Implantations distribuées de modèles à base de composants communicants par interactions multiparties avec priorités : application au langage BIP

Jean Quilbeuf

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Jean Quilbeuf

Thèse dirigée par **Marius Bozga**
et codirigée par **Joseph Sifakis**

préparée au sein **VERIMAG, UMR5104**
et de **MSTII**

Distributed Implementations of Component-based Systems with Prioritized Multiparty Interactions.

Application to the BIP Framework.

Thèse soutenue publiquement le **16 septembre 2013**,
devant le jury composé de :

Yassine Lakhnech
Professeur, Université Joseph Fourier, Président
Michel Raynal
Professeur, Université de Rennes, Rapporteur
Gul Agha
Professeur, University of Illinois, Rapporteur
Ugo Montanari
Professeur, Università di Pisa, Examinateur
Marius Bozga
Docteur, Université Joseph Fourier, Directeur de thèse
Joseph Sifakis
Professeur, Université Joseph Fourier, Co-Directeur de thèse

Remerciements

Je voudrais tout d'abord remercier mon directeur de thèse, Marius Bozga, qui a toujours été disponible, patient et plein de ressources pour résoudre les différents problèmes rencontrés. Je suis très honoré d'être son premier thésard "officiel". Je voudrais également remercier mon codirecteur de thèse, Joseph Sifakis, pour sa vision des choses et son exigence de précision qui m'ont permis d'avancer réellement.

I would like to thank Gul Agha and Michel Raynal, for reading this thesis. Thanks also to Ugo Montanari for taking part in my committee and to Yassine Lakhnech for leading it. Once again, thanks to all of them for the meaningful and interesting questions they asked.

Merci à Léon, Anakreontas, Paris, Petro pour avoir lu et commenté des chapitres de cette thèse.

Merci à Laurent, pour m'avoir proposé un stage à Verimag. Merci à Borzoo pour ses conseils de rédaction de papier et sa collaboration. Merci à Doron Peled, pour ses idées et son efficacité. Merci à Saddek pour ses conseils et ses contacts. Merci à Mohamad, Jacques, Ahlem pour leur collaboration et la bonne ambiance régnant pendant les réunions. Merci à tous les membres de Verimag pour l'ambiance qui y règne. C'est très agréable de travailler dans ce lieu, et les pauses midi de la cafeteria du CTL resteront, pas forcément dans l'histoire, mais quelque part.

Merci à Nathalie et à Dominique pour leurs conseils concernant mes premières passes devant des étudiants. Merci également aux collègues de C2i pour la bonne humeur pendant les réunions de préparations et corrections de rattrapage.

Bashar, Simon, Thomas, Gérolin, Mélaine et Camille, je vous remercie pour cet atelier-projet CIES en Palestine. Ce fut une agréable pause, bien qu'un peu caniculaire, dans cette thèse.

Merci enfin à ceux qui ont été présents dans ces quatre (dernières) années grenobloises. Qu'ils soient collègues, colocataires, mathématiciens, grimpeurs, anciens mathématiciens, voisins de table de bar ou musiciens de tous niveaux, merci à eux pour avoir rendu agréable cette période. Mention spéciale à Maximiliano, pour son art de l'artisanat, ses cours de mécanique, ses petits tours et sa bonne humeur. Simon, sans tes questions sur l'informatique en général, de la définition du mot algorithme à l'installation de Debian, ma vie n'aurait pas été la même. Sans ton ukulele non plus. Bashar, merci pour ces

discussions du petit matin ou du grand soir, à quand la prochaine ? Thomas, merci pour la musique (Maxime aussi) et les cours de vulgarisation sur le TR-909. Je me lance, mais je vais en oublier, merci à Tof, Guilux, Aline, Gunnar, Abby, Fred, Ariadna, Abby, Josue, Adrien, Lemon, Robin, Mathieu, Florence, Yvan, Florian, Valls, Alban, Manon, Dorothée, Gentiane, Paris, Balaji, Emmanuel, Tesnim, Vasso, Jérôme, Steffen, Eduardo, Julia, Artur, Julien, Mathilde, Fanny, Yan et Koju. Merci aussi à ceux que j'ai oubliés...

Merci enfin à mon frère et mes parents qui m'ont permis d'arriver jusque là sans avoir à m'occuper d'autre chose que de mes études. Merci aussi à eux pour la relecture du résumé en français, et le pot !

