 \times Q_E) \times Q_2$ such that $(q_1^0, q_E^0)\, \mathcal{R}\, q_2^0$ and where $(q_1, q_E)\, \mathcal{R}\, q_2$ implies:*

1. *if $q_1 \overset{\alpha_K}{\dashrightarrow}_1 q_1'$ and $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}$, then there exists q_2' such that $q_2 \overset{\alpha_K}{\dashrightarrow}_2 q_2'$ and any q_E' such that $q_E \overset{\alpha_E}{\dashrightarrow}_E q_E'$ satisfies $(q_1', q_E')\, \mathcal{R}\, q_2'$*
2. *if $q_2 \xrightarrow{\alpha_K}_2 q_2'$ and $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}$, then there exists q_1' such that $q_1 \xrightarrow{\alpha_K}_1 q_1'$ and any q_E' such that $q_E \xrightarrow{\alpha_E}_E q_E'$ satisfies $(q_1', q_E')\, \mathcal{R}\, q_2'$*

We use here again the convention that $\forall q : q \overset{\emptyset}{\dashrightarrow} q$ and $\forall q : q \xrightarrow{\emptyset} q$. Note however, that if there is a *must*-transition $q_2 \xrightarrow{\alpha_K}_2 q_2'$, it is not permitted to K_1 to fire an internal transition before offering α_K. This definition is directly inspired by the refinement under context for LTS of Chapter 4. In particular, it can adapted to take into account triggers and synchrons in a way similar to that discussed in the previous section.

7.2.2 Refinement in any context

Interestingly, here also, refinement in any context and conformance are identical.

Theorem 7.2.4 *Modal refinement $\preceq$ corresponds to refinement in any context.*

7.2.3 Structural consistency

The contract $(A, \mathcal{I}, G)$ represented in Figure 7.4 states that in an environment that may always offer a_2 and b_2, a component satisfying $\mathcal{C}$ must offer only a_1 in state q_0 and then it must offer b_1 and possibly a_1 in state q_1. However, this behavior cannot be obtained by a composition using $\mathcal{I}_I$ defined as two rendezvous connectors between respectively a_1 and a_2 and then b_1 and b_2. Indeed, firing a_1 modifies the conditions imposed on b_1, while these two interactions are part of two different components which cannot observe each other. Thus, b_1 must be offered and forbidden in states which are indistinguishable by the component in charge of it.

We now define a characterization of structural consistency for MTS, based on a projection of the *must*-relation of G. Interestingly, the projection discussed in Section 5.3 cannot be directly adapted to MTS without interfering with the definition of refinement under context that we have introduced for MTS. The reason has already been mentioned: to establish that $K_1 \sqsubseteq_{A,\mathcal{I}} K_2$, if there is a *must*-transition $q_2 \xrightarrow{\alpha_K}_2 q_2'$, it is not permitted to K_1 to fire an internal transition before offering α_K. As a result, the projection that we propose here does not introduce transitions labeled by $\emptyset$.

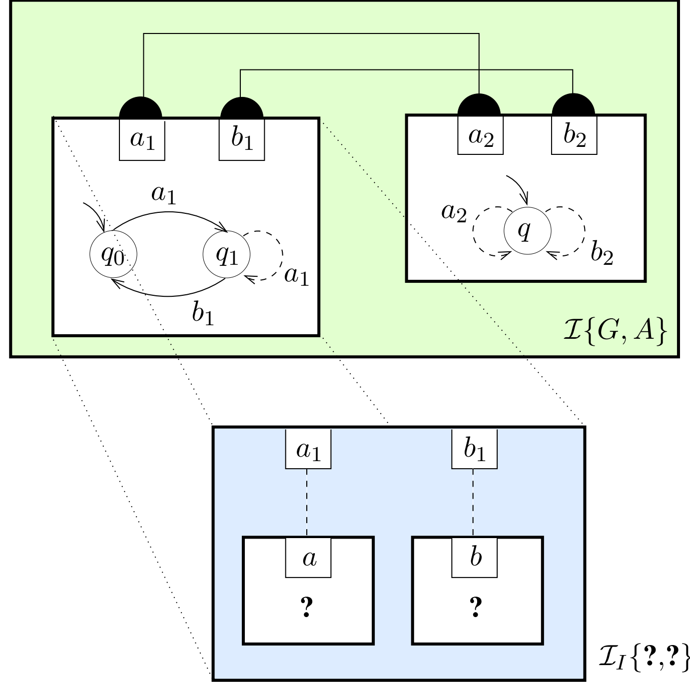


Figure 7.4 – An example of structural inconsistency.

Definition 7.2.5 (Projection of an MTS) *Let $K = (Q, q^0, 2^{\mathcal{P}}, \dashrightarrow, \longrightarrow)$ be an MTS on $\mathcal{P}$. The projection of K onto $\mathcal{P}' \subseteq \mathcal{P}$ is an MTS $(2^Q, \mathcal{Q}^0, 2^{\mathcal{P}'}, \dashrightarrow_\pi, \longrightarrow_\pi)$ where:*

– $\mathcal{Q}^0 = \{q^0\} \cup \{q \mid q^0 \overset{\alpha_1}{\dashrightarrow} \ldots \overset{\alpha_k}{\dashrightarrow} q$ *where* $\forall i \in [1,k] : \alpha_i \cap \mathcal{P}' = \emptyset\}$

– $\dashrightarrow_\pi$ *and* $\longrightarrow_\pi$ *are the smallest relations in* $2^Q \times 2^{\mathcal{P}'} \backslash \{\emptyset\} \times 2^Q$ *such that for* $\mathcal{Q}, \mathcal{Q}' \subseteq Q$*:*

- $\mathcal{Q} \overset{\alpha'}{\dashrightarrow}_\pi \mathcal{Q}' \triangleq \exists q \in \mathcal{Q}\ s.t.\ q \overset{\alpha}{\dashrightarrow} q'$ *with* $\alpha \cap \mathcal{P}' = \alpha'$ *and* $\mathcal{Q}' = \{q'\} \cup \{q'' \mid q' \overset{\alpha_1}{\dashrightarrow} \ldots \overset{\alpha_k}{\dashrightarrow} q''$ *where* $\forall i \in [1,k] : \alpha_i \cap \mathcal{P}' = \emptyset\}$

- $\mathcal{Q} \overset{\alpha'}{\longrightarrow}_\pi \mathcal{Q}' \triangleq \exists q \in \mathcal{Q}\ s.t.\ q \overset{\alpha}{\longrightarrow} q'$ *with* $\alpha \cap \mathcal{P}' = \alpha'$ *and* $\mathcal{Q}' = \{q'\} \cup \{q'' \mid q' \overset{\alpha_1}{\longrightarrow} \ldots \overset{\alpha_k}{\longrightarrow} q''$ *where* $\forall i \in [1,k] : \alpha_i \cap \mathcal{P}' = \emptyset\}$

Informally, $\mathcal{Q}'$ is the set of states in Q that can be reached from q' by firing only transitions that are invisible in the projection. Note that this definition preserves modal consistency.

Let $\mathcal{C} = (A, \mathcal{I}, G)$ be a contract for an interface $\mathcal{P}$. Let $\mathcal{I}_I$ be a glue on $\mathcal{P}$ and $\{P_i\}_{i=1}^n$ a partition of $\mathcal{P}$. As defined in Section 3.2.1, $\mathcal{C}$ is *consistent with* $\mathcal{I}_I$ and $\{P_i\}$ if and only if there exist $B_1, \ldots, B_n$ on respectively $P_1, \ldots, P_n$ such that $\mathcal{I}_I\{B_1, \ldots, B_n\} \models \mathcal{C}$.

By projecting the *must*-transitions of G onto each $\mathcal{P}_i$, we obtain an under-approximation of the *must*-transitions of the possible K_i. As a consequence, if $\mathcal{I}_I\{G_1, \ldots, G_n\}$ does not satisfy $\mathcal{C}$, then $\mathcal{C}$

is *not* consistent with $\mathcal{I}_I$ and $\{\mathcal{P}_i\}$. In a top-down design approach, this means that either contract $\mathcal{C}$ must be modified, or another decomposition has to be found. This is the meaning of the following theorem.

Formally, define for $i \in [1, n]$ a component G_i obtained from the projection of G onto $\mathcal{P}_i$ by replacing the *may* relation by the *must* relation — that is, if the projection of G onto $\mathcal{P}_i$ is $(2^Q, \mathcal{Q}^0, 2^{\mathcal{P}'} \backslash \{\emptyset\}, \dashrightarrow_\pi, \longrightarrow_\pi)$, then $G_i = (2^Q, \mathcal{Q}^0, 2^{\mathcal{P}'} \backslash \{\emptyset\}, \longrightarrow_\pi, \longrightarrow_\pi)$.

Theorem 7.2.6 *$\mathcal{C}$ is* consistent with *$\mathcal{I}_I$ and $\{P_i\}$ if and only if $\mathcal{I}_I\{G_1, \dots, G_n\} \models \mathcal{C}$.*

Note that by projecting the *may*-transitions of G onto each $\mathcal{P}_i$, we obtain an under-approximation of the *may*-transitions of the possible K_i, which boils down to an under-approximation of an over-approximation, thus being useless.

7.3 Labeled transition systems with priorities

7.3.1 Definitions

- $\mathcal{K}$, GL and $\circ$ are respectively the set of components, the set of glues and the composition operator on glues defined in the BIP semantic framework of Section 1.2.1
- $K_1 \cong K_2 \triangleq |K_1| = |K_2|$
- $K_1 \preccurlyeq K_2 \triangleq [\![K_1]\!]$ simulates $[\![K_2]\!]$

Unlike in Chapter 4 and because of the priorities, it is not sufficient here to define refinement under context for atomic components, namely LTS. The reason for this is that it is not possible to define a compositional semantics in the form of LTS, and one needs to preserve the priority order of a composite component. As a result, the following definitions are given for components which can be atomic as well as composite. These definitions are all based on the compositional semantics of components defined in Section 1.2.1, which associates with a component K a pair $(B_K, \prec_K)$, also denoted $|K|$. In order to define refinement under context, we introduce the notion of *inhibited* interaction. An interaction is inhibited in a local state when it cannot have maximal priority in a given composition.

Definition 7.3.1 (Inhibited interaction) *Consider an LTS B on $\mathcal{P}$ and a glue $gl = (\Gamma, \prec)$ on $\mathcal{P} \cup \mathcal{P}_E$. An interaction $\alpha \subseteq \mathcal{P}$ is* inhibited *in a state $q \in Q_B$ w.r.t. gl iff it is possible in q and one of the following propositions holds:*

- *There is no interaction $\alpha_E \subseteq \mathcal{P}_E$ such that $\alpha \cup \alpha_E \in \mathcal{I}(\Gamma)$.*

– *For any* $\alpha_E \subseteq \mathcal{P}_E$ *such that* $\alpha \cup \alpha_E \in \mathcal{I}(\Gamma)$*, there exists* $\alpha' \subseteq \mathcal{P}$ *that is possible in* q *and such that* $\alpha' \cup \alpha_E \in \mathcal{I}(\Gamma)$ *and* $\alpha \cup \alpha_E \prec \alpha' \cup \alpha_E$*.*

Sometimes, we mention explicitly the transitions by which α is inhibited, namely the transitions α' defined above. Typically, an inhibited interaction is a subset of an interaction in the same connector that is also possible.

Let $(K_E, \Gamma_\prec)$ be a context for an interface $\mathcal{P}$ with $|K_E| = (B_E, \prec_E)$. Let $K_i = (\mathcal{B}_i, \Gamma_i, \prec_i)$, $i \in \{1, 2\}$, be two components on P. $|K_i|$ is denoted $(B_i, \prec_i)$.

Definition 7.3.2 (Refinement under context) $K_1 \sqsubseteq_{E,\Gamma_\prec} K_2 \triangleq \prec_1 = \prec_2$ *and there exists* $\mathcal{R} \subseteq (Q_1 \times Q_E) \times Q_2$ *such that:*

– $(q_1^0, q_E^0) \,\mathcal{R}\, q_2^0$

– *if* $(q_1, q_E) \,\mathcal{R}\, q_2$, $q_1 \xrightarrow{\alpha_K}_1 q_1'$ *and* $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}$*, then:*

1. *there exists* q_2' *such that* $q_2 \xrightarrow{\alpha_K}_2 q_2'$
2. *all interactions that can inhibit* α *in* q_2 *are also possible in* q_1
3. *any* q_E' *such that* $q_E \xrightarrow{\alpha_E}_E q_E'$ *satisfies* $(q_1', q_E') \,\mathcal{R}\, q_2'$

The only difference between this refinement relation and the one introduced in Section 1.2.1 for the BIP semantic level is item 2. which adds a condition related to priorities, as we explain now. Even if a transition $q_1 \xrightarrow{\alpha_K}_1 q_1'$ has a counterpart in K_2, it may happen that this counterpart is inhibited by another transition (or a set of transitions). As a consequence, K_1 may not behave like K_2 with respect to $\Gamma_\prec$. To avoid this, we require that an interaction α_K may be possible in q_1 only if all the interactions that inhibit α in q_2 are also possible in q_1.

Interestingly, refinement in any context is much stronger than conformance, as $K_1 \sqsubseteq K_2$ if and only if $|K_1|$ ready-simulates $|K_2|$.

7.4 Modal transition systems with priorities

7.4.1 Definitions

In this section, we present a contract-based verification framework which uses Modal Transition Systems (MTS) to represent atomic behaviors, and such that interaction is complex and involves static priorities.

- $\mathcal{K}$, GL and $\circ$ are respectively the set of components, the set of glues and the composition operator on glues defined in the BIP semantic framework of Chapter 4 adapted to handle MTS instead of LTS
- $K_1 \cong K_2 \triangleq |K_1| = |K_2|$
- $K_1 \preccurlyeq K_2 \triangleq [\![K_1]\!]$ refines $[\![K_2]\!]$

Because we are interested in structural verification and we do not want to compute an order in which assumptions must be discharged, we want refinement under context to preserve soundness of circular reasoning. In the previous chapter, this question was answered in the simpler case where there are no priorities. Priorities increase significantly the complexity of the problem, because interactions that are inhibited may become enabled in a given context.

Definition 7.4.1 (Possible vs. enabled interaction) *Let B be an MTS and $(\Gamma, \prec)$ a composition operator on $\mathcal{P}$. An interaction $\alpha \in 2^{\mathcal{P}}$ is called* must-possible *(resp.* may-*possible) in a state $q \in Q$ iff there exists $q' \in Q$ s.t. $q \xrightarrow{\alpha} q'$ (resp. $q \overset{\alpha}{\dashrightarrow} q'$).*
An interaction is said must-enabled *(resp.* may-enabled*) in q iff is* must-*possible (resp.* may-*possible) in q and there is no interaction with higher priority* may-*possible in q.*

The function $pos_\Box$ (resp. $pos_\Diamond$) : $Q \longrightarrow 2^{2^P}$ returns for any state $q \in Q$ the set of *must*-possible (resp. *may*-possible) interactions in q, and similarly for $en_\Box$ and $en_\Diamond$. Note that *must*-enabledness is lost as soon as an interaction with higher priority may be possible. As some transitions that are inhibited may get higher priority through composition, we need another notion which we call *maximality*. Maximal interactions are possible in a given state q but not necessarily enabled. They might be inhibited by another interaction, but in the given context they are not necessarily inhibited by it. Besides, interactions that are enabled in q are not necessarily maximal because they may not be part of any possible interaction in the global system.

One can do better than considering all interactions possible in q_1. Indeed, an interaction α_K that is possible in q_1 may not be part of any interaction enabled in q, for example in the following cases:

- such an interaction is structurally impossible
- any such interaction has lower priority than a *must*-enabled interaction.
- it is a subset of an interaction in the same connector that is also possible.

Interactions that are *must*-enabled are preserved by refinement, except those that are inhibited by another *must*-transition. Besides, forbidden interactions (interactions which are not *may*-enabled) are also preserved by refinement except those that are structurally impossible. Hence the following definition of maximality.

Definition 7.4.2 (Maximal interaction) *Let K be a component on an interface $\mathcal{P}$ with open semantics $(B, \prec_P)$. Let γ_E be a glue on $\mathcal{P} \cup \mathcal{P}_E$. We denote $(\Gamma, \prec) = \gamma_E \circ (\Gamma_P, \prec_P)$. An interaction α_K is* $*$-maximal *for $* \in \{\Box, \Diamond\}$ in q_B, q_A w.r.t. γ_E, which is denoted $\alpha_K \in \max_* (q_B, q_A, \gamma)$, iff $\alpha_K \in \mathrm{pos}_* (q_B)$ and $\exists \alpha_E$ such that:*

- $\alpha_K \cup \alpha_E \in \mathrm{pos}_\Diamond (q_B, q_A)$
- $\forall \alpha.\ \alpha_K \cup \alpha_E \prec \alpha \implies \alpha \notin \mathrm{pos}_\Box (q_B, q_A)$
- $\forall \alpha'_K . \forall \alpha'_E .\ \alpha_K \cup \alpha_E \prec \alpha'_K \cup \alpha'_E \implies \alpha'_K \notin \mathrm{pos}_\Box (q_B) \vee \alpha'_E \backslash \alpha_E \notin \mathrm{pos}_\Box (q_E)$

This definition of maximality is based on the properties proposed above. We denote by $\max (q_1, q_2, \gamma)$ the set of interactions that are maximal in q_1 w.r.t. γ and q_2. We can now define refinement under context.

Let (K_E, γ) be a context for an interface $\mathcal{P}$ with $K_E = (B_E, \gamma_E, \prec_E)$ and $\gamma = (\Gamma, \prec)$. Let $K_i = (B_i, \gamma_i, \prec_i)$, $i \in \{1, 2\}$, be two components on P. We define $\gamma_G = \gamma \circ \gamma_E$.

Definition 7.4.3 (Refinement under context) $K_1 \sqsubseteq_{K_E, \gamma} K_2 \triangleq$ *there exists $\mathcal{R} \subseteq (Q_1 \times Q_E) \times Q_2$ such that $(q_1^0, q_E^0) \mathcal{R}\, q_2^0$ and $(q_1, q_E) \mathcal{R}\, q_2$ implies:*

1. $\max_\Diamond (q_1, \gamma_G) = \max_\Diamond (q_2, \gamma_G)$
2. $\forall \alpha_K \cup \alpha_E \in \mathrm{en}_\Diamond (q_1, q_E), \forall q_1\ s.t.\ q_1 \overset{\alpha_K}{\dashrightarrow} q'_1$,
$$\exists q'_2 : \begin{cases} q_2 \overset{\alpha_K}{\dashrightarrow} q'_2 \\ \forall q'_E,\ q_E \overset{\alpha_E}{\dashrightarrow} q'_E \implies (q'_1, q'_E) \mathcal{R}\, q'_2 \end{cases}$$
3. $\max_\Box (q_2, \gamma_G) = \max_\Box (q_1, \gamma_G)$
4. $\forall \alpha_K \cup \alpha_E \in \mathrm{en}_\Box (q_2, q_E), \forall q_2\ s.t.\ q_2 \xrightarrow{\alpha_K} q'_2$,
$$\exists q'_1 : \begin{cases} q_1 \xrightarrow{\alpha_K} q'_1 \\ \forall q'_E,\ q_E \xrightarrow{\alpha_E} q'_E \implies (q'_2, q'_E) \mathcal{R}\, q'_1 \end{cases}$$

As usual, conditions 2 and 4 correspond to the intuitive notion that $B \sqsubseteq_{A,\gamma} G$ iff $\gamma\{B, A\}$ ready-simulates $\gamma\{G, A\}$, except that they imply determinization of E. Conditions 1 and 3 ensure that only the past is used, while abstracting away transitions that are harmless because they will never be enabled in the global system. With MTS and priorities, a component may thus rely on transitions from its environment to inhibit some of its own transitions, which was not possible without modalities.

7.5 Proofs

Theorem 7.1.2 $\sqsubseteq^1 \implies \sqsubseteq^4 \implies \sqsubseteq^3 \implies \sqsubseteq^2$.

Proof. Let K_1, K_2 be defined on P, let $(E, \mathcal{I})$ be a context for P with E defined on P_E.

Step 1: $K_1 \sqsubseteq^1_{E,\mathcal{I}} K_2 \implies K_1 \sqsubseteq^4_{E,\mathcal{I}} K_2$.
- Suppose $K_1 \sqsubseteq^1_{E,\mathcal{I}} K_2$. By definition, there exists a relation $\mathcal{R}$ s.t. for any pair $(q_1, q_2) \in Q_1 \times Q_2$: $q_1 \,\mathcal{R}\, q_2 \wedge q_1 \xrightarrow{\alpha_K}_1 q_1' \implies \exists q_2' \; s.t. \; q_2 \xrightarrow{\alpha_K}_2 q_2' \wedge q_1' \,\mathcal{R}\, q_2'$.
- We define $\mathcal{R}' \subseteq (Q_1 \times Q_E) \times Q_2$ as follows:

$$(q_1, q_E) \,\mathcal{R}'\, q_2 \triangleq q_1 \,\mathcal{R}\, q_2$$

- Let us show that this relation has the right properties. Obviously, $(q_1^0, q_E^0) \,\mathcal{R}'\, q_2^0$.
- Now, suppose that $q_1 \xrightarrow{\alpha_K}_1 q_1'$ and $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}$. We have to show that there exists q_2' such that $q_2 \xrightarrow{\alpha_K}_2 q_2'$ and for any q_E' such that $q_E \xrightarrow{\alpha_E}_E q_E'$, it holds that $(q_1', q_E') \,\mathcal{R}'\, q_2'$.
- The q_2' provided by $\mathcal{R}$ (as $q_1 \,\mathcal{R}\, q_2$ and $q_1 \xrightarrow{\alpha_K}_1 q_1'$) is a solution. Indeed, it is such that $q_2 \xrightarrow{\alpha_K}_2 q_2'$ and $q_1 \,\mathcal{R}\, q_2$. So, if there exists q_E' such that $q_E \xrightarrow{\alpha_E}_E q_E'$, then by definition $(q_1', q_E') \,\mathcal{R}'\, q_2'$.

Step 2: $K_1 \sqsubseteq^4_{E,\mathcal{I}} K_2 \implies K_1 \sqsubseteq^3_{E,\mathcal{I}} K_2$.
- Suppose $K_1 \sqsubseteq^4_{E,\mathcal{I}} K_2$. According to the characterization of $\sqsubseteq^4$ provided by Lemma 6.2.11, there exists $\mathcal{R} \subseteq (Q_1 \times 2^{Q_E}) \times Q_2$ such that $(q_1^0, \{q_E^0\}) \,\mathcal{R}\, q_2^0$ and: if $(q_1, \mathcal{Q}_E) \,\mathcal{R}\, q_2$ and $q_1 \xrightarrow{\alpha_K}_1 q_1'$, there exists q_2' such that: (1) $q_2 \xrightarrow{\alpha_K}_2 q_2'$; (2) if there exists α_E, $q_E \in \mathcal{Q}_E$ and $q_E' \in Q_E$ such that $q_E \xrightarrow{\alpha_E}_E q_E'$ and $\alpha_K \cup \alpha_E \in \mathcal{I}$, then $(q_1', \mathcal{Q}_E') \,\mathcal{R}\, q_2'$ where $\mathcal{Q}_E'$ is $\{q_E' \mid \exists q_E \in \mathcal{Q}_E \; s.t. \; q_E \xrightarrow{\alpha_E}_E q_E'\}$.
- We define $\mathcal{R}' \subseteq (Q_1 \times Q_E) \times Q_2$ as follows:

$$(q_1, q_E) \,\mathcal{R}'\, q_2 \triangleq \exists \mathcal{Q}_E.\; q_E \in \mathcal{Q}_E \wedge (q_1, \mathcal{Q}_E) \,\mathcal{R}\, q_2$$

- Let us show that this $\mathcal{R}'$ has the expected properties.
- First, as $(q_1^0, \{q_E^0\}) \,\mathcal{R}\, q_2^0$, we have by definition $(q_1^0, q_E^0) \,\mathcal{R}'\, q_2^0$.
- Now suppose that $(q_1, \mathcal{Q}_E) \,\mathcal{R}\, q_2$ and $q_1 \xrightarrow{\alpha_K}_1 q_1'$. The q_2' provided by $\mathcal{R}$ is the one we are looking for, as it ensures: from (1), that $q_2 \xrightarrow{\alpha_K}_2 q_2'$; from (2), that if there exists α_E and $q_E' \in Q_E$ such that $q_E \xrightarrow{\alpha_E}_E q_E'$ and $\alpha_K \cup \alpha_E \in \mathcal{I}$, then $(q_1', \mathcal{Q}_E') \,\mathcal{R}\, q_2'$ where $\mathcal{Q}_E'$ is $\{q_E' \mid \exists q_E \in \mathcal{Q}_E \; s.t. \; q_E \xrightarrow{\alpha_E}_E q_E'$. This, in particular, implies that $q_E' \in \mathcal{Q}_E'$, hence $(q_1', q_E') \,\mathcal{R}\, q_2'$.

<table>
<tr><td>i \ j</td><td>1</td><td>4</td><td>3</td><td>2</td></tr>
<tr><td>1</td><td colspan="4">Step 1</td></tr>
<tr><td>4</td><td rowspan="3">Step 5</td><td>✓</td><td colspan="2">Step 2</td></tr>
<tr><td>3</td><td rowspan="2">Step 5</td><td>✓</td><td>Step 3</td></tr>
<tr><td>2</td><td>Step 4</td><td>✗</td></tr>
</table>

Table 7.1 – Structure of the proof of Theorem 7.1.3

Step 3: $K_1 \sqsubseteq^3_{E,\mathcal{I}} K_2 \implies K_1 \sqsubseteq^2_{E,\mathcal{I}} K_2$.
This has already been proven in Chapter 4: indeed, this corresponds to the coherence condition between $\preccurlyeq$ and $\sqsubseteq$ as they are defined in the contract framework for the semantic level of BIP. □

Theorem 7.1.3 *$K \sqsubseteq^i_{A,\mathcal{I}} G$ and $E \sqsubseteq^j_{G,\mathcal{I}} A$ implies $K \sqsubseteq^i_{E,\mathcal{I}} G$ iff i or j is not* 2.

Proof. The structure of the proof is presented in Table 7.1. It has already been proved that circular reasoning is sound for $\sqsubseteq^4$ and for $\sqsubseteq^3$. Besides, we have two counterexamples (see Figure 7.1) showing it is not sound for $\sqsubseteq^2$. We prove the rest in five steps.

Step 1: for any $j \in \{1, 4\}$, $K \sqsubseteq^1_{A,\mathcal{I}} G$ and $E \sqsubseteq^j_{G,\mathcal{I}} A$ implies $K \sqsubseteq^1_{E,\mathcal{I}} G$.
- The reason for that is very simple: $K \sqsubseteq^1_{A,\mathcal{I}} G \iff K$ simulates $G \iff K \sqsubseteq^1_{E,\mathcal{I}} G$.

Lemma 7.5.1 *If pseudo-circular reasoning is sound for $\sqsubseteq^\alpha$ and $\sqsubseteq^\beta$, and if $\sqsubseteq^\gamma \implies \sqsubseteq^\beta$, then pseudo-circular reasoning is also sound for $\sqsubseteq^\alpha$ and $\sqsubseteq^\gamma$.*

The proof of this lemma is trivial: suppose that $K \sqsubseteq^\alpha_{A,\mathcal{I}} G$ and $E \sqsubseteq^\gamma_{G,\mathcal{I}} A$. Then $E \sqsubseteq^\beta_{G,\mathcal{I}} A$, thus by pseudo-circular reasoning $K \sqsubseteq^\alpha_{E,\mathcal{I}} G$.

Step 2: We have shown in Chapter 6 that pseudo-circular reasoning is sound for $\sqsubseteq^{L1}$ (that is, $\sqsubseteq^4$) and $\sqsubseteq^{L0}$. According to Lemma 7.5.1, we only have to prove that $\sqsubseteq^2 \implies \sqsubseteq^{L0}$ to have soundness of pseudo-circular reasoning for $\sqsubseteq^4$ and $\sqsubseteq^2$, and as $\sqsubseteq^3 \implies \sqsubseteq^2$ (Theorem 7.1.2) also for $\sqsubseteq^4$ and $\sqsubseteq^3$. Now, remember that $K_1 \sqsubseteq^2_{E,\mathcal{I}} K_2$ is defined as $\mathcal{I}\{K_1, E\}$ simulates $\mathcal{I}\{K_2, E\}$. This naturally implies that $Tr(\mathcal{I}\{K_1, E\}) \subseteq Tr(\mathcal{I}\{K_2, E\})$ (see Property 1.1.13), which is how $K_1 \sqsubseteq^{L0}_{E,\mathcal{I}} K_2$ is defined.

Lemma 7.5.2 ***Characterization of*** $\sqsubseteq^2$

$K_1 \sqsubseteq^2_{E,\mathcal{I}} K_2$ if and only if there exists a relation $\mathcal{R} \subseteq (Q_1 \times Q_E) \times Q_2$ such that:

- $(q_1^0, q_E^0)\, \mathcal{R}\, q_2^0$
- *If $(q_1, q_E)\, \mathcal{R}\, q_2$, $(q_1, q_E) \xrightarrow{\alpha} (q_1', q_E')$ and $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}$, then $\exists q_2'$ s.t. $(q_2, q_E) \xrightarrow{\alpha} (q_2', q_E')$ and $(q_1', q_E')\, \mathcal{R}\, q_2'$.*

Step 3: ***Pseudo circular reasoning is sound for $\sqsubseteq^3$ and $\sqsubseteq^2$.***

- We suppose that $K \sqsubseteq^3_{A,\gamma} G$ and $E \sqsubseteq^2_{G,\gamma} A$ and we prove that $K \sqsubseteq^3_{E,\gamma} G$.
- As $K \sqsubseteq^3_{A,\gamma} G$, there exists a relation $\mathcal{R}_3$ on $(Q_K \times Q_A) \times Q_G$ as ni
- As $E \sqsubseteq^2_{G,\gamma} A$, there exists a simulation relation $\mathcal{R}_2$ on $(Q_E \times Q_G) \times Q_A$ as in Lemma 7.5.2.
- We define $\mathcal{R} \subseteq (Q_K \times Q_E) \times Q_G$ as follows:

$$(q_K, q_E)\, \mathcal{R}\, q_G \triangleq \exists q_A \in Q_A \text{ such that } (q_K, q_A)\, \mathcal{R}_3\, q_G \text{ and } (q_E, q_G)\, \mathcal{R}_2\, q_A$$

- We have to prove that $\mathcal{R}$ ensures the conditions of Definition 4.2.3. Obviously, $(q_K^0, q_E^0)\, \mathcal{R}\, q_G^0$.
- Let $q_K \in Q_K, q_E \in Q_E, q_G \in Q_G$ be such that $(q_K, q_E)\, \mathcal{R}\, q_G$. Let q_A be such that $(q_K, q_A)\, \mathcal{R}_3\, q_G$ and $(q_E, q_G)\, \mathcal{R}_2\, q_A$.
- Suppose $q_K \xrightarrow{\alpha_K}_K q_K'$ and $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}$. We have to prove that there exists q_G' such that:
  1. $q_G \xrightarrow{\alpha_K}_G q_G'$
  2. $\forall q_E',\ q_E \xrightarrow{\alpha_E}_E q_E' \implies (q_K', q_E')\, \mathcal{R}\, q_G'$
- As $(q_K, q_A)\, \mathcal{R}_3\, q_G$, $q_K \xrightarrow{\alpha_K}_K q_K'$ and $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}$, we know that there exists q_G' such that:
  - $q_G \xrightarrow{\alpha_K}_G q_G'$
  - $\forall q_A',\ q_A \xrightarrow{\alpha_E}_A q_A' \implies (q_K', q_A')\, \mathcal{R}_1\, q_G'$
- We show that this q_G' satisfies the conditions above for $\mathcal{R}$.
- The first condition is exactly the same as for $\mathcal{R}_3$. Let us show that the second condition holds.
- Suppose that $q_E \xrightarrow{\alpha_E}_E q_E'$. We show that $(q_K', q_E')\, \mathcal{R}\, q_G'$.
- Because $(q_E, q_G)\, \mathcal{R}_2\, q_A$ and $(q_E, q_G) \xrightarrow{\alpha} (q_E', q_G')$, there exists q_A' such that $(q_A, q_G) \xrightarrow{\alpha} (q_A', q_G')$ and $(q_E', q_G')\, \mathcal{R}_2 q_A'$.
- This implies in particular that $q_A \xrightarrow{\alpha_E}_A q_A'$.
- Thus, applying the second property offered by $\mathcal{R}_3$ to this q_A', we obtain that $(q_K', q_A')\, \mathcal{R}_1\, q_G'$.
- Finally, according to the definition of $\mathcal{R}$, we can conclude that $(q_K', q_E')\, \mathcal{R}\, q_G'$.

Step 4: ***Pseudo circular reasoning is sound for $\sqsubseteq^2$ and $\sqsubseteq^3$.***

- We suppose that $K \sqsubseteq^2_{A,\gamma} G$ and $E \sqsubseteq^3_{G,\gamma} A$ and we prove that $K \sqsubseteq^2_{E,\gamma} G$.

- As $K \sqsubseteq^2_{A,\gamma} G$, there exists a relation $\mathcal{R}_2$ on $(Q_K \times Q_A) \times Q_G$ as in Lemma 7.5.2.
- As $E \sqsubseteq^3_{G,\gamma} A$, there exists a simulation relation $\mathcal{R}_3$ on $(Q_E \times Q_G) \times Q_A$ as in Definition 4.2.3.
- We define $\mathcal{R} \subseteq (Q_K \times Q_E) \times Q_G$ as follows:

$$(q_K, q_E)\,\mathcal{R}\,q_G \triangleq \exists q_A \in Q_A \text{ such that } (q_K, q_A)\,\mathcal{R}_2\,q_G \mathrm{and} (q_E, q_G)\,\mathcal{R}_3\,q_A$$

- We have to prove that $\mathcal{R}$ ensures the conditions of Lemma 7.5.2. Obviously, $(q^0_K, q^0_E)\,\mathcal{R}\,q^0_G$.
- Let $q_K \in Q_K, q_E \in Q_E, q_G \in Q_G$ be such that $(q_K, q_E)\,\mathcal{R}\,q_G$ and $(q_K, q_E) \xrightarrow{\alpha} (q'_K, q'_E)$ with $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}$.
- We have to prove that there exists q'_G such that $(q_G, q_E) \xrightarrow{\alpha} (q'_G, q'_E)$ and $(q'_K, q'_E)\,\mathcal{R}\,q'_G$.
- As $(q_K, q_E) \xrightarrow{\alpha} (q'_K, q'_E)$, we have $q_E \xrightarrow{\alpha_E}_E q'_E$. Then, as $(q_E, q_G)\,\mathcal{R}_3\,q_A$, we know that there exists q'_A such that:
  - $q_A \xrightarrow{\alpha_E}_A q'_A$
  - $\forall q'_G,\ q_G \xrightarrow{\alpha_K}_G q'_G \implies (q'_E, q'_G)\,\mathcal{R}_2\,q'_A$
- We fix this q'_A. From the first item above, and because $(q_K, q_E) \xrightarrow{\alpha} (q'_K, q'_E)$ implies that $q_K \xrightarrow{\alpha_K}_K q'_K$, we get that $(q_K, q_A) \xrightarrow{\alpha} (q'_K, q'_A)$.
- As $(q_K, q_A)\,\mathcal{R}_2\,q_G$, this implies that there exists q'_G such that $(q_G, q_A) \xrightarrow{\alpha} (q'_G, q'_A)$ and $(q'_K, q'_A)\,\mathcal{R}_2\,q'_G$.
- Thus, applying the second property offered by $\mathcal{R}_3$ to this q'_G, we obtain that $(q'_E, q'_G)\,\mathcal{R}_2\,q'_A$.
- Finally, according to the definition of $\mathcal{R}$, we can conclude that $(q'_K, q'_E)\,\mathcal{R}\,q'_G$.

Step 5: This step is a direct application of Lemma 7.5.1. □

Theorem 7.1.5 *$K_1 \sqsubseteq^i K_2$ if and only if K_1 simulates K_2, for any $i \in \{1,4\}$*

Proof.

$\Longleftarrow$ The proof that simulation implies refinement in any context is trivial. Indeed, if K_1 simulates K_2, then by definition for any $E, \mathcal{I}$, $K_1 \sqsubseteq^1_{E,\mathcal{I}} K_2$, which according to Theorem 7.1.2 implies that $K_1 \sqsubseteq^i_{E,\mathcal{I}} K_2$ for any $i \in \{1,4\}$.

$\Longrightarrow$ Now let us consider the other implication. Let K_1, K_2 be two LTS on $\mathcal{P}$. We only need to prove this implication for $\sqsubseteq^2$, because $K_1 \sqsubseteq^i_{E,\mathcal{I}} K_2 \implies K_1 \sqsubseteq^4_{E,\mathcal{I}} K_2$ for any $i \in \{1,4\}$.

Suppose that for any context $(E, \mathcal{I})$ for $\mathcal{P}$, $K_1 \sqsubseteq^2_{E,\mathcal{I}} K_2$. Let us prove that K_1 simulates K_2. We proceed in two steps: we define a context $(E_\bot, \mathcal{I}_\bot)$ which corresponds to an environment that does nothing. As our composition of LTS can only restrict behavior, such an environment is in fact the "least helpful" with respect to refinement under context. We then prove that refinement in the context of $(E_\bot, \mathcal{I}_\bot)$ implies simulation.

Let $\mathcal{P}$ be the interface of K_1 and K_2. We define $E_\bot$ as $(\{q_\bot\}, q_\bot, \emptyset, \emptyset)$, that is, an LTS with only one state and no transition. We define $\mathcal{I}_\bot$ as $2^{\mathcal{P}}$, that is, all interactions of K_1 and K_2 are allowed. We have $K_1 \sqsubseteq^2_{E_\bot,\mathcal{I}_\bot} K_2$, so according to Lemma 7.5.2, there exists a relation $\mathcal{R} \subseteq (Q_1 \times \{q_\bot\}) \times Q_2$ such that:

- $(q_1^0, q_\bot) \,\mathcal{R}\, q_2^0$
- If $(q_1, q_\bot) \,\mathcal{R}\, q_2$ and $(q_1, q_\bot) \xrightarrow{\alpha} (q_1', q_\bot)$, then there exists q_2' such that $(q_2, q_\bot) \xrightarrow{\alpha} (q_2', q_\bot)$ and $(q_1', q_\bot) \,\mathcal{R}\, q_2'$.

Define $\mathcal{R}' \subseteq Q_1 \times Q_2$ by: $q_1 \,\mathcal{R}'\, q_2 \triangleq (q_1, q_\bot) \,\mathcal{R}\, q_2$. Then $\mathcal{R}'$ is obviously a simulation relation. □

Theorem 7.2.4 *Modal refinement $\preceq$ corresponds to refinement in any context.*

Proof. This proof is inspired by that of the corresponding theorem for LTS. Let K_1 and K_2 be two components on the same interface $\mathcal{P}$.