Contents

Notations

ϵ The empty string. 72

γ A set of BIP interactions. 77

κ Condition for a BIC model. 78

π A BIP priority rule. 78

σ The prefix of an execution trace. 72

τ A transition in a Petri net or in a concrete BIP atomic component. 44

Γ A BIP connector. 83

Δ Amount of time to wait in Joung's randomized algorithm. 57

$\mathcal{B}$ The Boolean domain, i.e. $\{True, False\}$. 68

$\mathcal{D}$ Domain of definition of the data variables. 82

$\mathcal{I}$ Set of multiparty interactions. 42, 59

$\mathcal{I}_k$ The set of interactions managed by M_k. 52

$\mathcal{K}_i$ Support automaton for the process $\mathcal{P}_i$ or component B_i. 73

$\mathcal{L}$ Set of observed (Local) processes. 66

$\mathcal{P}$ A process involved in multiparty interactions. 40, 41, 52

$\mathcal{R}$ Set of reachable states. 66

$\tilde{\mathcal{R}}$ Over approximation of the reachable states. 68

a An action or interaction. 41, 49, 77, 83

at_ℓ Predicate that is true whenever the local state ℓ is active. 67

$invl_\kappa$ The set $invl_\kappa(a)$ contains the components that either participants in a or observed by a. 88

$observed_\kappa$ The set $observed(a)$ contains the components whose state is needed to evaluate the predicate associated to a by the Condition layer κ, but who are not participants in a. 88

participants The set $participants(a)$ contains the components that are participants in the interaction a. 83

usedin The set $usedin(\phi)$ contains the variables that appear in the predicate ϕ. 86

A The set of actions of a process. 40, 41

B A component (or behavior), can be atomic or composite. Often denoted B_i when atomic. 76

B^{SR} A distributed component, can be atomic or composite. Often denoted B_i^{SR} when atomic. 94, 95, 119

C State predicate used to define a concrete priority rule. 86

CP Centralized version of the conflict resolution protocol. 123

DP Dining philosophers version of the conflict resolution protocol. 127

E The centralized engine. 98, 121

EN_a State predicate that is true when the interaction a is enabled. 80, 89

EN_{p_i} State predicate that is true when the port p_i is enabled. 80, 89

$Exec$ The set of all valid execution prefixes. 49

I A subset of $\{1, \ldots, n\}$ that denotes indices of components/processes. It can be the set of participants in an interaction. 42

$K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\phi$ Knowledge predicate that is true whenever the predicate ϕ holds in all global states of $\tilde{\mathcal{R}}$ that are indistinguishable from the local state of $\mathcal{L}$. 68

$K_{\mathcal{P}_i}^{PR}\phi$ Knowledge predicate that is true whenever the predicate ϕ holds in all global states reached after executing any global execution prefix that is consistent with the local history of $\mathcal{P}_i$. 72

L Set of places for a Petri net or control locations for a concrete BIP atomic component. 44, 81

M_k A manager from Bagrodia's solutions. 9, 52

P A set of communication ports. 77

Q Set of states. 40–42, 76

R Candidate observational equivalence in the proofs. 103, 133

TR Token ring version of the conflict resolution protocol. 125

X A set of data variables. 81

Y^X The set of all applications from the set X to the set Y. 44

busy state State of a distributed atomic component/process corresponding to a computation and that will terminate by the emission of a message. 47, 96

conflicting Two interactions are conflicting if they need a common resource. 48

enabled An interaction is enabled if all its participants are ready to perform it. 42

externally conflicting An interaction is externally conflicting within an engine if it conflicts with at least one interaction handled by another engine. 120

internally conflicting An interaction is internally conflicting within an engine if it conflicts only with interaction handled by the same engine. 120

stable state State of a distributed atomic component/process where no computation can occur until reception of a message. 47, 96

synchron Type of port depicted ● that needs to be activated to interact. 84

trigger Type of port depicted ▲ that can initiate an interaction. 84

◄ Condition conflict relation between interactions. 113

\# Interaction conflict relation between interactions. 113

$\rightarrow$ A transition relation. 40–42

Acronyms

AADL Architecture Analysis and Design Language. 149

BBC Boolean Behavioral Constraints. 70

BDD Binary Decision Diagram. 152

BIC Behavior Interaction Condition. 76

BIP Behavior Interaction Priority. 33, 76

CA Condition-aware. 173

CCS Calculus of Communicating Systems. 60, 88

CSP Communicating sequential processes. 88

DOL Distributed Operation Layer. 149

GALS Globally Asynchronous Locally Synchronous. 61

LOTOS Language Of Temporal Ordering Specification. 57

LTS Labeled Transition System. 40, 42, 59

MB Multiparty-based. 173

MPI Message-Passing Interface. 33

PBLTL Probabilistic Bounded Linear Temporal Logic. 149

SCCS Synchronous Calculus of Communicating Systems. 88

Résumé en français

1 Introduction

Les systèmes conçus aujourd'hui ont de plus en plus recours à du logiciel distribué. La principale raison est probablement l'efficacité, mais cela peut aussi être du à l'implantation physique des capteurs et actuateurs qui impose d'avoir plusieurs unités de calculs physiquement séparées. Un système distribué est constitué d'un ensemble de processus communicant par envoi de messages. Raisonner directement à ce niveau est très difficile et la vérification formelle devient vite impossible en raison de l'imposant espace d'états généré par tous les ordonnancements possibles des messages.