$\Longleftarrow$ Suppose $K_1 \preceq K_2$. Let $(E, \mathcal{I})$ be a context for $\mathcal{P}$. We show that $K_1 \sqsubseteq_{E,\mathcal{I}} K_2$.

By definition of $\preceq$, there exists a relation $\mathcal{R}' \subseteq Q_1 \times Q_2$ such that $q_1^0 \, \mathcal{R}' \, q_2^0$ and for any pair $(q_1, q_2) \in Q_1 \times Q_2$, whenever $q_1 \, \mathcal{R}' \, q_2$, the following holds:

- for any $q_1' \in Q_1$, $q_1 \overset{\alpha}{\dashrightarrow} q_1'$ implies that there exists $q_2' \in Q_2$ such that $q_2 \overset{\alpha}{\dashrightarrow} q_2'$ and $q_1' \, \mathcal{R}' \, q_2'$
- for any $q_2' \in Q_2$, $q_2 \overset{\alpha}{\longrightarrow} q_2'$ implies that there exists $q_1' \in Q_1$ such that $q_1 \overset{\alpha}{\longrightarrow} q_1'$ and $q_1' \, \mathcal{R}' \, q_2'$

We define $\mathcal{R} \subseteq (Q_1 \times Q_E) \times Q_2$ as: $(q_1, q_E) \, \mathcal{R} \, q_2 \triangleq q_1 \, \mathcal{R}' \, q_2$. Let us show that this relation is a refinement under context as in Definition 7.2.3. It trivially holds that $(q_1^0, q_E^0) \, \mathcal{R} \, q_2^0$. Now, let $q_1 \in Q_1$, $q_2 \in Q_2$ and $q_E \in Q_E$ be such that $(q_1, q_E) \, \mathcal{R} \, q_2$. We have to prove that:

1. if $q_1 \overset{\alpha_K}{\dashrightarrow}_1 q_1'$ and $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}$, then there exists q_2' such that $q_2 \overset{\alpha_K}{\dashrightarrow}_2 q_2'$ and any q_E' such that $q_E \overset{\alpha_E}{\dashrightarrow}_E q_E'$ satisfies $(q_1', q_E') \, \mathcal{R} \, q_2'$
2. if $q_2 \overset{\alpha_K}{\longrightarrow}_1 q_2'$ and $\alpha = \alpha_K \cup \alpha_E \in \mathcal{I}$, then there exists q_1' such that $q_1 \overset{\alpha_K}{\longrightarrow}_1 q_1'$ and any q_E' such that $q_E \overset{\alpha_E}{\longrightarrow}_E q_E'$ satisfies $(q_1', q_E') \, \mathcal{R} \, q_2'$

Those conditions are obviously satisfied according to the definitions of $\mathcal{R}$ and $\mathcal{R}'$.

$\Longrightarrow$ Now let us consider the other implication. Let K_1, K_2 be two LTS on $\mathcal{P}$. Suppose that for any context $(E, \mathcal{I})$ for $\mathcal{P}$, $K_1 \sqsubseteq_{E,\mathcal{I}}^2 K_2$. Let us prove that $K_1 \preceq K_2$.

We proceed in two steps: we define a context $(E_\perp, \mathcal{I}_\perp)$ which corresponds to an environment that does not interfere with the execution of the component. We then prove that refinement in the context of $(E_\perp, \mathcal{I}_\perp)$ implies modal refinement.

We define $E_\perp$ as $(\{q_\perp\}, q_\perp, \emptyset, \emptyset, \emptyset)$, that is, an LTS with only one state and no transition. We define $\mathcal{I}_\perp$ as $2^{\mathcal{P}}$, that is, all interactions of K_1 and K_2 are allowed. We have $K_1 \sqsubseteq_{E_\perp, \mathcal{I}_\perp} K_2$, so there exists a relation $\mathcal{R} \subseteq (Q_1 \times \{q_\perp\}) \times Q_2$ such that $(q_1^0, q_\perp) \, \mathcal{R} \, q_2^0$ and where $(q_1, q_\perp) \, \mathcal{R} \, q_2$ implies:

1. if $(q_1, q_\perp) \overset{\alpha}{\dashrightarrow} (q_1', q_\perp)$, then there exists q_2' such that $(q_2, q_\perp) \overset{\alpha}{\dashrightarrow} (q_2', q_\perp)$ and $(q_1', q_\perp) \, \mathcal{R} \, q_2'$
2. if $(q_2, q_\perp) \overset{\alpha}{\longrightarrow} (q_2', q_\perp)$, then there exists q_1' such that $(q_1, q_\perp) \overset{\alpha}{\longrightarrow} (q_1', q_\perp)$ and $(q_1', q_\perp) \, \mathcal{R} \, q_2'$

Define $\mathcal{R}' \subseteq Q_1 \times Q_2$ by: $q_1 \, \mathcal{R}' \, q_2 \triangleq (q_1, q_\perp) \, \mathcal{R} \, q_2$. Then $\mathcal{R}'$ is obviously a modal refinement. □

Theorem 7.2.6 *$\mathcal{C}$ is* consistent with *$\mathcal{I}_I$ and $\{\mathcal{P}_i\}$ if and only if $\mathcal{I}_I\{G_1, \ldots, G_n\} \models \mathcal{C}$.*

Proof.

$\Longleftarrow$ This is a direct application of the definition of structural consistency.

$\Longrightarrow$ Suppose that $\mathcal{I}_I\{G_1, \ldots, G_n\} \not\models \mathcal{C}$. Let us show that there exist no $K_1, \ldots, K_n$ such that $\mathcal{I}_I\{K_1, \ldots, K_n\} \models \mathcal{C}$. Denote $G_\pi = \mathcal{I}_I\{G_1, \ldots, G_n\}$. We denote by Q_π the set of states of G_π.

Define $\mathcal{R}' \subseteq (Q_\pi \times Q_A) \times Q_G$ as $(q_\pi, q_A)\, \mathcal{R}'\, q_G \triangleq \forall i \in [1, n] : q_G \in \mathcal{Q}^i_\pi$ where $q_\pi = (\mathcal{Q}^1_\pi, \ldots, \mathcal{Q}^n_\pi)$. We show that if $\mathcal{R}'$ is not a refinement under context, then no relation can be.

Observation 1. All *must*-transitions in G have a counterpart in $G_\pi = \mathcal{I}_I\{G_1, \ldots, G_n\}$.
Suppose that $q_G \xrightarrow{\alpha_K}_G q'_G$. Let us show that there exists q'_π such that $q_\pi \xrightarrow{\alpha_K}_\pi q'_\pi$. As $(q_\pi, q_A)\, \mathcal{R}'\, q_G$, $q_G \in \mathcal{Q}^i_\pi$ for all i. Denote $\alpha_i = \alpha_K \cap \mathcal{P}_i$ for $i \in [1, n]$. By definition of projection, if $\alpha_i \neq \emptyset$, then $\mathcal{Q}^i_\pi \overset{\alpha_i}{\dashrightarrow}_\pi \mathcal{Q}^{i'}_\pi$ where $q'_G \in \mathcal{Q}^{i'}_\pi$; if $\alpha_i = \emptyset$, then $q'_G \in \mathcal{Q}^i_\pi$. By definition of composition using $\mathcal{I}_I$, this implies that $(\mathcal{Q}^1_\pi, \ldots, \mathcal{Q}^n_\pi) \xrightarrow{\alpha_K}_\pi (\mathcal{Q}^{1'}_\pi, \ldots, \mathcal{Q}^{n'}_\pi)$ where $\mathcal{Q}^{i'}_\pi = \mathcal{Q}^i_\pi$ for i such that $\alpha_i = \emptyset$. Moreover, $q'_\pi = (\mathcal{Q}^{1'}_\pi, \ldots, \mathcal{Q}^{n'}_\pi)$ is such that $(q'_\pi, q_A)\, \mathcal{R}'\, q'_G$.

Observation 2. The *may* transition relation is equal to the *must* transition relation in all G_i, hence in G_π. The conclusion from this observation is that if $\mathcal{R}'$ is not a refinement under context, then the only possible reason for that is that some *must*-transition of G_π has no *may*-counterpart in G. Let us focus on this situation. Suppose that $(q_\pi, q_A)\, \mathcal{R}'\, q_G$ and $q_\pi \xrightarrow{\alpha}_\pi q'_\pi$ and $q_G \overset{\alpha}{\not\dashrightarrow}_G$. We know that:

- $\forall i : q_G \in \mathcal{Q}^i_\pi$
- by definition of projection, there exists q and q' such that $q \xrightarrow{\alpha}_G q'$ and for all $i \in [1, n]$, $q \in \mathcal{Q}^i_\pi$ and $q' \in Q^{i'}_\pi$.

States in the same set $\mathcal{Q}^i_\pi$ are indistinguishable by component G_i and by any other component on $\mathcal{P}_i$ according to the definition of composition. So, a refinement under context would have to be a relation making q_π unreachable. This is however not possible because all combinations of states in q_π are reachable by a sequence of *must*-transitions starting in the initial state, which has to be preserved by refinement. □

7.6 Conclusion

7.6.1 Summary

The motivation for our work was not directly to help system designers by providing a specific contract framework and tools supporting them but rather to help the designers of contract frameworks by introducing a notion of contract framework that should be as general as possible and provide rules for reasoning with contracts depending on properties ensured by the component framework on top of which the contract framework is built.

The basic notion of contract framework poses the minimal conditions for soundness of circular reasoning, which allows any component in a system to rely on assumptions of its environment without the risk of introducing inconsistency. By loosening the relation between refinement in a closed system and refinement in a given context, we have defined for several component frameworks a refinement relation such that circular reasoning can be applied even though the usual refinement does not permit it. Also, we have provided a generalized version of circular reasoning based on two refinement relations.

We use the structural part of contracts to define refinement under context, but also to provide structural counterparts to properties such as consistency and compatibility. Besides, we have deliberately avoided to use composition of contracts and shown that it is possible to reason about contracts without ever composing them.

The top-down design and verification methodology that we have proposed can be applied to systems of arbitrary size. Our abstract definition of component framework encompasses a large class of formalisms and interaction models: I/O automata, several variants of BIP, HRC L0 and L1. Our encoding of I/O interface automata emphasizes the interest of splitting proofs between generic concepts (circular reasoning, preservation of refinement by composition etc.) and ad-hoc definitions. The three contract frameworks based on the BIP component framework make it clear that providing a compositional semantics that is consistent with flattening is essential. Finally, our encoding of the HRC L0 and L1 frameworks provide a formal basis for the integration of their respective tool chains in the SPEEDS project using combined circular reasoning.

7.6.2 Perspectives

There is no application of the frameworks defined in this thesis. Our goal here was rather to build a formal and abstract methodology for building contracts frameworks. We have indeed applied our methodology to a concrete verification problem for a protocol in tree-like networks. This problem

requires data transformation and data transfer as well as safety and some progress requirements. Thus, a rather ad-hoc framework has to be defined, and we have shown how the use of our approach to avoid as much as possible reasoning about the actual framework. The interested reader is referred to [BHQG10c].

We have only hinted at the possibility of using multiple contracts and multiple hierarchical decompositions. This area of research is very promising and research as well as experiments are required to invent heuristics for taking advantage of such features.

One question raised in this work concerns what to do if an assumption cannot be discharged in a dominance check. Modifying either this assumption or the other contracts is one possibility, but there are others. For example, if one is able to identify prefixes of the executions leading to a violation of the assumption, one could check precisely the possible executions of the system after this violation. One could also estimate or compute the probability that such a violation takes place. In other words, it would be interesting to consider an approach in which contracts may be violated but a diagnostic is then provided. Note that this suggestion is different from using probabilistic contracts: the idea here is rather to use a non-probabilistic framework and to work with probabilities only for specific executions.

Part II

Implementation of Distributed Systems with Complex Interaction

Chapter 8

Achieving distributed control through model checking

The practical motivation for the work presented in this part of the thesis is the distributed implementation of BIP systems. We have shown in Part I that component frameworks which offer a complex interaction layer (such as BIP) can be a powerful tool for design and verification of large systems, because they allow enforcing some properties structurally. There remains the question of how such systems can be implemented in a distributed setting. Although the formalism used in Part II is different from that of Part I — we consider processes rather than components, and their behaviors are not transition systems but Petri nets — the connection between the two parts should be clear.

8.1 Preliminaries

Before presenting a generalization of the support policy introduced in [BBPS09], we need to define some concepts related to Petri nets, distributed control and knowledge.

8.1.1 Petri nets

We represent distributed systems as Petri nets, but the method and algorithms developed here can equally apply to other models, e.g., communicating automata or transition systems, which form the basis of the formalism used in [GPQ10].

Definition 8.1.1 *A* Petri net N *is a tuple* (P, T, E, s_0) *where*

– P *is a finite set of* places. *The set of* states *(markings) is defined as* $S = 2^P$.

- T *is a finite set of* transitions.
- $E \subseteq (P \times T) \cup (T \times P)$ *is a bipartite relation between the places and the transitions.*
- $s_0 \subseteq 2^P$ *is the* initial state *(initial marking).*

For a transition $t \in T$, we define the set of *input places* ${}^\bullet t$ as $\{p \in P|(p,\, t) \in E\}$, and the set of *output places* $t^\bullet$ as $\{p \in P|(t,\, p) \in E\}$.

Definition 8.1.2 *A transition t is* enabled *in a state s if ${}^\bullet t \subseteq s$ and $(t^\bullet \backslash {}^\bullet t) \cap s = \emptyset$. We denote the fact that t is enabled from s by $s[t\rangle$.*

A state s is in *deadlock* if there is no enabled transition from it.

Definition 8.1.3 *A transition t can be* fired *(*executed*) from state s to state s', which is denoted by $s[t\rangle s'$, when t is enabled in s. Then, $s' = (s \backslash {}^\bullet t) \cup t^\bullet$.*

We will use the Petri net of Figure 8.1 as a running example. As usual, transitions are represented as lines, places as circles, and the relation E as a set of arrows from transitions to places and from places to transitions. The Petri net of Figure 8.1 has places named $p_1, p_2, \ldots, p_8$ and transitions $a, b, \ldots, e$. We represent a state by putting *tokens* inside the places of this state. In the example of Figure 8.1, the depicted initial state s_0 is $\{p_1, p_4\}$. The transitions enabled in s_0 are a and b. Firing a from s_0 means removing the token from p_1 and adding one in p_3.

Definition 8.1.4 *An* execution *is a maximal (i.e., it cannot be extended) alternating sequence of states and transitions $s_0 t_1 s_1 t_2 s_2 \ldots$ with s_0 the initial state of the Petri net, such that for each state s_i in the sequence, $s_i[t_{i+1}\rangle s_{i+1}$.*

We denote the executions of a Petri net N by $exec(N)$. The prefixes on the executions in a set X are denoted by $pref(X)$. A state is *reachable* in a Petri net if it appears in at least one of its executions. We denote the reachable states of a Petri net N by $reach(N)$. The reachable states of our running example N are: $\{p_1, p_2\}$, $\{p_1, p_4\}$, $\{p_2, p_3\}$, $\{p_3, p_4\}$, $\{p_5, p_6\}$, $\{p_5, p_8\}$, $\{p_6, p_7\}$ and $\{p_7, p_8\}$.

8.1.2 Constraints

The constraints that we want to enforce in a distributed way are of the form $\Psi \subseteq S \times T$ for a given Petri net. That is, Ψ defines which transition may be fired in which global state. Note that there may be states in which no transition is enabled with respect to Ψ. Let (N, Ψ) be the pair made of a Petri net N and the constraint Ψ that we want to enforce.

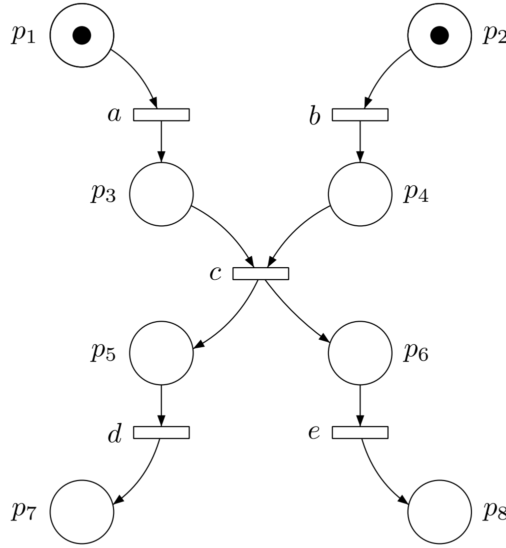


Figure 8.1 – A Petri net with initial state $\{p_1, p_2\}$

Definition 8.1.5 *A transition t of N is* enabled with respect to Ψ *in a state s if $s[t\rangle$ and, furthermore, $(s, t) \in \Psi$. An* execution *of (N, Ψ) is a maximal prefix $s_0 t_1 s_1 t_2 s_2 t_3 \ldots$ of an execution of N such that for each state s_i in the sequence, $(s_i, t_{i+1}) \in \Psi$.*

We denote the executions of (N, Ψ) by $exec(N, \Psi)$, and the set of states reachable in these executions by $reach(N, \Psi)$. We assume that those sets are nonempty. Clearly, $reach(N, \Psi) \subseteq reach(N)$ and $exec(N, \Psi) \subseteq pref(exec(N))$; recall that restricting N according to Ψ may introduce deadlocks. The problem we want to solve is as follows:

> Given a Petri net with a constraint (N, Ψ), we want to obtain a Petri net N' such that $exec(N') \subseteq exec(N, \Psi)$. In particular, this means that the states in $reach(N')$ which are deadlocks of N' must be deadlocks in $reach(N, \Psi)$ or be empty.

It is possible to avoid the situation where (N, Ψ) introduces deadlocks which are not in N. This is achieved by computing based on Ψ a global constraint Ψ' such that: (1) the executions of (N, Ψ') satisfy Ψ and (2) Ψ' does not introduce deadlocks with respect to N because it blocks transitions which lead to states in which the system cannot progress without violating Ψ. As already mentioned, this can be done using game theory [Tho95].

We are interested in particular in enforcing priority orders. Indeed, a priority order $\ll$ is a partial order relation among the transitions T of N and thus defines a constraint Ψ in a straightforward

manner. Given a priority order $\ll$, Ψ is defined by: $(s, t) \in \Psi$ if and only if t is enabled in s and has a maximal priority among the transitions enabled in s. That is, there is no other transition r with $s[r\rangle$ such that $t \ll r$. We write $exec(N, \ll)$ and $reach(N, \ll)$ instead of $exec(N, \Psi)$ and $reach(N, \Psi)$, respectively. Note that priority orders do not introduce new deadlocks, and thus we have $exec(N, \ll) \subseteq exec(N)$.

Let us now consider the Petri net of Figure 8.1 and the priorities $a \ll b$ and $d \ll e$. When the priorities are *not* taken into account, there are four different executions of N, namely $abcde$, $bacde$, $abced$ and $baced$ (states are abstracted away). However, when taking the priorities into account, there is only one execution left: $baced$. Thus, priorities are used in this context for scheduling purposes.

8.1.3 Distributed setting

Definition 8.1.6 *A* process *of a Petri net N is a subset of the transitions $\pi \subseteq T$.*

We assume a given set of processes Π_N that covers all the transitions of net N, i.e., $\bigcup_{\pi \in \Pi_N} \pi = T$. A transition can belong to several processes, e.g., when it models a synchronization between processes. Note that we do not require our processes to be sequential.

The neighborhood of a process π describes the part of the system whose state π can observe at any given moment. Our definition is only one among others, and our results still apply to other definitions of neighborhood. In particular, it may be more realistic not to consider outputs places as part of the neighborhood, but only input places.

Definition 8.1.7 *The* neighborhood $ngb(\pi)$ *of a process π is the set of places $\bigcup_{t \in \pi}({}^\bullet t \cup t^\bullet)$.*

Definition 8.1.8 *The* local state *of a process π of a Petri net N in a state s is $s|_\pi = s \cap nbg(\pi)$.*

That is, the local state of a process π in a global state s consists of the restriction of s to the neighborhood of π. It describes what π can see of s based on its limited view.

Definition 8.1.9 *Define an equivalence relation $\equiv_\pi \subseteq S \times S$ such that $s \equiv_\pi s'$ when $s|_\pi = s'|_\pi$.*

Figure 8.2 represents one possible distribution of our running example. We represent processes by drawing dashed lines between them. Here, the left process π_l consists of transitions a, c and d while the right process π_r consists of transitions b, c and e. The neighborhood of π_l contains all the places of the Petri net except p_2 and p_8. The local state $s_0|_{\pi_l}$ corresponding to the initial state $s_0 = \{p_1, p_2\}$ is $\{p_1\}$. Note that the local state $s|_{\pi_l}$ corresponding to $s = \{p_1, p_8\}$ is also $\{p_1\}$, hence $s_0 \equiv_{\pi_l} s$.

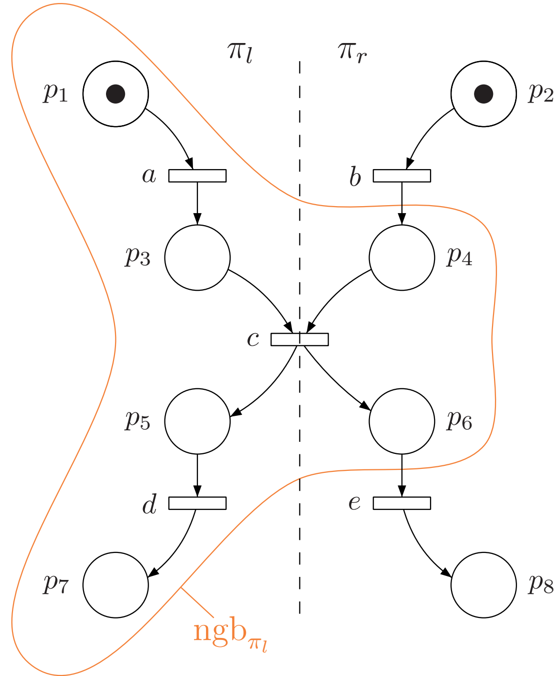


Figure 8.2 – A distributed Petri net with two processes π_l and π_r

It is easy to see that the enabledness of a transition depends only on the local state of a process that contains it, i.e., if $t \in \pi$ and $s \equiv_\pi s'$ then $s[t\rangle$ if and only if $s'[t\rangle$. This means that executing the unconstrained system in a distributed way only requires a mechanism for handling conflicting transitions, as each process locally knows which of its transitions are enabled in a given state. Thus, the only difficulty is when looking for a mechanism to safely execute the system in a distributed way lies in handling the additional constraint.

8.1.4 Defining properties

We use places also as state predicates where $s \models p_i$ if and only if $p_i \in s$. This is extended to Boolean combinations on such predicates in a standard way. For a state s, we denote by φ_s the formula that is a conjunction of the places that are in s and the negated places that are not in s. Thus, φ_s is satisfied by state s and by no other state. For the Petri net of Figure 8.2, the initial state s is characterized by $\varphi_s = p_1 \wedge p_2 \wedge \neg p_3 \wedge \neg p_4 \wedge \neg p_5 \wedge \neg p_6 \wedge \neg p_7 \wedge \neg p_8$. For a set of states $Q \subseteq S$, we can write a *characteristic formula* $\varphi_Q = \bigvee_{s \in Q} \varphi_s$ or use any equivalent propositional formula.

Definition 8.1.10 *A formula φ is an* invariant *of a Petri net N if $s \models \varphi$ for each $s \in reach(N)$, i.e., if φ holds in every reachable state.*

We define the following formulae representing sets of states as explained above. These formulae will be useful for implementing our distributed controller.

- $\varphi_{reach(N)}$: all the reachable states of N.
  Similarly, $\varphi_{reach(N,\Psi)}$ denotes the reachable states of (N, Ψ).
- $\varphi_{en(t)}$: the states in which transition t is enabled.
- $\varphi_{\Psi(t)}$: the states s in which transition t is enabled and $(s, t) \in \Psi$.
  Formally: $\varphi_{\Psi(t)} = \varphi_{en(t)} \wedge \bigvee_{(s,t)\in\Psi} \varphi_s$.
- φ^{Ψ}_{df}: the reachable states in which at least one transition is enabled w.r.t. Ψ, i.e., the reachable states which are deadlock-free w.r.t. Ψ.
  Formally: $\varphi^{\Psi}_{df} = \varphi_{reach(N,\Psi)} \wedge \bigvee_{t\in T} \varphi_{\Psi(t)}$.
- $\varphi_{s|_\pi}$: the states in which the local state of process π is $s|_\pi$.

We can perform model checking in order to calculate these formulae, and store them in a compact way, e.g., using BDDs. For Ψ representing priority constraints, we denote $\varphi_{\Psi(t)}$ by $\varphi_{max(t)}$: it corresponds to the states in which transition t has a maximal priority among all the enabled transitions of the system. That is, $\varphi_{max(t)} = \varphi_{en(t)} \wedge \bigwedge_{t\ll r} \neg\varphi_{en(r)}$.

8.1.5 Knowledge

Our approach for a local or semi-local decision on firing transitions is based on the *knowledge* of processes [FHVM95]. Basically, the knowledge of a process in a given global state is the set of reachable global states that are consistent with the local state of that process. For example, in the initial state represented in Figure 8.2, the left process π_l *knows* that the current global state is $\{p_1, p_2\}$, because it is the only *reachable* state that projects onto $\{p_1\}$. Indeed, neither $\{p_1, p_8\}$, nor $\{p_1\}$ nor $\{p_1, p_2, p_8\}$ are reachable. In fact, in this example, both processes always know the exact global state of the system based on their local state.

Definition 8.1.11 *A process π* knows *a (Boolean) property φ in a state s, denoted $s \models K_\pi\varphi$, exactly when for each reachable s' such that $s \equiv_\pi s'$, we have that $s' \models \varphi$.*

We obtain immediately from the definitions that if $s \models K_\pi\varphi$ and $s \equiv_\pi s'$, then $s' \models K_\pi\varphi$. Furthermore, the process π knows φ in state s exactly when $(\varphi_{reach(N)} \wedge \varphi_{s|_\pi}) \Rightarrow \varphi$ is a tautology. Given a Petri net and a Boolean property φ, one can perform model checking in order to decide whether $s \models K_\pi\varphi$.

The definitions of neighborhood, local state, equivalence and knowledge all naturally extend to sets of processes: given a set of processes $\Pi \subseteq \Pi_N$, we define $ngb(\Pi) = \bigcup_{\pi \in \Pi} ngb(\pi)$. The other definitions follow immediately. Note that a *joint* local state, that is, the local state of a set of processes Π, is equivalently represented as a tuple consisting of the local states of the processes in Π.

8.2 The support policy

In this section, we generalize the *support policy* introduced in [BBPS09] for priorities to any constraint Ψ of the form presented before. This method uses model checking to analyze the system and identify when a process can decide, based only on its local state, whether some enabled transition is allowed by Ψ or should be blocked. The support policy is based on a *support table* Δ which indicates, for each process in each local state, which transitions are *supported* and may thus be safely fired. The basic principle of the support policy is the following:

> In a state s, a transition t is *supported* by a process π containing t if and only if π knows in s (based on its limited view of the system) about (s, t) respecting Ψ, i.e.: $s \models K_\pi \varphi_{\Psi(t)}$. A transition can be fired in a state only if, in addition to its original enabledness condition, at least one of the processes containing it supports it.

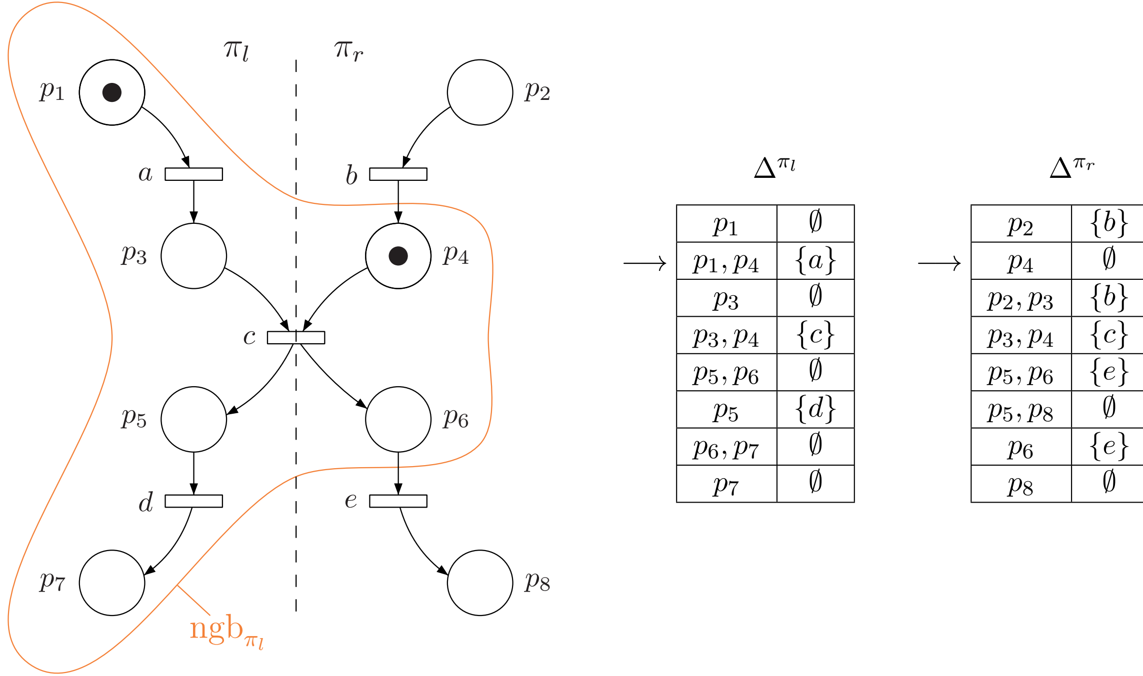
The disjunctive nature of the controller that will result from this policy appears in the fact that a transition needs only one local controller to support it in order to be fired.

8.2.1 Building the support table

Given a Petri net N and a constraint Ψ, the corresponding support table Δ is built as follows: we check for each process π, reachable state $s \in reach(N)$ and transition $t \in \pi$, whether $s \models K_\pi \varphi_{\Psi(t)}$. If it holds, we put in the support table at the entry $s|_\pi$ the transitions t that are responsible for satisfying this property. In fact, as $s \models K_\pi \varphi$ and $s \equiv_\pi s'$ implies that $s' \models K_\pi \varphi$, it is sufficient to check this for a single representative state containing $s|_\pi$ out of each equivalence class of '$\equiv_\pi$'. The construction of the support table is simple and its size is limited to the number of different local states of the process and not to the (sometimes exponentially larger) size of the state space.

The support table Δ corresponding to our running example is shown in Figure 8.3. Δ is split into two parts, one per process. The arrows point to the entries in the table corresponding to the global state represented on the left. In this state, process π_r does not support any of its transitions as none of them is enabled. On the other hand, process π_l supports a because it knows that the global state is

$\{p_1, p_4\}$, hence it knows that b is not enabled. Note that in this example a local state corresponds to exactly one global state, but this is in general not the case, as will be illustrated in the following.



Δ^{π_l}

p_1	$\emptyset$
p_1, p_4	$\{a\}$
p_3	$\emptyset$
p_3, p_4	$\{c\}$
p_5, p_6	$\emptyset$
p_5	$\{d\}$
p_6, p_7	$\emptyset$
p_7	$\emptyset$

Δ^{π_r}

p_2	$\{b\}$
p_4	$\emptyset$
p_2, p_3	$\{b\}$
p_3, p_4	$\{c\}$
p_5, p_6	$\{e\}$
p_5, p_8	$\emptyset$
p_6	$\{e\}$
p_8	$\emptyset$

Figure 8.3 – A Petri net with priorities $a \ll b$ and $d \ll e$ along with its support table Δ

8.2.2 Distributed control based on the support table

We use the support table Δ to control (restrict) the executions of N so as to satisfy Ψ. Each process π of N (that is, $\pi \in \Pi_N$) is equipped with the entries of this table of the form $s|_\pi$ for s a reachable state. Before firing a transition, process π consults the entry $s|_\pi$ that corresponds to its current local state, and supports only the transitions that appear in that entry. This can be represented as an extended Petri net N^Δ, as we explain now.

For simplicity of the transformation, we consider extended Petri nets [GL81], where processes may have local variables, and transitions have an enabling condition and a data transformation.

Definition 8.2.1 *An* extended Petri net *has, in addition to the Petri net components, for each process $\pi \in \Pi_N$ a finite set of variables V_π and (1) for each variable $v \in V_\pi$, an initial value v_0 (2) for each transition $t \in T$, an enabling condition en_t and a transformation predicate f_t on the variables*

$V_t = \bigcup_{\pi \in proc(t)} V_\pi$, *where* $proc(t)$ *is the set of processes to which* t *belongs. In order to fire* t, en_t *must hold in addition to the usual Petri net enabling condition on the input and output places of* t. *When* t *is executed, in addition to the usual changes to the tokens, the variables in* V_t *are updated according to* f_t.

A Petri net N' *extends* N if N' is an extended Petri net obtained from N according to Definition 8.2.1. The comparison between the original Petri net N and N' extending it is based only on places and transitions. That is, we ignore (project out) the additional variables.

Lemma 8.2.2 *For a Petri net* N' *extending* N, $exec(N') \subseteq pref(exec(N))$.

Proof. The extended Petri net N' only strengthens the enabling conditions and gives values to the added variables, thus it can only restrict the executions. However, these restrictions may result in new deadlocks. □

We have the following monotonicity property.

Theorem 8.2.3 *Let* N *be a Petri net and* N' *an extension of* N. *If* $s \models K_\pi\varphi$ *in* N, *then* $s \models K_\pi\varphi$ *also in* N'.

Proof. The extended Petri net N' restricts the set of executions, and possibly the set of reachable states, of N. Each local state $s|_\pi$ is part of fewer global states, and thus the knowledge in $s|_\pi$ can only increase. □

Monotonicity is important to ensure Ψ in N^Δ. Indeed, the knowledge allowing to enforce Ψ by the imposed transformation is calculated based on N, but is used to control the execution of the transitions of N^Δ. Monotonicity thus ensures the correctness of N^Δ with respect to Ψ.

The extended Petri net N^Δ is obtained from N and Ψ by defining the additional condition en_t for an enabled transition t to be fired as: $\bigvee_{\pi \in proc(t)} K_\pi\Psi(t)$. That is, t can be fired if is supported by at least one process containing it. The knowledge properties calculated in Δ are encoded in the variables and updated as transitions are fired. Note that N^Δ is indeed a controlled version of N, as it can only restrict the executions of N. It is distributed, because one set of variables per process is used to define the additional enabledness conditions. Only variables of processes involved in a transition t can be used to determine whether t can be fired or not, and only those variables are updated when t is fired. Finally, it is disjunctive, because a transition can be fired if at least one process supports it.

Note that defining controllers as extended Petri nets allows the use of some finite memory that is updated with the execution of observable transitions. This can be useful, e.g., when constructing a controller based on knowledge with perfect recall [vdM98]. However, a controller based on simple knowledge, as in Definition 8.1.11, does not have to exercise this capability.

8.2.3 Deadlock-freedom

The extended Petri net N^{Δ} obtained from N and Ψ obviously enforces Ψ, since only supported transitions are fired, and only transitions which are known to be enabled with respect to Ψ are supported. However, N^{Δ} does not ensure that no deadlock is added with respect to (N, Ψ). If N^{Δ} does not introduce any deadlock with respect to (N, Ψ), we say that it *implements* (N, Ψ). We now focus on the issue of determining whether an extended Petri net N^{Δ} implements (N, Ψ) or not.

Definition 8.2.4 *We define the following properties k_i^{π}:*

- $k_1^{\pi} = \bigvee_{t \in \pi} K_{\pi} \Psi(t)$*: process π can identify a transition t such that it knows that t is enabled with respect to Ψ.*
- $k_2^{\pi} = \neg k_1^{\pi} \wedge K_{\pi} \bigvee_{\rho \neq \pi} k_1^{\rho}$*: process π does not know whether it has a transition with maximal priority, but in all the global states s' with $s'|_{\pi} = s|_{\pi}$ some other process ρ is in a local state where k_1^{ρ} holds. This allows π to remain inactive without risk of introducing a deadlock.*
- $k_3^{\pi} = \neg k_1^{\pi} \wedge \neg k_2^{\pi}$*: π does not know whether or not there is a supported transition.*

k_1^{π} can be extended to sets of processes: $k_1^{\Pi} = \bigvee_{t \in T_{\Pi}} K_{\Pi} \Psi(t)$, where $T_{\Pi} = \bigcup_{\pi \in \Pi} \pi$.

Note that $k_1^{\pi} \vee k_2^{\pi} \vee k_3^{\pi} \equiv true$.

The construction in [BBPS09] checks whether $\bigvee_{\pi \in \Pi} k_1^{\pi}$ holds in all reachable states of the original system that are not deadlock (or termination). If so, it is sufficient that each process supports a transition when it knows that $\Psi(t)$ holds in order to enforce the additional constraint Ψ (in that case, priority) without introducing any additional deadlock. What to do when this check fails is the object of the next chapter.

Chapter 9

A synchronization-based approach

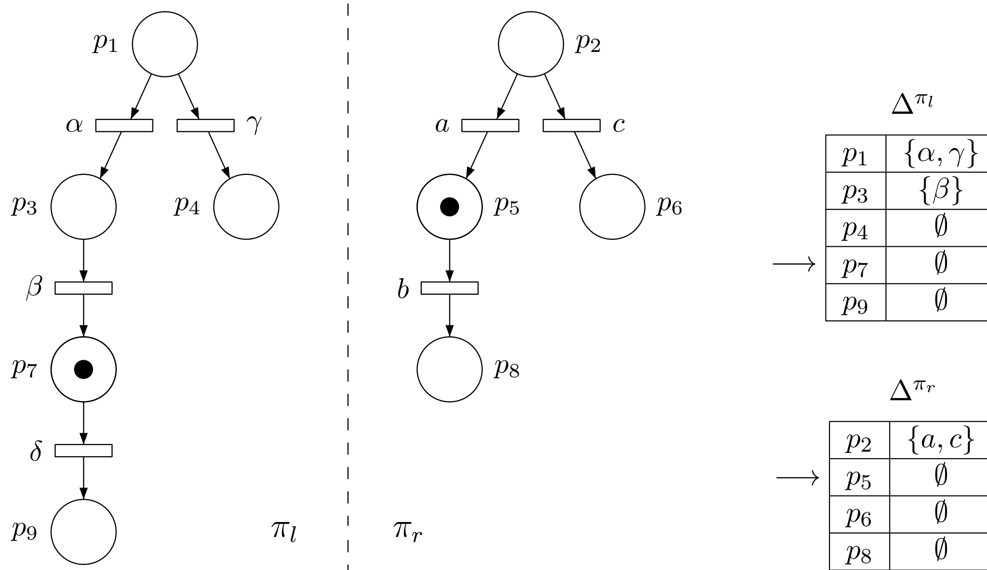
9.1 A synchronization-based approach

It is not possible in general to decide, based only on the local state of a process or a set of processes, whether some enabled transition is allowed by Ψ. Thus, there are cases where the support policy as introduced in the previous chapter fails, because N^{Δ} has more deadlocks than (N, Ψ). Before discussing existing solutions to this and presenting a new one, let us look at one example where this situation arises.