Afin d'éviter ces écueils, notre approche se base sur un flot de conception permettant de garantir la correction et l'efficacité de l'implémentation, ainsi que de limiter l'intervention humaine autant que possible. La correction du flot est garantie par le fait que chaque transformation le constituant est assez simple pour être prouvée correcte. L'efficacité peut être optimisée en réglant les différents paramètres des transformations afin d'adapter l'implémentation à la plateforme visée. Enfin, l'intervention humaine est limitée au choix des paramètres, les tâches répétitives sont automatisées.

Dans [47], l'auteur affirme que l'envoi de message est trop bas niveau pour programmer et propose d'utiliser les opérations collectives de MPI [41]. Ces opérations collectives sont des synchronisations fortes entre plusieurs processus, mais requièrent que chacun des processus ne propose d'effectuer qu'une seule opération de ce type à chaque étape. Nous proposons d'utiliser des interactions multiparties où chacun des processus peut proposer plusieurs interactions et le choix de l'interaction effectivement exécutée dépend de celles qui sont possibles.

Les modèles que nous considérons sont décrits en BIP [10]. Un flot de conception rigoureux [80] est utilisé pour construire une implémentation distribuée d'un tel modèle. Un tel flot est constitué d'une série de raffinements menant du modèle original à un modèle du logiciel tel qu'il sera implémenté. Le même cadre sémantique est utilisé pour décrire tous les modèles intermédiaires. Ces modèles sont des assemblages de composants, permettant de réutiliser du code de source hétérogène. Chacune des transformations est suffisamment simple pour être prouvée, permettant d'assurer la correction de l'implémentation générée par construction. Enfin, le flot de conception est automatisé autant que possible, par des outils effectuant les différentes transformations, seul le choix, difficile, des paramètres est laissé au concepteur.

Notre flot de conception part d'un modèle contenant des composants, dont le comportement est décrit par un automate ou un réseau de Petri. Trois opérateur de composition permettent de coordonner les composants. Le premier est spécifié par un ensemble d'interactions, chacune consistant en une synchronisation de transitions appartenant à

différents composants. Un interaction est active si tous les composants sont prêts à exécuter la transition correspondante. L'exécution d'une interaction est atomique, tous les composants changent d'états simultanément. Le deuxième opérateur, appelé Priorité, définit un ordre partiel sur les interactions. Si deux interactions comparables sont active à partir d'un état global, seulement celle de plus haute priorité peut être exécutée. Le troisième opérateur, appelé Condition, qui est utilisé au dessus des interactions, comme les priorités, assigne un prédicat à chaque interaction. L'interaction ne peut s'effectuer que si le prédicat, qui dépend de l'état de certains composants, est vrai. L'opérateur Priorité est facilement exprimé au moyen de l'opérateur Condition.

La première étape de décentralisation consiste à casser l'atomicité des interactions, en séparant la participation des composants en deux temps. Dans un premier temps, le composant envoie une offre, indiquant les actions localement possibles. Dans un deuxième temps, il attend une notification lui indiquant laquelle des actions a été choisie. Un engin centralisé implémente la sémantique de manière distribuée: il reçoit les offres et renvoie les notifications conformément au modèle de départ.

L'engin centralisé prend la décision d'exécuter une interaction à partir d'un état global. En particulier, pour implémenter l'opérateur Condition, il faut attendre de connaître l'état de tous les composants nécessaires à l'évaluation du prédicat. Dans certain cas, l'état d'une partie de ces composants est suffisant pour savoir que le prédicat est vrai. Nous proposons de détecter ces cas en utilisant la théorie de la connaissance.

La deuxième étape de décentralisation sépare l'engin centralisé en un ensemble d'engins décentralisés. Cela peut induire des conflits entre engins et nécessite le cas échéant l'introduction d'un protocole de résolution de conflit.

Le protocole de résolution de conflit peut être optimisé en augmentant la connaissance des composants. Dans ce cas, un composant n'envoie pas d'offre pour une interaction localement active mais qu'il sait globalement non active.

Finalement, on obtient un modèle BIP dont les interactions sont limitées à l'envoi de messages asynchrones. L'implémentation est obtenue en générant du code utilisant les primitives disponibles sur la plateforme pour implémenter ces interactions.

2 Interactions Multiparties

Ce chapitre présente le concept d'interaction multipartie dont les différentes versions sont présentées dans [56]. Dans cette thèse, une interaction multipartie est la synchronisation d'un ensemble fixé de processus pour effectuer une action commune. Cette action est atomique dans le sens où aucune autre action n'a lieu pendant l'exécution d'une interaction.