9.1.1 An example where the support policy fails

Consider a concurrent system as in Figure 9.1, with two processes π_l (left) and π_r (right) with disjoint sets of transitions, each one of them having initially a nondeterministic choice. The priorities in this system are $\delta \ll b \ll \beta$. Each process can observe only its own transitions.

In the initial state, all four enabled transitions α, γ, a, c are unordered by priorities, and thus all are maximal. If α is fired and subsequently a (or vice versa), we reach a global state where process π_r does not have any enabled transition with maximal priority since $b \ll \beta$. Process π_l does, and it can execute β. Thereafter, since $\delta \ll b$, process π_l cannot execute δ and must wait for process π_r to execute b. Now, with its limited observability, π_l cannot distinguish between the situation before or after b was executed by π_r. Thus π_l lacks the capability, and the corresponding knowledge, of deciding whether to execute δ. In this state, π_r cannot distinguish between the situation before and after β was executed, and cannot decide to execute b. Accordingly, the local knowledge of the processes in this example is not sufficient to construct a controller. In the initial state, both processes can progress



Δ^{π_l}

p_1	$\{\alpha, \gamma\}$
p_3	$\{\beta\}$
p_4	$\emptyset$
$\longrightarrow$ p_7	$\emptyset$
p_9	$\emptyset$

Δ^{π_r}

p_2	$\{a, c\}$
$\longrightarrow$ p_5	$\emptyset$
p_6	$\emptyset$
p_8	$\emptyset$

Figure 9.1 – A Petri net with priorities $\delta \ll b \ll \beta$

freely, but then reach a situation where they do not know locally when they can safely progress.

9.1.2 Existing solutions

To handle situations where the support policy fails, two suggestions have been made:

1. Use knowledge of perfect recall [vdM98, BBPS09]. This means that the knowledge is not based only on the local state, but also on the limited history that each process can observe. Although the history is not finitely bounded, it is enough to calculate the set of states where the rest of the system can reside at each point. A subset construction can be used to supply for each process an automaton that is updated according to the local history. This construction is very expensive: the size of this automaton can be exponential in the number of global states. Furthermore, although in this way we extend our knowledge (by separating local states according with different histories), this still does not guarantee that a distributed controller can be found. In particular, knowledge of perfect recall is useless in the situation of Figure 9.1.
2. Combine the knowledge of some processes together by synchronizing them [BBPS09]. The definition of knowledge can be used for sets of processes rather than individual processes. With their combined knowledge, one can achieve more situations where the maximal priority transition is known. However, to use this knowledge at runtime, these sets of processes need to

be able to access their joint local state. This means synchronizing them, at the cost of losing concurrency. At the limit, all processes can be combined, and no concurrency remains.

9.1.3 Adding synchronizations to provide sufficient knowledge

Instead of the fixed synchronization between processes suggested in [BBPS09], we propose to use *temporary* synchronizations: processes coordinate to achieve *joint* knowledge (i.e. knowledge of a set of processes), whenever their local knowledge is not sufficient to ensure deadlock-freedom. This does not reduce the concurrency as much as the previous method, but induces some communication overhead as the temporary synchronizations are achieved through exchange of messages.

We now calculate the support table Δ iteratively, first adding entries corresponding to local states as in Chapter 8, then (joint) local states of pairs of processes, then triples etc. At each stage of the construction, Δ is identified with its set of (non-empty) entries, which are (joint) local states $s|_\Pi$ satisfying k_1^Π — that is, local states in which the joint knowledge of the processes in Π is sufficient to ensure progress. Remember that the joint local state of a set of processes Π can be seen as a tuple consisting of the local states of the processes in Π.

Definition 9.1.1 *A set of (joint) local states Δ is an* invariant *if each non-deadlock reachable global state contains at least one (joint) local state from Δ.*

The first iteration includes in Δ, for every $\pi \in \Pi_N$, the singleton local states satisfying k_1^π, i.e. states in which progress of π is guaranteed. With each entry corresponding to such a local state $s|_\pi$, we associate the actual transitions t that make k_1^π hold.

If Δ is an invariant, then the method presented in the previous chapter is sufficient to build N^Δ implementing (N, Ψ). However, if Δ is not an invariant, then we need to consider joint local states.

Definition 9.1.2 *A* synchronization state *is a reachable global state in which* none *of the corresponding individual local states satisfies k_1^π.*

The existence of a synchronization state means that Δ is not an invariant without adding some tuples for synchronization.

We first calculate for each local state not satisfying k_1^π whether it satisfies k_2^π. Let U_π be the set of local states of process π satisfying k_3^π, that is, satisfying neither k_1^π nor k_2^π. Now, in a second iteration, we add to Δ pairs $(s_\pi, s_\rho) \in U_\pi \times U_\rho$ for $\pi \neq \rho$ if there exists a synchronization state s such that $s|_\pi = s_\pi$, $s|_\rho = s_\rho$ and furthermore $s \models k_1^{\{\pi,\rho\}}$. Again, we associate with that entry of the table Δ

the transitions t that witness the satisfaction of $k_1^{\{\pi,\rho\}}$. The second iteration terminates as soon as Δ is an invariant or if all such pairs of local states have been classified.

In a third iteration, we consider triples of local states from $U_\pi \times U_\rho \times U_\sigma$ such that no subtuple is in Δ, and so forth. Eventually, Δ becomes an invariant, in the worst case when synchronization states themselves are added to Δ: indeed, synchronizing all the processes ensures that any transition enabled and allowed by Ψ in such a state will be supported. Our construction guarantees that each time the transition associated with a tuple $(s|_{\pi_1} \ldots s|_{\pi_k})$ from Δ is executed from a state that includes these local components, the constraint Ψ we want to impose is preserved.

If we go back to our example, the support table presented in Figure 9.1 must be enriched by the entries of Table 9.1. Note that in this example, there is one entry in the support table per synchronization state. This is in general not the case for systems with more than two processes because one synchronization may be sufficient to ensure progress in several synchronization states. Also, during execution of our example, a synchronization may take place only when the system has reached a synchronization state. This again is not the case in general. Indeed, as will be illustrated later, two processes may decide to synchronize because they both know that a synchronization state may have been reached, although this is actually not the case.

Δ^{π_l,π_r}

p_5, p_7	$\{b\}$
p_7, p_8	$\{\delta\}$

Table 9.1 – Additional entries for the support table of Figure 9.1 to become an invariant

9.1.4 A distributed controller imposing the global property

We now have to explain how the joint local knowledge used to enforce the invariance of Δ is achieved in practice. Indeed, the method proposed in the previous chapter for building the extended Petri net N^Δ from N and Ψ does not apply directly. The reason is that joint knowledge cannot be expressed by disjoint sets of variables. We solve this by adding synchronizations amongst the processes involved.

Such synchronizations are achieved by using an algorithm like α-core [PCT04], which allows processes to notify, using asynchronous message passing, a set of coordinators about their wish to be involved in a joint action. Once a coordinator has been notified by all the participants in the synchronization it is in charge of, it checks whether conflicting synchronizations are already under way

(a process may have notified several coordinators). If this is not the case, coordination succeeds, and the synchronization can take place. We assume that the correctness of the algorithm guarantees the atomic-like behavior of the coordination process, allowing us to reason at a higher level of abstraction where we treat the synchronizations provided by α-core (or any similar algorithm) as transitions that are joint between several participating processes.

Thus, if a transition t is associated with a singleton element $s|_\pi$ in Δ, then the controller for π, in local state $s|_\pi$, supports t. Otherwise, t is associated with a tuple of local states in Δ; when reaching any of these local states, the corresponding processes $\pi_1 \dots \pi_k$ try to achieve a synchronization using the coordination algorithm. If coordination succeeds, and the synchronization takes place, the associated transition t is then supported by all the participating processes (there may be several such transitions). Formally, for each transition t associated with a tuple of local states $(s|_{\pi_1} \dots s|_{\pi_k})$, we execute a transition enabled exactly in the global states containing this tuple and performing the original transformation of t.

9.1.5 Minimizing the number of coordinators

It is wasteful to set up one coordination for each joint local state involving at least two processes in Δ. We now show how to minimize the number of coordinators for pairs of the form $(s|_\pi, r|_\rho)$ in Δ. The general version of this method for larger tuples is analogous. We denote by $\Delta_{\pi,\rho}$ the set of pairs of Δ made of a local state from process π and one from process ρ.

A naive implementation may use a coordination for every pair in Δ. Nevertheless, the large number of messages needed to implement coordination by an algorithm like α-core suggests that we minimize their number. The opposite extreme would be to use a unique coordination between every two processes π and ρ. However, as α-core does not offer guarded coordinations, success of a coordination does not imply in this case that the resulting synchronization will be useful. Thus, many (expensive) useless coordinations may be achieved, not even guaranteeing eventual progress.

We propose an intermediate solution. Consider now a set of pairs $\Gamma \subseteq \Delta_{\pi,\rho}$ such that if $(s,r),(s',r') \in \Gamma$, then also $(s,r'),(s',r) \in \Gamma$ (s and s' do not have to be disjoint, and neither do r and r'). This means that Γ is a complete bipartite subgraph of $\Delta_{\pi,\rho}$. It is sufficient to generate one coordination for all the pairs in Γ: upon success of the coordination, the precalculated table $\Delta_{\pi,\rho}$ will be consulted about which transition to allow, depending on $s|_\pi$ and $s|_\rho$. Thus, according to this strategy, a sufficient number of coordinations is formed by finding a covering partition $\Gamma_1,\dots,\Gamma_m$ of complete bipartite subgraphs of $\Delta_{\pi,\rho}$. That is, each pair $(s|_\pi, r|_\rho) \in \Delta_{\pi,\rho}$ must be in some set Γ_i. However, the minimization problem for such a partition turns out to be in NP-Complete, as stated in

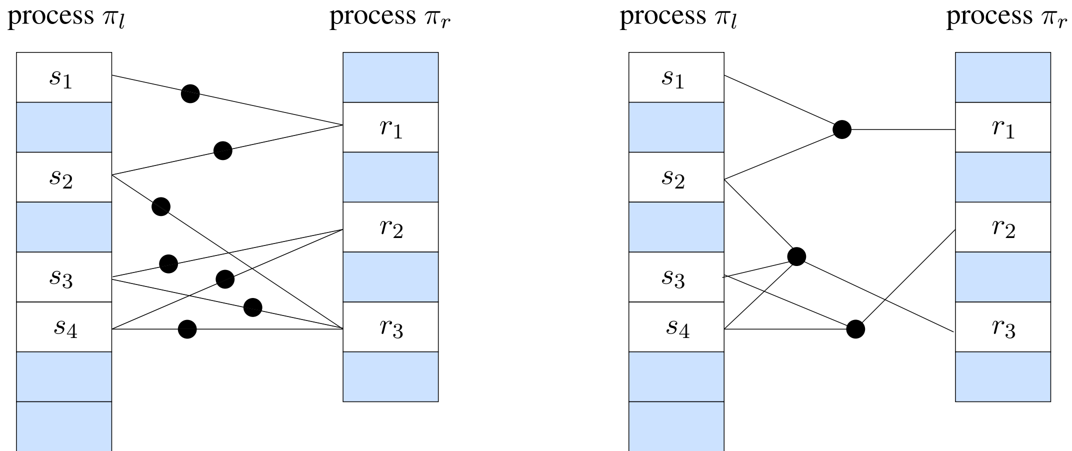


Figure 9.2 – Minimizing the number of coordinators

the following theorem.

Property 9.1.3 [Orl77] *Given a bipartite graph $G = (N, E)$ and a positive integer $K \leq |E|$, finding whether there exists a set of subsets $N_1, \ldots N_k$ for $k \leq K$ of complete bipartite subgraphs of G such that each edge (u, v) is in some N_i is in NP-Complete.*

We use the following notation: when Γ is a set of pairs of local states, one of π and the other of ρ, we denote by $\Gamma|_\pi$ and by $\Gamma|_\rho$ the π and the ρ components in these pairs, respectively. We suppose that $|\Delta_{\pi,\rho}|_\pi| \leq |\Delta_{\pi,\rho}|_\rho|$, i.e., the number of elements paired up in $\Delta_{\pi,\rho}$ is smaller for π than for ρ. If this is not the case, one simply has to replace π with ρ and vice versa in the sequel. We apply the following heuristics to calculate a (not necessarily minimal) set of complete bipartite subsets $\Gamma_i \subseteq \Delta_{\pi,\rho}$ covering $\Delta_{\pi,\rho}$.

Let the elements of $\Delta_{\pi,\rho}|_\pi$ (that is, the π components of the pairs in $\Delta_{\pi,\rho}$) be $x_1, \ldots, x_m$. We start with a first partition $\Gamma_1, \ldots, \Gamma_m$ where Γ_i is the set of pairs in $\Delta_{\pi,\rho}$ containing x_i, for any $i \in [1, m]$. These sets are obviously complete, and the partition is a covering.

Now, in order to refine this partition, we check for each two sets Γ_i and Γ_j whether $\Gamma_i|_\rho = \Gamma_j|_\rho$. If it is the case, we merge them into a single set $\Gamma_i \cup \Gamma_j$. The resulting set is complete because it contains a pair (x_i, y) if and only if it also contains (x_j, y), where $y \in \Delta_{\pi,\rho}|_\rho$. Note that each x_i always appears in exactly one subgraph, thus we cannot repeat the process for π.

Figure 9.2 shows how this heuristics works for an example. Lines represent pairs of local states must be synchronized; blacks dots represent coordinators. The left-hand side of the figure shows the

coordinators induced by $\Delta_{\pi,\rho}$ and the right-side the minimal set of coordinators that is obtained by our heuristics. We start with process π_r: each Γ_i contains a single state of π_r. In this case, the initial partition turns out to be already the solution. Note that starting with process π_l would also result in a solution with three coordinators, because the coordinators for s_3 and s_4 can be merged.

9.2 Implementation and experimental results

We have implemented a prototype for experimenting with this approach. This tool first builds the set of reachable states and the corresponding local knowledge of each process. Then, it checks whether local knowledge is sufficient to ensure correct distributed execution of the system under study. We allow simulating the system while counting the number of synchronizations and synchronization states encountered during execution as a measurement of the amount of additional synchronization required.

9.2.1 The pragmatic dining philosophers

The example that we used in our experiments is a variant of the dining philosophers where philosophers may arbitrarily take first either the fork that is on their left or right, provided it is on the table (see Figure 9.3). In addition, a philosopher may hand over a fork to one of his neighbors when his second fork is not available and the neighbor is looking for a second fork as well. Such an exchange (labeled ex) is a way to avoid the well-known deadlocks when all philosophers hold one fork in their left (respectively right) hand: our philosophers are pragmatic enough to exchange forks when they have nothing better to do. This example is partially represented by the Petri net of Figure 9.4.

In our example, places (concerning philosopher β) are defined as follows:

- $fork^i$: the i-th fork is on the table.
- $0fork_\beta$ (respectively $2forks_\beta$): philosopher β has no fork (respectively 2 forks) in his hands.
- $1fork^l_\beta$ (respectively $1fork^r_\beta$): philosopher β holds his left (respectively right) fork.

Transitions (concerning philosopher β) play the following role:

- get^{kl}_β (respectively get^{kr}_β), $k = 1, 2$: philosopher β takes the fork on his left (respectively on his right). This is his k-th fork.
- *eat-and-return*$_\beta$: philosopher β eats and puts both forks back on the table.
- $ex_{\alpha,\beta}$: philosopher α gives his right fork to philosopher β.
- $ex_{\beta,\alpha}$: philosopher β gives his left fork to philosopher α .

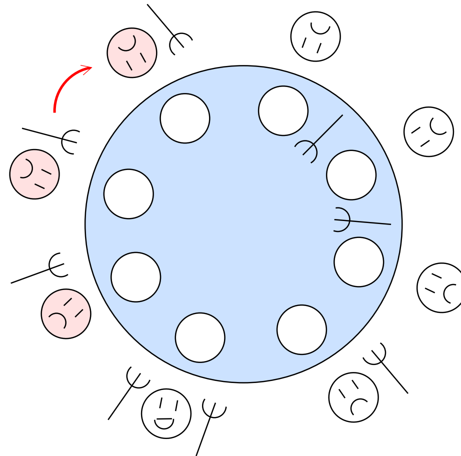


Figure 9.3 – The pragmatic dining philosophers

Processes correspond to philosophers. The transitions defining a process β are those with a β in their name, including the four exchange transitions $ex_{\alpha,\beta}$, $ex_{\beta,\alpha}$, $ex_{\beta,\gamma}$ and $ex_{\gamma,\beta}$. In Figure 9.4, transitions related only to philosopher β are in blue. Transitions in orange and green are shared between β and one of his neighbors (respectively α on the left and γ on the right).

Not controlling exchanges at all allows non-progress cycles, that is, philosophers exchanging forks without ever eating. To avoid this, we add priorities which allow exchange actions only when a blocking situation has been reached within some degree of locality.

First variant. We use a priority rule stating that an exchange between philosophers α and β has lower priority than α or β taking a fork. This leads to the following priorities for each α and β such that α is βs left neighbor:

- $ex_{\alpha,\beta} \ll get^{2l}_{\alpha}$: if α can pick up a left fork, he may not give his right fork to β.
- $ex_{\beta,\alpha} \ll get^{2r}_{\beta}$: symmetrically if β can pick up a right fork.

In this variant, local knowledge is sufficient. Indeed, when philosopher β and both his neighbors are blocked in a state where they all have a left (respectively a right) fork, then philosopher β has enough knowledge to support an exchange with his left (respectively right) neighbor — because he knows that he has nothing better to do. For any number of philosophers, there is no synchronization state. Thus, no extra synchronization is needed.

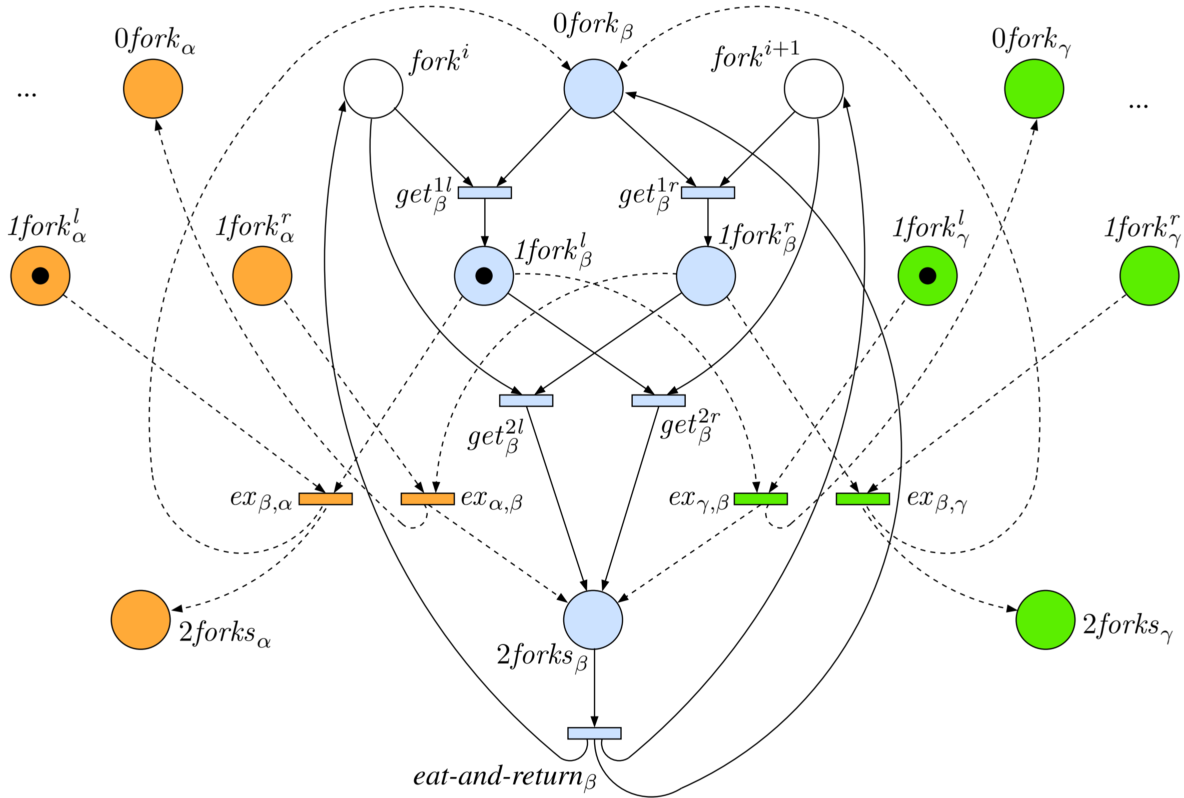


Figure 9.4 – A partial representation of the dining philosophers (philosopher β)

Second variant. Now, to further reduce the number of exchanges, one may decide that philosopher β may give his left fork to his left neighbor α only if (1) α is blocked (2) β is blocked and (3) βs right neighbor γ is also blocked (the exchange of right forks is similar). This is situation represented in Figure 9.3. This translates into adding the following priorities:

– $ex_{\alpha,\beta} \ll get_\delta^{2l}$, *eat-and-return*$_\delta$ (with δ the left neighbor of philosopher α)
– $ex_{\beta,\alpha} \ll get_\gamma^{2r}$, *eat-and-return*$_\gamma$ (with γ the right neighbor of philosopher β)

Local knowledge alone cannot ensure here correct distributed execution. However, binary synchronizations are sufficient in this example to ensure that the system is always able to move on, for any number of philosophers.

In Table 9.2, we show results for the second variant with 6, 8 and 10 philosophers. In all cases, there are two synchronization states which correspond to the situation where all philosophers hold their left fork, or they all hold their right fork.

philosophers	6		8		10	
strategy	min	average	min	average	min	average
reachable states	729		6561		59049	
synchronizations	322	354	229	285	178	237
synchronization states encountered		253		149		100

Table 9.2 – Results for 100 executions of 10,000 steps for the second variant

For computing the number of synchronizations, we used each time 100 runs of a length of 10,000 steps (i.e. transitions). Note that the number of exchange transitions is identical to the number of synchronizations. We provide results according to two different strategies:

1. Synchronizations are allowed only when no other transition is supported.
2. Synchronizations and supported transitions have the same probability.

The first strategy (denoted *min* in Table 9.2) cannot be distributed and only aims at simulating how many synchronizations are needed to escape the synchronization states encountered during execution. The second strategy (denoted *average*) is implementable in a distributed setting. As one can see, this strategy increases the number of synchronizations taking place (because a synchronization can take place as soon as the philosophers involved in it all believe that a synchronization state may have been reached), but allows reaching synchronization states less often. Thus, the communication overhead induced by synchronizations which are unnecessary with respect to deadlock freedom is compensated by the added degree of progress achieved by the system.

9.2.2 Of tracks and trains

Another example that we worked with is taken from [BBPS09]. This example is interesting in so far it shows the limit of an approach to distributed control that only aims at avoiding deadlocks.

In this example, trains enter and exit a train station, evolving between track segments. A track segment can accept a single train at a time, therefore there must be some mechanism to detect and resolve conflicts amongst trains trying to access the same track segment.

The Petri net associated with a given a set of trains K and a set of segments S has the following sets of places:

– $p_{outside-k}$ has a token when train k is outside the train station.
– $p_{k@s}$ has a token when train k is at segment s.
– $p_{empty-s}$ has a token when segment s is empty.

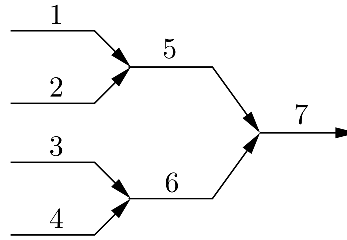


Figure 9.5 – Topology of the train station

Transitions represent a train entering or exiting the train station, or progressing from one track to another. We consider in our example the topology shown in Figure 9.5: a train entering via segment 1 can progress to segment 5 and then segment 7 before exiting the station again.

More generally, transitions are of the following form:

- Train k enters the train station at a segment $s \in \{1, 2, 3, 4\}$: input places are $p_{outside-k}$ and $p_{empty-s}$; the unique output place is $p_{k@s}$.
- Train k moves from segment s to segment s' for $(s, s') \in \{(1, 5), (2, 5), (3, 6),$ $(4, 6), (5, 7), (6, 7)\}$: input places are $p_{k@s}$ and $p_{empty-s'}$; output places are $p_{empty-s}$ and $p_{k@s'}$.
- Train leaves the station at segment $s = 7$: the unique input place is $p_{k@s}$ and output places are $p_{outside-k}$ and $p_{empty-s}$.

Unlike in [BBPS09], our processes are the trains instead of the segments. Thus, all transitions belong to exactly one process. The neighborhood of a process k — that is all input or output places of one of its transitions — consists of:

- $p_{empty-s}$ for any segment s
- $p_{k@s}$ for any segment s
- $p_{outside-k}$

Note that the places $p_{empty-s}$ are in the neighborhood of all processes while all other places are in the neighborhood of exactly one process. Intuitively, this means that a train can detect whether another train wants to access the same segment, but not which train it is.

We consider 3 types of trains: high-speed TGV trains, local trains and freight trains. Priorities are defined so that the high-speed TGV trains can progress more quickly than other trains, whereas freight trains can move only if they are not in conflict with any other train. More precisely:

- There is no priority to enter the system.

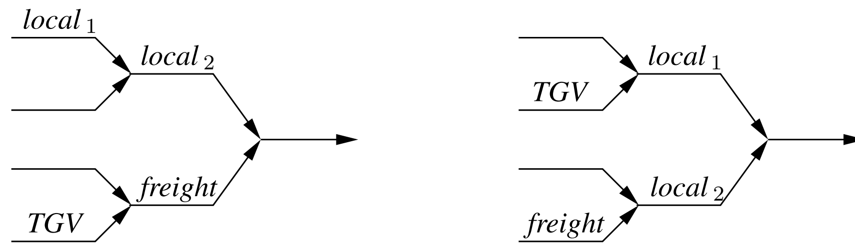


Figure 9.6 – Two examples of synchronization states

- If two trains want to access the same segment, then the one that has the highest priority will progress.
- Because of the topology of the train station, only one train at a time can exit the system, and no priorities are needed amongst exit transitions.

For the experiments, we have fixed the number of trains as follows: 1 TGV train, 2 local trains, and 1 freight train. Out of 1961 reachable states, 72 are synchronization states. They correspond to situations where the TGV is blocked by another train that cannot move on. Figure 9.6 shows two such cases. In the example on the left, only $local_2$ can move on. It is stuck however because it knows that there is a train on segment 6 but does not know which train. The example on the right is slightly different because both $local_1$ and $local_2$ can move on but are stuck because they do not know the type of each other.

In our example, binary synchronizations are sufficient to ensure that the system is always able to move on, and this also if we increase the number of trains. Synchronization states are handled differently depending on whether the TGV is blocked by the freight train or by a local train (say, $local_1$). In the first scenario represented in Figure 9.6, any of the following two synchronizations is sufficient: synchronization between $local_2$ and the TGV, or between $local_2$ and the freight train. Indeed, both synchronizations provide $local_2$ with enough knowledge to support its transition to segment 7. In the second scenario of Figure 9.6, any of the following three synchronizations is helpful: synchronization between the TGV and any local train, or between the local trains. Let us recall that synchronizations are defined between given local states of processes. This implies that, although all processes are willing to synchronize because they suspect a synchronization state, only synchronizations which lead to a transition being supported take place.

We have simulated this example 1,000 times for traces of length 10,000 steps (i.e. transitions). We observe that in most cases (785 out of 1,000 executions), no synchronization is needed at all.

The reason for this surprising result is that partial deadlocks are frequent in this system. Suppose, for example, that the freight train enters segment 1, and then $local_1$ enters segment 2. At this moment, neither $freight$ nor $local_1$ can move on anymore. Once such a state is reached, no synchronization state can be reached anymore because the remaining trains, namely the TGV and $local_2$, can go on forever on segments 3, 4, 6 and 7. Thus, deadlock-freedom is achieved here by blocking a part of the system so that the rest of it can go on forever without problems. We will discuss this issue later.

Chapter 10

Reducing the need for additional synchronizations

In this chapter, we look at the problem of reducing the need for additional synchronizations in order to control distributed systems. We identify a key issue of the knowledge approach, which is that knowledge is computed based on the original, uncontrolled system (N). In fact, this is merely an approximation, as the actual knowledge needs to be satisfied by the controlled system (N^{Δ}). After control has been applied, there are fewer executions, and fewer reachable states, hence the knowledge cannot decrease. This observation leads to the two following results.

The first result is somewhat surprising: we prove that it is safe to calculate knowledge by considering only the executions of the original system that satisfy the desired constraint, that is, the executions of (N, Ψ). This provides a smaller set of executions and reachable states, hence also potentially more knowledge.

A second result is that once we control a system according to its knowledge properties, we obtain again a system with fewer executions and reachable states: even if in the original system there are reachable states where the system lacks the knowledge to continue, these states may, in fact, already be unreachable in N^{Δ}. Thus, one needs to make another round of checks.

These two results can be used in conjunction with other methods for constructing distributed controllers based on knowledge:

– Using knowledge of perfect recall (proposed in [BBPS09]).
– Adding coordinations to combine knowledge (proposed in the previous chapter).

We show here that all these techniques are independent of each other, hence can be combined.

10.1 Alternative to adding synchronizations

In the following two sections, we develop the two results briefly described in the introduction.

10.1.1 Support policy based on the controlled system

The first technique is based on the following observation:

> Instead of calculating the knowledge with respect to *all* the executions of the original system, we may calculate it based on the executions of the original system that satisfy Ψ.

The states reachable in these executions represent a subset of the states reachable in the original system. Furthermore, for each local state, the set of global states containing it is contained in the corresponding set of the original Petri net. Thus, our knowledge in each global configuration may not decrease, but possibly grows. Still, we need to show that calculating knowledge using this set of executions produces a correct controller.

Theorem 10.1.1 *Let N be a Petri net and Ψ a property to be enforced. Let Δ be the support table calculated for $reach(N, \Psi)$, and let N^{Δ} be the extended Petri net constructed for Δ. Then $exec(N^{\Delta}) \subseteq pref(exec(N, \Psi))$.*

Proof. When a transition t of N^{Δ} is supported in some state s according to the support table Δ, then for some supporting process $\pi \in \Pi_N$, $s \models K_\pi \varphi_{\Psi(t)}$. By definition of the knowledge operator, this implies that $(s, t) \in \Psi$. Thus, each firing of a transition of N^{Δ} preserves Ψ. However, it is possible that at some point, there is not enough knowledge to support any transition. □

Note that this proof does not guarantee that N^{Δ} *implements* (N, Ψ), i.e., that $reach(N^{\Delta})$ does not contain deadlocks which are not in $reach(N, \Psi)$ because in some states not enough knowledge is available to support transitions.

Let $\varphi_{support(\pi)}$ denote the disjunction of the formulae $\varphi_{s|_\pi}$ such that the entry $s|_\pi$ is nonempty in the support table. A sufficient condition for N^{Δ} to implement (N, Ψ) is:

$$\varphi_{df}^{\Psi} \rightarrow \bigvee_{\pi \in \Pi} \varphi_{support(\pi)} \tag{10.1}$$

This condition requires that for each state in $reach(N, \Psi)$ that is not a deadlock, at least one transition is supported. When this condition does not hold, we say that the support table is *incomplete*.

Consider Petri net N_1 of Figure 10.1 with the given priority rules. The separation of transitions of N_1 according to processes is represented using dashed lines.

The example shows three processes π_1 (left), π_2 (in the middle), π_3 (right) that use binary synchronizations and priorities to enforce mutual exclusion for the execution of critical sections $(b_i r_i e_i)_{i \in [1,3]}$. Intuitively, priority rules $b_i \ll \{r_j, e_j\}$ and $r_i \ll e_j$ give higher priority to transitions close to the end of the critical sections over the others. This enforces the mutual exclusion. Moreover, priority rules $s_{23} \ll \{b_1, r_1, e_1\}$ and $b_2 \ll b_3$ enforce a particular execution order of critical sections: repeatedly π_1 followed by π_3 and then by π_2.

Using the method of [BBPS09] described in Chapter 8, no controller is found because the support table is incomplete. Indeed, as all states are reachable, no process has enough knowledge to enter or progress in its critical section.

Now, if we calculate the support table on the prioritized executions, we are able to construct a controller for N_1. Indeed, in the prioritized executions, there is always at most one process in its critical section. Thus, process π_1 always supports all its transitions as it can only enter the critical section in global states in which the other processes are blocked in front of a synchronization. Process π_3 supports all its transitions except s_{23}. Process π_2 supports transition s_{23} when π_1 is in p_1, transition b_2 when π_3 is in p_{10}, and transitions r_2 and e_2 in all cases. Thus, considering only states which are reachable in the prioritized system when building the support table yields a solution without additional synchronization.

Note that if we use the synchronization-based approach of Chapter 9, several additional synchronizations are added in order to check maximality of transitions in the critical section. Out of 80 reachable states, 26 are global states in which no process can support an interaction. More precisely, 4 transitions out of 11 always require a synchronization to be fired: these transitions are b_1, r_1, b_3, r_3. As a result, an execution of 10,000 steps contains exactly 3636 ($= 10000 \times 4/11$) synchronizations.

Consider now a simplification of this example with two processes instead of three. In this case, interestingly, the synchronization-based approach does not result in the execution of any additional synchronization. The reason for this is that the states requiring additional synchronizations are exactly the states in which no transition can be supported, meaning that synchronizations are added only when they are necessary. As these states are unreachable in the prioritized executions, no synchronization ever takes place. This emphasizes the fact that both approaches can be combined efficiently.

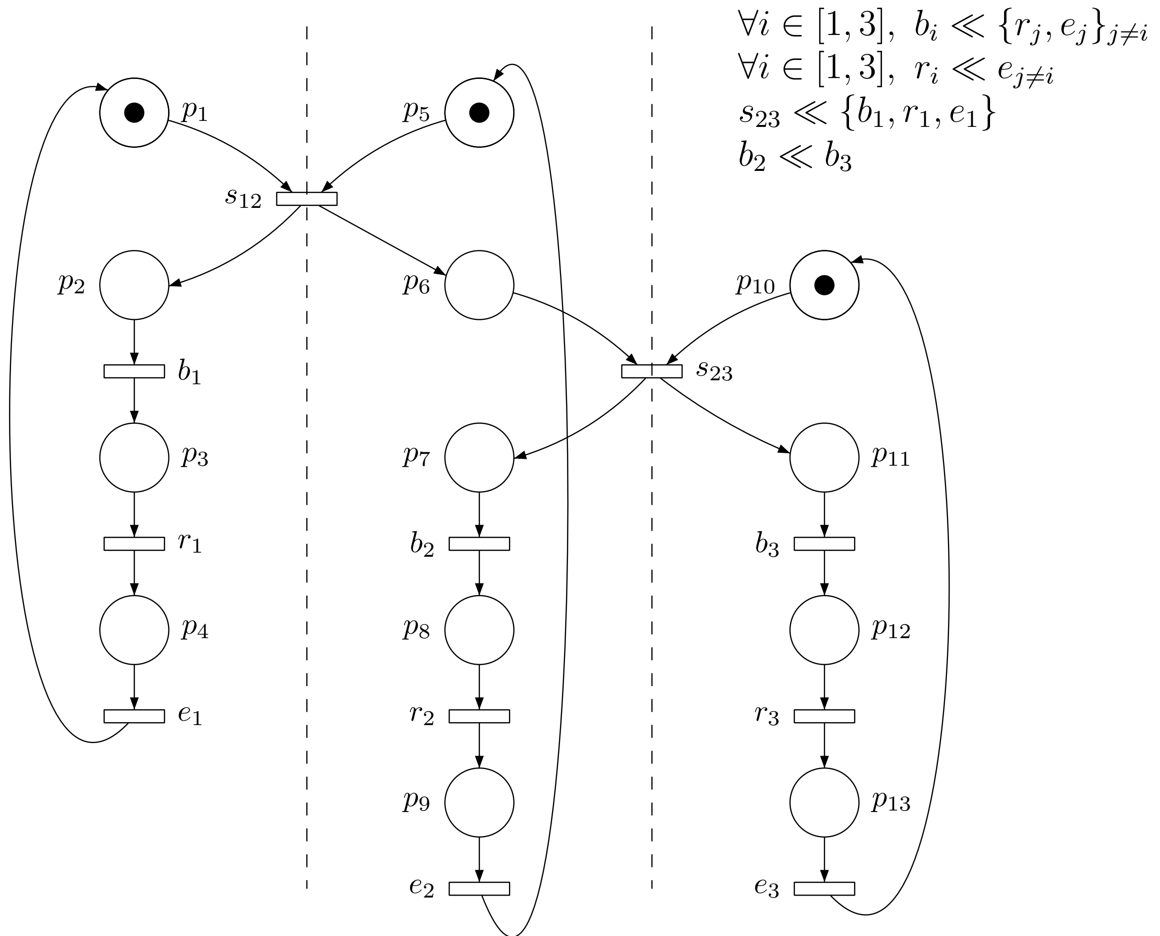


Figure 10.1 – A Petri net N_1 with three processes π_1, π_2 and π_3

10.1.2 Controllers based on an incomplete support table

We now show that even an incomplete support table Δ for (N, Ψ) may still define a controller N^{Δ} implementing (N, Ψ). The reason is that states that are reachable in the executions of (N, Ψ) may be unreachable when applying the controlling the system according to this support table. The executions according to the support table may be a subset of the executions of (N, Ψ), and the synchronization states may not be reachable.

We illustrate this on an example, again using priorities to define the constraint. Consider Petri net N_2 of Figure 10.2. It represents two processes π_l (left) and π_r (right) with a single joint transition, which means that π_l can observe whether π_r is in one of the places p_8 and p_9. Similarly, π_l can observe whether π_r is in p_2 or in p_3.

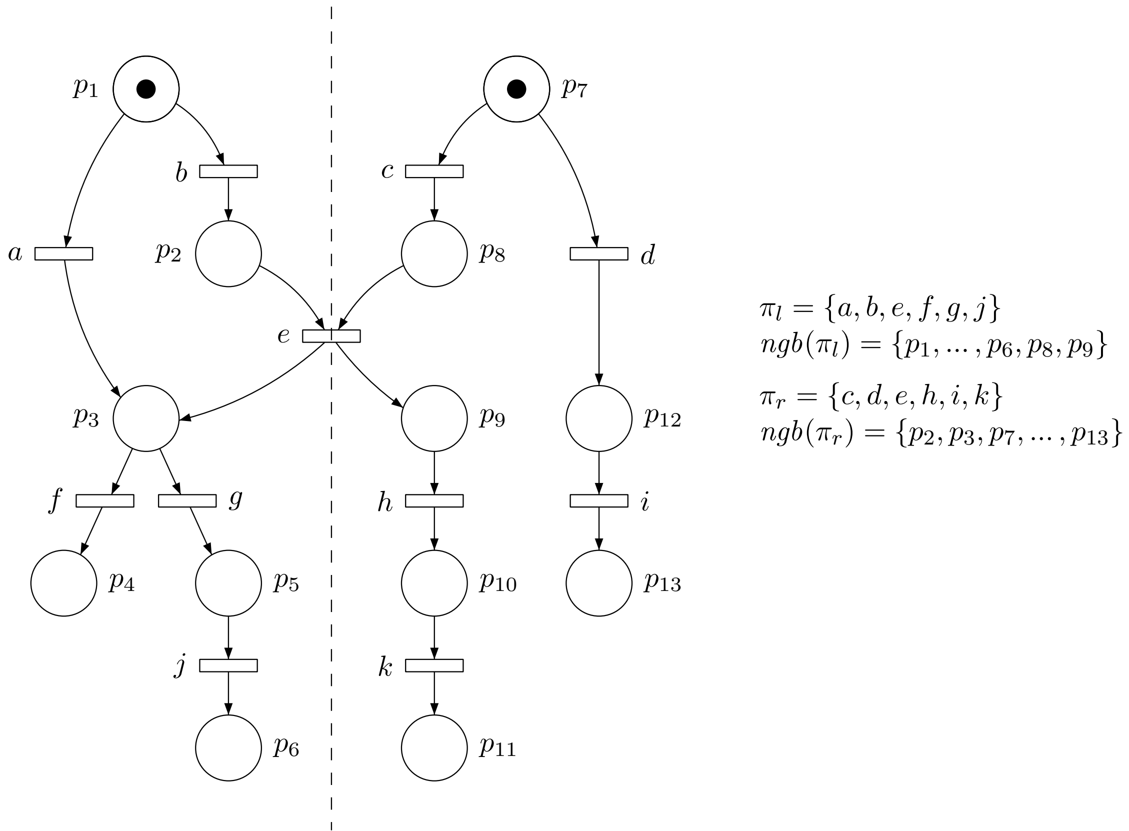


Figure 10.2 – A Petri net N_2 with two processes π_l and π_r

Suppose that Petri net N_2 must be controlled according to the following set of priority rules:

	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}
p_1	a, d					a, i	a
p_2	c, d	e				i	■
p_3	f, g, c, d	f, g	f, g, h	f, g, k	f, g	f, g, i	f, g
p_4	d	■	h	✘	■	i	■
p_5	j, d	j	j, h	j	j	j, i	j
p_6	d	■	h	✘	■	i	■

Table 10.1 – Support policy for N_2 with priorities $k \ll j$ and $c \ll b \ll i$

$k \ll j$ and $c \ll b \ll i$. The support table is calculated based on the reachable states in the prioritized system (priorities make state $\{p_1, p_8\}$ unreachable).

Table 10.1 presents a view of the global states of the Petri net. These states are of one of the following type:

– non-reachable in the prioritized system (they are represented in gray)
– in termination/deadlock in the prioritized system (■)
– deadlock only in the controlled system (✘)
– non-deadlock

In the latter case, the cell contains the transitions which are supported in this state by any of the processes (i.e., we have accumulated all the transitions supported by the local states that constitute together the global state). A red cross in this incomplete table represents a state in which no process supports any transition. There are two such states, namely $\{p_4, p_{10}\}$ and $\{p_6, p_{10}\}$. The situation in both states is the following: π_l has terminated and π_r could take transition k, but without an additional synchronization, there is no way for π_r to know that it may safely execute k.

Note that in state $\{p_1, p_7\}$, process π_l supports a and process π_r supports d; it is impossible for π_l to know whether π_r is in p_{12} or not, and therefore b (which has lower priority than i) is not supported by π_l. Similarly, π_r does not support c (which has lower priority than b). While c is supported, e.g., in $\{p_2, p_7\}$, transition b is never supported, hence never fired in N_2^Δ, although it is allowed according to the priority rules in some states.

As a consequence, the set of states reachable in N_2^Δ may be smaller than $reach(N_2, \ll)$. Indeed, $reach(N_2^\Delta)$ does not contain any state including place p_2 together with p_9, p_{10} or p_{11}. This means in particular that the red crosses in Table 10.1, corresponding to states in which no process supports any transition, are in fact not reachable, and thus N_2^Δ implements $(N_2, \ll)$.

	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}
p_1	a, b, c, d	a, b				a, b, i	a, b
p_2	c, d	e				i	■
p_3	c, d	f, g	f, g, h	k	✘	i	✘
p_4	c, d	■	h	k	■	i	■
p_5	j, c, d	j	j, h	j, k	j	j, i	j
p_6	c, d	■	h	k	■	i	■

Table 10.2 – Support policy for N_2 with priorities $g \ll k$ and $f \ll i$ without history

10.2 Comparison with existing work

10.2.1 History-based controllers

We show now that the use of perfect recall is independent of the methods proposed in this chapter, meaning that in some cases only history is able to provide a controller, while in others it is still relevant to check whether an incomplete table provides a controller.

Consider the Petri net N_2 of Figure 10.2, this time with priorities $g \ll k$ and $f \ll i$. In this case, the set of reachable states is the same, regardless of the use or not of priorities. Consequently, there is no difference between the support policy based on the unrestricted system and the prioritized executions. Moreover, this support policy fails because there are two reachable global states in the support table where no process is supporting a transition, which are marked by red crosses in Table 10.2: $\{p_3, p_{11}\}$ and $\{p_3, p_{13}\}$. Furthermore, these global states are also reachable in the controlled system, meaning that the heuristics applied in the previous example does not help either.

Nevertheless, this example may be controlled if perfect recall is used. If the left process π_l can remember the path it takes to reach p_3, it can distinguish between reaching p_3 directly after p_1 (by firing a) or respectively by passing through place p_2. Now, the set of reachable states contains enough information for the support policy to succeed.

Our last example illustrates the combined use of perfect recall and an incomplete table to build a controller. Consider again the Petri net N_2 of Figure 10.2, now with priorities $g \ll k$, $f \ll \{i, k\}$ and $c \ll b \ll i$. On one hand, building the support table using the prioritized executions does not provide enough knowledge to control the system, and the incomplete support table does not provide a controller. On the other hand, the use of history as shown previously does not help either. Table 10.3 reflects the incomplete support table constructed using jointly the prioritized executions and perfect

	p_7	p_7 after p_3	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}
p_1	a, d						a, i	a
p_2	c, d		e				i	■
p_3 after p_1		c, d, g	f, g				g, i	g
p_3 after p_2				f, g, h	k	✘		
p_4		c, d	■	h	k	■	i	■
p_5		j, c, d	j	j, h	j, k	j	j, i	j
p_6		c, d	■	h	k	■	i	■

Table 10.3 – Support policy for N_2 with $g \ll k$, $f \ll \{i, k\}$, $c \ll b \ll i$ and history recall. Additional information related to perfect recall is presented in the rows and columns of the table only when it is relevant for the support table. We can observe that in $\{p_3$ after $p_2, p_{11}\}$, no transition is supported by any process. However, the system can be controlled according to this table. Indeed, the additional deadlock marked by a red cross in Table 10.3 is actually unreachable within the controlled system — for a reason similar to one presented in the example of Section 10.1.2. This illustrates that sometimes, only the combination of several techniques leads to a controller.

10.2.2 A practical solution to the distributed control problem

We now show some connections between the classical controller synthesis problem (see, e.g., [RW92]) and knowledge-based control. We have provided a solution to the synthesis of distributed controllers, based on adding synchronizations in order to combine the knowledge of individual processes. In this section, we want to put the knowledge-based solution in the context of the distributed control problem when adding synchronizations is *not* allowed.

The knowledge approach to control in [RR00] requires that there is sufficient knowledge to allow *any* transition of the controlled system that does not violate the constraint Ψ. In [BBPS09], which we extend here, this requirement is relaxed; the knowledge must suffice to execute *at least one* enabled transition not violating Ψ when such a transition exists. In the more general case of distributed controller design, one may want to block some enabled transitions even if their execution does not immediately violate the enforced property. This is required to prevent the transformed system from later reaching deadlock states, where the controlled system originally had a way to progress (thus, introducing new deadlocks). When a controller is allowed to block transitions even when their execution does not immediately lead to violation of the property to be preserved, the situation can be recovered.

Notice that the Petri net of Figure 9.1, where local knowledge is not sufficient for controlling the system, can be easily controlled by blocking transitions — even when they are known to be maximal: we may choose either to block α in favor of γ, or to block a in favor of c. Blocking both α and a is not necessary. This illustrates that distributed controllers are more general than knowledge-based controllers. This example also shows that there is no *unique maximal* solution to the control problem that blocks the *smallest* number of transitions. Note that an alternative solution to blocking α or a can be achieved using a temporary synchronization between the processes, as shown earlier.

Actually, even if we redefine our knowledge-based controllers so that they can block transition in order to avoid a deadlock later in the execution, they will still be less powerful than general distributed controllers. The reason for that is that local (knowledge-based) controllers lack the ability to agree *a priori* on transitions which should be taken or not. Consider a variant of the Petri net of Figure 9.1, where a situation similar to that between δ, b and β occurs in the right branch of the processes after γ and c are fired. A distributed controller can still decide that the left process will go left while the right process will go right. This is not feasible using knowledge-based controllers.

On the other hand, there is no algorithm that guarantees constructing distributed controllers. It was shown in [Tri04, Thi05] that the problem of synthesizing a distributed controller is, in general, undecidable. We show here that even when restricting the synthesis problem to priority policies, the problem remains undecidable. The proof for that is given below. Notice that when we have the flexibility of allowing additional synchronizations, as in Chapter 9, the problem, in the limit, becomes a sequential control problem, which is decidable.

Theorem 10.2.1 *Constructing a distributed controller that enforces a priority policy is undecidable.*

Proof. Following [Tri04], the proof is by reduction from the post correspondence problem (PCP). In PCP, there is a finite set of pairs $\{(l_1, r_1), \ldots, (l_n, r_n)\}$, where the components l_i, r_i are words over a common alphabet Σ, and one needs to decide whether one can concatenate separately a *left word* from the left components and a *right word* from the right components according to a mutual nonempty sequence of indexes $i_1 i_2 \ldots i_k$, such that $l_{i_1} l_{i_2} \ldots l_{i_k} = r_{i_1} r_{i_2} \ldots r_{i_k}$.

Let $i \in \{1..n\}$, $\hat{l}_i$ be the word $l_i i$, i.e., the ith left component concatenated with the index i. Similarly, let $\hat{r}_i$ be $r_i i$. We consider two regular languages: $L = (\hat{l}_1 + \hat{l}_2 + \ldots + \hat{l}_k)^+$ and $R = (\hat{r}_1 + \hat{r}_2 + \ldots + \hat{r}_k)^+$. Now suppose a process π_p executes according to the regular expression $l.L.x.a.b + r.R.x.c.d$. The choice of π_p between l and r is uncontrollable. Suppose also that π_p coordinates (through shared transitions) the alphabet letters from Σ with a process π_{q_1}, and the indexes letters from Σ with another process π_{q_2}. After that, π_{q_1} and π_{q_2} are allowed to interact with each

other. Specifically, π_{q_2} sends π_{q_1} the sequences of indexes it has observed. Suppose that now π_{q_1} has a nondeterministic choice between two transitions: α or β. The priorities are set as $b \ll \alpha \ll a$ and $d \ll \beta \ll c$. All other pairs of transitions are unordered according to $\ll$. If π_{q_1} selects α and r was executed, or π_{q_1} selects β and l was executed, then there is no problem, as α is unordered with respect to c and d, and also β is unordered with respect to a and b, respectively. Otherwise, there is no way to control the system so that it executes the sequence $a.\alpha.b$ or $c.\beta.d$ allowed by the priorities.

We show by contrapositive that if there is a controller, then the answer to the PCP problem is negative. Suppose the answer to the PCP problem is positive, i.e., some left and right words are identical and with the same indexes. Then process π_{q_1} cannot make a decision: the information that π_{q_1} observed and later received from π_{q_2} is exactly the same in both cases for the mutual left and right word. Thus, π_{q_1} cannot anticipate whether $c.d$ or $a.b$ will happen and cannot make a safe choice between α and β accordingly.

Conversely, if there is no controller, it means that π_{q_1} cannot make a safe choice between α and β. This can only happen if π_{q_1} and π_{q_2} can observe exactly the same visible information for both an l and an r choice by π_p.

This means that deciding the existence of a controller for this system would solve the corresponding PCP problem. It is thus undecidable. □

Note that in this proof we do not ensure a finite memory controller, even when one exists. Indeed, a finite controller may not exist. To see this, assume a PCP problem with one word $\{(a, aa)\}$. To check whether we have observed a left or a right word, we may just compare the number of a's that p has observed with the number of indexes that q has observed.

We have shown that even our limited problem (and running example) of controlling a system according to priorities is already undecidable. This advocates that the construction of knowledge-based controllers, and furthermore, the use of additional synchronization, is a practical solution to the distributed control problem.

10.3 Conclusion

10.3.1 Summary

Imposing a global constraint upon a distributed system by blocking transitions is, in general, undecidable [Tri04, Thi05]. One practical approach for this problem is to use model checking of knowledge properties [BBPS09], where a precalculation is used to determine when processes can

decide, autonomously, to take or block an action so that the global constraint will not be violated. If we allow additional synchronizations, the problem becomes decidable: at the limit, everything becomes synchronized, although this, of course, is highly undesirable. Since the overhead induced by such synchronizations is important, we strive to minimize their number, again using model checking. This framework applies in particular to the design of controllers that guarantee a priority policy among transitions.

For achieving a distributed implementation, one can use a multi-party synchronization algorithm such as the α-core algorithm [PCT04]. Based on that, we presented an algorithm that uses model checking to calculate when synchronization between local states is needed. The synchronizing processes, successfully coordinating, are then able to use the support table calculated by model checking, which dictates to them which transition can be executed. Some small corrections to the original presentation of the α-core algorithm appear in [KP10].

Furthermore, we have observed that the knowledge used for constructing a distributed controller is computed based on the original (uncontrolled) system. Thus, it may be pessimistic in concluding when transitions can be supported. This has led us to two useful observations that can remove the need for some of the additional synchronizations used to control the system:

1. Although the analysis of the knowledge of the system is done with the original system, it is safe to use only the executions that satisfy the constraint. This results in fewer executions and fewer reachable states, thus enhancing the knowledge.
2. Blocking transitions (not supporting them) because of lack of knowledge has a propagating effect that can prevent reaching other states. Thus, even when the support policy may seem to fail without additional synchronization, this may not be the case. Indeed, analyzing the system when it is restricted according to the support table may be sufficient: the deadlocks corresponding to states where no enabled transition is supported are in fact unreachable.

We have shown that using these two observations is orthogonal to other tools used to force knowledge-based control such as using knowledge of perfect recall and adding temporary or fixed synchronizations between processes.

These two results have been integrated into the prototype presented in Chapter 9. More precisely, the support table is actually built directly from the set of reachable states in the prioritized executions. Then, if the table contains empty entries, we check the reachability of the states in which no transition can be supported before adding synchronization.

Finally, we have proved that the distributed control problem is undecidable even for the special case of enforcing a priority policy, which is the original motivation for this work. This advocates

using knowledge-based controllers enriched with additional synchronizations as a practical solution to the distributed control problem.

10.3.2 Perspectives

There are many interesting ways of refining the approach presented here. A first refinement is to find a compromise between progress and communication overhead. The example with tracks and trains illustrates that deadlock-freedom is not sufficient in many contexts. Allowing for more progress implies adding communication overhead. Thus, we need to define other meaningful criteria to decide when a synchronization should be added. In particular, this requires to have a better understanding of the impact of synchronizations on the number of messages exchanged during the coordinator process. We have already started to study the question of how the synchronization layer and the coordination layer can be profitably merged in [BHGQ10], but this is preliminary work.

Another improvement on our work would be to combine it with abstraction techniques. Indeed, knowledge of a process, defined as the set of global reachable states consistent with its local state, is well-suited for being obtained based on an abstraction of the rest of the system.

Finally, it would be meaningful to integrate this approach into the distributed implementation of BIP [BBS06] which is currently under development. So far, only systems without priorities have been implemented [BBJ$^+$a, BBJ$^+$b]. The question of how to implement BIP systems in a distributed setting remains a challenging task.

Conclusion

We have focused in this thesis on three different aspects of the building of systems: design, verification and implementation. We have advocated that the first two phases should be combined to be more efficient. In particular, designing systems using contracts makes it possible to use these contracts as verification steps. We have presented frameworks in which interaction is complex and can thus simplify both the design and the verification phase, because the structure of the system and its behavior are separated. Part II illustrates the difficulty of implementing such complex interaction models efficiently in a distributed setting. The approach proposed here for distributed control is based on model checking, and thus does not scale to large systems as those targeted by Part I.

Two improvements could help this method scale. The first one would be to find other, maybe more suitable notions of knowledge. So far, we have used a rather naive definition based on the set of reachable global states. Many other possibilities exist and they have to be studied. Also, observability can be modified: neighborhood as it is defined here is not so easy to implement with α-core and other definitions could probably improve the efficiency of the whole approach.

A second possible improvement would to find adequate abstractions that can be combined with the model checking phase of the method. Indeed, a nice development of this thesis would be to use the contracts from the design and verification phase to improve the implementation phase. That is, just like contracts can be used as abstractions provided by the user for verification purposes, they could be used for implementation purposes.

In a different direction, protocols implementing distributed algorithms usually collect and encode some informal "knowledge" into local states by circulating messages. These various "knowledge" properties could be a source of inspiration for new formal definitions of knowledge, and they could lead to formal methods for the verification and distributed implementation of protocols.

Bibliography

[AHKV98] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In *Proceedings of CONCUR'98*, pages 163–178, 1998.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3), 2004.

[Bag89] Rajive Bagrodia. Synchronization of asynchronous processes in CSP. *ACM Trans. Program. Lang. Syst.*, 11(4):585–597, 1989.

[BBBS08] Ananda Basu, Philippe Bidinger, Marius Bozga, and Joseph Sifakis. Distributed semantics and implementation for systems with interaction and priority. In *Proceedings of FORTE'08*, volume 5048 of *LNCS*, pages 116–133. Springer, 2008.

[BBG+10] Saddek Bensalem, Marius Bozga, Susanne Graf, Doron Peled, and Sophie Quinton. Methods for knowledge-based controlling of distributed systems. In *Proceedings of ATVA'10*, volume 6252 of *LNCS*, pages 52–66. Springer, 2010.

[BBJ+a] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. Automated conflict-free distributed implementation of component-based models. To appear in Proceedings of SIES'10.

[BBJ+b] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. From high-level component-based models to distributed implementations. To appear in Proceedings of EMSOFT'10.

[BBNS09] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. D-finder: A tool for compositional deadlock detection and verification. In *Proceedings of CAV'09*, volume 5643 of *LNCS*, pages 614–619. Springer, 2009.

[BBPS09] Ananda Basu, Saddek Bensalem, Doron Peled, and Joseph Sifakis. Priority scheduling of distributed systems based on model checking. In *Proceedings of CAV'09*, volume 5643 of *LNCS*, pages 79–93. Springer, 2009.

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in BIP. In *Proceedings of SEFM'06*, pages 3–12. IEEE Computer Society, 2006.

[BBSN08] Saddek Bensalem, Marius Bozga, Joseph Sifakis, and Thanh-Hung Nguyen. Compositional verification for component-based systems and application. In *Proceedings of ATVA'08*, volume 5311 of *LNCS*, pages 64–79. Springer, 2008.

[BCF+07] Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Multiple viewpoint contract-based specification and design. In *Proceedings of FMCO'07*, volume 5382 of *LNCS*, pages 200–225. Springer, 2007.

[BCH05] Dirk Beyer, Arindam Chakrabarti, and Thomas A. Henzinger. Web service interfaces. In *Proceedings of WWW'05*, pages 148–159, 2005.

[BCP07] Albert Benveniste, Benoît Caillaud, and Roberto Passerone. A generic model of contracts for embedded systems. *CoRR*, abs/0706.1456, 2007.

[BFM+08] Luca Benvenuti, Alberto Ferrari, Leonardo Mangeruca, Emanuele Mazzi, Roberto Passerone, and Christos Sofronis. A contract-based formalism for the specification of heterogeneous systems. In *FDL*, pages 142–147. IEEE Computer Society, 2008.

[BGJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Sci. Comput. Program.*, 16(2):103–149, 1991.

[BGL03] Saddek Bensalem, Susanne Graf, and Yassine Lakhnech. Abstraction as the key for invariant verification. In *Verification: Theory and Practice*, pages 67–99, 2003.

[BGO+04] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF toolset. In *Proceedings of SFM'04*, pages 237–267, 2004.

[BGP03] Howard Barringer, Dimitra Giannakopoulou, and Corina S. Pasareanu. Proof rules for automated compositional verification. In *Proceedings of ESEC/FSE'03*, 2003.

[BHGQ10] Imene Ben-Hafaiedh, Susanne Graf, and Sophie Quinton. Building distributed controllers for systems with priorities. Technical Report TR-2010-15, Verimag, 2010.

[BHQG10a] Imene Ben-Hafaiedh, Sophie Quinton, and Susanne Graf. A contract approach for reasoning about progress: Application to resource-sharing in a network, 2010. To appear in Proceedings of FLACOS'10.

[BHQG10b] Imene Ben-Hafaiedh, Sophie Quinton, and Susanne Graf. A contract framework for reasoning about safety and progress. Technical Report TR-2010-11, Verimag, 2010.

[BHQG10c] Imene Ben-Hafaiedh, Sophie Quinton, and Susanne Graf. Reasoning about safety and progress using contracts, 2010. To appear in Proceedings of ICFEM'10.

[BHS07] Dirk Beyer, Thomas A. Henzinger, and Vasu Singh. Algorithms for interface synthesis. In *Proceedings of CAV'07*, pages 4–19, 2007.

[BJS09] Marius Bozga, Mohamad Jaber, and Joseph Sifakis. Source-to-source architecture transformation for performance optimization in BIP. In *SIES*, pages 152–160. IEEE Computer Society, 2009.

[Blo93] Bard Bloom. *Ready Simulation, Bisimulation, and the Semantics of CCS-Like Languages*. PhD thesis, MIT, 1993.

[BLO98] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of CAV'98*, pages 319–331, 1998.

[BM79] Robert S. Boyer and J. Strother Moore. *A computational logic*. Academic Press, 1979.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of PLDI'01*, pages 203–213, 2001.

[BS83] Gael N. Buckley and Abraham Silberschatz. An effective implementation for the generalized input-output construct of CSP. *ACM Trans. Program. Lang. Syst.*, 5(2):223–235, 1983.

[BS03] Philippe Bidinger and Jean-Bernard Stefani. The Kell calculus: operational semantics and type systems. In *Proc. of FMOODS'03*, volume 2884 of *LNCS*, 2003.

[BS08a] Simon Bliudze and Joseph Sifakis. The algebra of connectors - structuring interaction in BIP. *IEEE Trans. Computers*, 57(10):1315–1330, 2008.

[BS08b] Simon Bliudze and Joseph Sifakis. A notion of glue expressiveness for component-based systems. In *Proceedings of CONCUR'08*, volume 5201 of *LNCS*, pages 508–522. Springer, 2008.

[BZ07] Mario Bravetti and Gianluigi Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In *Software Composition*, pages 34–50, 2007.

[CAC08] Jamieson M. Cobleigh, George S. Avrunin, and Lori A. Clarke. Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. *ACM Trans. Softw. Eng. Methodol.*, 17(2):1–52, 2008.

[CCF+10] Yu-Fang Chen, Edmund M. Clarke, Azadeh Farzan, Ming-Hsien Tsai, Yih-Kuen Tsay, and Bow-Yaw Wang. Automated assume-guarantee reasoning through implicit learning. In *Proceedings of CAV'10*, volume 6174 of *LNCS*, pages 511–526. Springer, 2010.

[CCST05] Sagar Chaki, Edmund M. Clarke, Nishant Sinha, and Prasanna Thati. Automated assume-guarantee reasoning for simulation conformance. In *Proceedings of CAV'05*, pages 534–547, 2005.

[CdAHS03] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Resource interfaces. In *Proceedings of EMSOFT'03*, pages 117–133. ACM, 2003.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proceedings of Logic of Programs'82*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of CAV'00*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.

[CGP03] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *Proceedings of TACAS'03*, volume 2619 of *LNCS*, pages 331–346. Springer, 2003.

[CGP08] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. In *Proceedings of POPL'08*, pages 261–272, 2008.

[CH07] Krishnendu Chatterjee and Thomas A. Henzinger. Assume-guarantee synthesis. In *Proceedings of TACAS'07*, pages 261–275, 2007.

[CLM89] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional model checking. In *Proceedings of LICS'89*, pages 353–362. IEEE Computer Society, 1989.

[CM07] Pablo F. Castro and T. S. E. Maibaum. A complete and compact propositional deontic logic. In *Proceedings of ICTAC'07*, pages 109–123, 2007.

[COM] Combest project. `http://www.combest.eu`.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *Proceedings of POPL'87*, pages 178–188, 1987.

[dAH01a] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120, 2001.

[dAH01b] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In *Proceedings of EMSOFT'01*, pages 148–165. ACM, 2001.

[dAH05] Luca de Alfaro and Thomas Henzinger. Interface-based design. In D. Harel M. Broy, J. Gruenbauer and C.A.R. Hoare, editors, *Engineering Theories of Software-intensive Systems*, volume 195 of *NATO Science Series: Mathematics, Physics, and Chemistry*, pages 83–104. Springer, 2005.

[dAHM00] Luca de Alfaro, Thomas A. Henzinger, and Freddy Y. C. Mang. Detecting errors before reaching them. In *Proceedings of CAV'00*, pages 186–201, 2000.

[dAHS02] Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Timed interfaces. In *Proceedings of EMSOFT'02*, pages 108–122. ACM, 2002.

[Dam05] Werner Damm. Controlling speculative design processes using rich component models. In *Proceedings of ACSD'05*, pages 118–119. IEEE Computer Society, 2005.

[DDHL09] Ajoy Kumar Datta, Stéphane Devismes, Florian Horn, and Lawrence L. Larmore. Self-stabilizing k-out-of- exclusion on tree networks. In *Proceedings of IPDPS'09*, pages 1–8. IEEE Computer Society, 2009.

[DHJP08] Laurent Doyen, Thomas A. Henzinger, Barbara Jobstmann, and Tatjana Petrov. Interface theories with component reuse. In *Proceedings of EMSOFT'08*, pages 79–88. ACM, 2008.

[dRdBH$^+$01] Willem P. de Roever, Frank S. de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.

[EC80] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of ICALP'80*, volume 85 of *LNCS*, pages 169–181. Springer, 1980.

[EJL+03] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, Stephen Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[FHVM95] Ronald Fagin, Joseph Y. Halpern, Moshe Y. Vardi, and Yoram Moses. *Reasoning about knowledge*. MIT Press, Cambridge, MA, USA, 1995.

[GGMC+06] Gregor Gößler, Susanne Graf, Mila E. Majster-Cederbaum, Moritz Martens, and Joseph Sifakis. Ensuring properties of interaction systems. In *Program Analysis and Compilation*, pages 201–224, 2006.

[GL81] Hartmann J. Genrich and Kurt Lautenbach. System modelling with high-level petri nets. *Theor. Comput. Sci.*, 13:109–136, 1981.

[GL91] Orna Grumberg and David E. Long. Model checking and modular verification. In *Proceedings of CONCUR'95*, volume 527 of *LNCS*, pages 250–265. Springer, 1991.

[GLL99] Alain Girault, Bilung Lee, and Edward A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 18(6):742–760, 1999.

[GMF07] Anubhav Gupta, Kenneth L. McMillan, and Zhaohui Fu. Automated assumption generation for compositional verification. In *Proceedings of CAV'07*, volume 4590 of *LNCS*, pages 420–432. Springer, 2007.

[GPB05] Dimitra Giannakopoulou, Corina S. Pasareanu, and Howard Barringer. Component verification with automatically generated assumptions. *Autom. Softw. Eng.*, 12(3):297–320, 2005.

[GPC04] Dimitra Giannakopoulou, Corina S. Pasareanu, and Jamieson M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *Proceedings of ICSE'04*, pages 211–220, 2004.

[GPQ10] Susanne Graf, Doron Peled, and Sophie Quinton. Achieving distributed control through model checking. In *Proceedings of CAV'10*, volume 6174 of *LNCS*, pages 396–409. Springer, 2010.

[GQ07] Susanne Graf and Sophie Quinton. Contracts for BIP: Hierarchical interaction models for compositional verification. In *Proceedings of FORTE'07*, volume 4574 of *LNCS*, pages 1–18. Springer, 2007.

[Gru05] Orna Grumberg. Abstraction and refinement in model checking. In *Proceedings of FMCO'05*, pages 219–242, 2005.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *Proceedings of CAV'97*, pages 72–83, 1997.

[GS03a] Gregor Gößler and Joseph Sifakis. Component-based construction of deadlock-free systems: Extended abstract. In *Proceedings of FSTTCS'03*, pages 420–433, 2003.

[GS03b] Gregor Gößler and Joseph Sifakis. Priority systems. In *Proceedings of FMCO'03*, volume 3188 of *LNCS*, pages 314–329. Springer, 2003.

[GS05] Gregor Gößler and Joseph Sifakis. Composition for component-based modeling. *Sci. Comput. Program.*, 55(1-3):161–183, 2005.

[GSL96] Susanne Graf, Bernhard Steffen, and Gerald Lüttgen. Compositional minimisation of finite state systems using interface specifications. *Formal Asp. Comput.*, 8(5):607–616, 1996.

[HJS01] Michael Huth, Radha Jagadeesan, and David A. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *Proceedings of ESOP'01*, volume 2028 of *LNCS*, pages 155–169. Springer, 2001.

[HM92] Joseph Y. Halpern and Yoram Moses. A guide to completeness and complexity for modal logics of knowledge and belief. *Artificial Intelligence*, 54(2):319–379, 1992.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HQR00] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *Proceedings of ICCAD'00*, pages 245–252, 2000.

[HQRT98] Thomas A. Henzinger, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. An assume-guarantee rule for checking simulation. In *Proceedings of FMCAD'98*, pages 421–432, 1998.

[HS07] Thomas A. Henzinger and Joseph Sifakis. The discipline of embedded systems design. *IEEE Computer*, 40(10):32–40, 2007.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[HZ92] Joseph Y. Halpern and Lenore D. Zuck. A little knowledge goes a long way: Knowledge-based derivations and correctness proofs for a family of protocols. *J. ACM*, 39(3):449–478, 1992.

[Jon83] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[KP10] Gal Katz and Doron Peled. Code mutation in verification and automatic code correction. In *Proceedings of TACAS'10*, volume 6015 of *LNCS*, pages 435–450. Springer, 2010.

[Lar89] Kim Guldstrand Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 232–246. Springer, 1989.

[LGS+95] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

[LL98] Bilung Lee and Edward A. Lee. Hierarchical concurrent finite state machines in ptolemy. In *Proceedings of ACSD'98*, pages 34–40. IEEE Computer Society, 1998.

[LN05] Edward A. Lee and Stephen Neuendorffer. Concurrent models of computation for embedded software. In *Computers and Digital Techniques*, pages 239–250, 2005.

[LNW06] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Interface input/output automata. In *Proceedings of FM'06*, volume 4085 of *LNCS*, pages 82–97. Springer, 2006.

[LNW07] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal I/O automata for interface and product line theories. In *Proceedings of ESOP'07*, volume 4421 of *LNCS*, pages 64–79. Springer, 2007.

[Lon93] David E. Long. *Model checking, abstraction and compositional reasoning*. PhD thesis, CMU, 1993.

[LP07] Cosimo Laneve and Luca Padovani. The *must* preorder revisited. In *Proceedings of CONCUR'07*, volume 4703 of *LNCS*, pages 212–225. Springer, 2007.

[LSW95] Kim Guldstrand Larsen, Bernhard Steffen, and Carsten Weise. A constraint oriented proof methodology based on modal transition systems. In *Proceedings of TACAS'95*, volume 1019 of *LNCS*, pages 17–40. Springer, 1995.

[LW94] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[LX90] Kim Guldstrand Larsen and Liu Xinxin. Equation solving using modal transition systems. In *Proceedings of LICS'90*, pages 108–117. IEEE Computer Society, 1990.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[LZZ05] Edward A. Lee, Haiyang Zheng, and Ye Zhou. Causality interfaces and compositional causality analysis. In *Proceedings of FIT'05*, 2005.

[Mai01] Patrick Maier. A set-theoretic framework for assume-guarantee reasoning. In *Proceedings of ICALP'01*, volume 2076 of *LNCS*, pages 821–834. Springer, 2001.

[Mai03a] Patrick Maier. Compositional circular assume-guarantee rules cannot be sound and complete. In *Proceedings of FoSSaCS'03*, volume 2620 of *LNCS*, pages 343–357. Springer, 2003.

[Mai03b] Patrick Maier. *A Lattice-Theoretic Framework for Circular Assume-Guarantee Reasoning*. PhD thesis, Universität des Saarlandes, 2003.

[Mau] Maude system. `http://maude.cs.uiuc.edu`.

[MB07] Florence Maraninchi and Tayeb Bouhadiba. 42: programmable models of computation for a component-based approach to heterogeneous embedded systems. In *Proceedings of GPCE'07*, pages 53–62. ACM, 2007.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[McM99] Kenneth L. McMillan. Circular compositional reasoning about liveness. In *Proceedings of CHARME'99*, volume 1703 of *LNCS*, pages 342–345. Springer, 1999.

[MCM08] Mila E. Majster-Cederbaum and Moritz Martens. Compositional analysis of deadlock-freedom for tree-like component architectures. In *Proceedings of EMSOFT'08*, pages 199–206. ACM, 2008.

[Mey92] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Theor. Comput. Sci.*, 25:267–310, 1983.

[Mil85] George J. Milne. Circal and the representation of communication, concurrency, and time. *ACM Trans. Program. Lang. Syst.*, 7(2):270–298, 1985.

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

[MP83] Zohar Manna and Amir Pnueli. How to cook a temporal proof system for your pet language. In *Proceedings of POPL'83*, pages 141–154, 1983.

[MP91] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer, 1991.

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4), 1989.

[Orl77] James B. Orlin. Contentment in graph theory: covering graphs with cliques. *Indagationes Mathematicae*, 80(5):406–424, 1977.

[Pad08] Luca Padovani. Contract-directed synthesis of simple orchestrators. In *Proceedings of CONCUR'08*, volume 5201 of *LNCS*, pages 131–146. Springer, 2008.

[PCT04] José Antonio Pérez, Rafael Corchuelo, and Miguel Toro. An order-based algorithm for implementing multiparty synchronization. *Concurrency - Practice and Experience*, 16(12):1173–1206, 2004.

[PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proceedings of POPL'89*, pages 179–190, 1989.

[PR90] Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of FOCS'90*, volume II, pages 746–757. IEEE Computer Society, 1990.

[PS07] Cristian Prisacariu and Gerardo Schneider. A formal language for electronic contracts. In *Proceedings of FMOODS'07*, volume 4468 of *LNCS*, pages 174–189. Springer, 2007.

[QBHG09] Sophie Quinton, Imene Ben-Hafaiedh, and Susanne Graf. From orchestration to choreography: Memoryless and distributed orchestrators. In *Proceedings of FLACOS'09*, 2009.

[QG08a] Sophie Quinton and Susanne Graf. Contract-based verification of hierarchical systems of components. In *Proceedings of SEFM'08*, pages 377–381. IEEE Computer Society, 2008.

[QG08b] Sophie Quinton and Susanne Graf. A framework for contract-based reasoning: Motivation and application. In *Proceedings of FLACOS'08*, pages 77–84, 2008.

[QGP10] Sophie Quinton, Susanne Graf, and Roberto Passerone. Contract-based reasoning for component systems with complex interactions. Technical Report TR-2010-12, Verimag, 2010.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of Symposium on Programming'82*, volume 137 of *LNCS*, pages 337–351. Springer, 1982.

[Rac08] Jean-Baptiste Raclet. Residual for component specifications. *Electr. Notes Theor. Comput. Sci.*, 215:93–110, 2008.

[RBB+09a] Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, Axel Legay, and Roberto Passerone. Modal interfaces: unifying interface automata and modal specifications. In *Proceedings of EMSOFT'09*, pages 87–96. ACM, 2009.

[RBB+09b] Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, and Roberto Passerone. Why are modalities good for interface theories? In *Proceedings of ACSD'09*, pages 119–127. IEEE Computer Society, 2009.

[Rei84] John H. Reif. The complexity of two-player games of incomplete information. *J. Comput. Syst. Sci.*, 29(2):274–301, 1984.

[RR00] Karen Rudie and S. Laurie Ricker. Know means no: Incorporating knowledge into discrete-event control systems. *IEEE Transactions on Automatic Control*, 45(9):1656–1668, 2000.

[RSW04] Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. Static program analysis via 3-valued logic. In *Proceedings of CAV'04*, volume 3114 of *LNCS*, pages 15–30. Springer, 2004.

[RW87] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, 1987.

[RW92] Karen Rudie and W. Murray Wonham. Think globally, act locally: decentralized supervisory control. *IEEE Transactions on Automatic Control*, 37(11):1692–1708, 1992.

[Sif05] Joseph Sifakis. A framework for component-based construction extended abstract. In *Proceedings of SEFM'05*, pages 293–300, 2005.

[Sim] Simulink. `http://www.mathworks.com/products:simulink/`.

[Sin07] Nishant Sinha. *Automated Compositional Analysis for Checking Component Substitutability*. PhD thesis, CMU, 2007.

[SPE] SPEEDS project. `http://www.speeds.eu.com`.

[Sys] Open systemC initiative. `http://www.systemc.org/`.

[Thi05] John G. Thistle. Undecidability in decentralized supervision. *System and Control Letters*, 54:503–509, 2005.

[Tho95] Wolfgang Thomas. On the synthesis of strategies in infinite games. In *Proceedings of STACS'95*, volume 900 of *LNCS*, pages 1–13. Springer, 1995.

[TLHL09] Stavros Tripakis, Ben Lickly, Thomas A. Henzinger, and Edward A. Lee. On relational interfaces. In *Proceedings of EMSOFT'09*, pages 67–76. ACM, 2009.

[Tri04] Stavros Tripakis. Undecidable problems of decentralized observation and control on regular languages. *Inf. Process. Lett.*, 90(1):21–28, 2004.

[vdM98] Ron van der Meyden. Common knowledge and update in finite environment. *Information and Computation*, 140(2):115–157, 1998.

[YL02] Tae-Sic Yoo and Stéphane Lafortune. A general architecture for decentralized supervisory control of discrete-event systems. *Discrete Event Dynamic Systems*, 12(3):335–377, 2002.