2.1 Système de processus

On représente un processus par un automate, dont les transitions sont étiquetées par des noms d'interactions. La composition de ces processus se fait en synchronisant les transitions étiquetées par le même nom. L'ensemble des participants dans une interaction est l'ensemble des processus dont au moins l'une des transitions est étiquetée par le nom

de l'interaction. La sémantique d'un tel système de processus est définie sur les états globaux du système qui sont définis comme le produit cartésien des état locaux de chacun des processus. Étant donné un état global, une interaction est active si tous ses participants sont dans un état à partir duquel une transition étiquetée par le nom de l'interaction est possible.

La sémantique d'un tel système est représentée par un automate dont les états sont les états globaux du système et les transitions sont les interactions du système. On peut obtenir un réseau de Petri ayant la même sémantique. On crée une place pour chaque état de chaque processus. Pour chaque ensemble de places activant une interaction, on crée une transition étiquetée par cette interaction qui déplace les jetons correspondants vers les places correspondant à l'état atteint après l'interaction.

2.2 Exécution distribuée

La définition précédente de la sémantique des interactions multiparties se base sur l'état global du système. L'une des principales difficultés est d'exécuter ces interactions dans un cadre distribué. Pour cela, chacun des processus indique à un protocole la liste des interactions qu'il peut effectuer, ce qui constitue son offre. Le processus attend alors une notification du protocole indiquant quelle interaction exécuter, l'exécute puis renvoie une nouvelle offre correspondant à l'état atteint. Le protocole doit récupérer les offres et envoyer les notifications, de façon à respecter la sémantique du modèle de départ. Le protocole peut prendre la forme d'un processus particulier, de plusieurs processus ou être directement réparti dans chacun des processus originaux.

Un tel protocole doit résoudre les conflits entre interactions. Un conflit apparaît entre deux interactions lorsqu'elles ont au moins un participant en commun. Si deux interactions en conflit sont exécutées simultanément par des processus distincts, la sémantique du modèle de départ n'est pas respectée, car chaque processus ne peut participer qu'à une seule interaction.

Dans la littérature, la sémantique des interactions multiparties est rarement définie à partir des états globaux du système. La correction d'un protocole est dans ce cas énoncée en termes de propriétés sur l'exécution distribuée. Ces propriétés comportent l'exclusion mutuelle des interactions en conflit, la synchronisation des participants dans une même interaction et le progrès si une interaction est active. Dans cette thèse, la correction d'un protocole, ou plus généralement, d'un modèle distribué, est énoncée comme une équivalence entre le modèle de départ et le modèle distribué.

2.3 Protocoles étudiés

Plusieurs auteurs ont proposé des protocoles pour exécuter des interactions multiparties dans un contexte distribué. Dans [5, 6] Bagrodia propose plusieurs protocoles utilisant des processus dédiés implémentant le protocole, appelés managers. Le plus simple utilise un seul manager centralisé. Ce manager est ensuite décentralisé en plusieurs managers, chacun responsable de l'exécution d'un ensemble d'interactions. L'exclusion mutuelle des interactions en conflit est assurée soit par un jeton circulant entre les différents managers,

soit par un algorithme issu d'une solution au problème du dîner des philosophes [32].

Un autre protocole, proposé par Kumar dans [61], n'utilise pas de processus externe mais embarque le protocole dans les processus du système. Ce protocole associe un jeton à chaque interaction, le jeton doit parcourir un chemin passant par tous les processus de l'interaction. Lorsqu'un jeton traverse un processus, ce dernier est bloqué par l'interaction correspondant au jeton, jusqu'à succès ou échec de l'interaction. Afin d'éviter les interblocages, chaque jeton parcourt les processus selon un ordre global. Si un jeton parvient au dernier processus du chemin, alors tous les processus sont bloqués et l'interaction peut avoir lieu. Sinon, un message d'annulation est envoyé aux processus déjà traversés afin qu'ils se débloquent et laissent passer un autre jeton.

Dans [55], Joung propose un algorithme randomisé, qui, comme celui de Kumar, ne nécessite pas de processus externe. Chaque processus tire au hasard une interaction de son offre et envoie, à chaque participant de l'interaction, un jeton étiqueté par cette interaction. Ensuite le processus attend pendant un intervalle de temps déterminé. Enfin, il redemande tous les jetons envoyés plus tôt. Durant son temps d'attente, chaque processus reçoit des jetons de la part des autres processus. Si un processus détient, pour une interaction donnée, un jeton de chacun de participants étiqueté par l'interaction, alors cette dernière doit être exécutée, ce qui est signalé en rajoutant une marque sur les jetons. Quand un processus récupère les jetons envoyés, si l'un d'entre eux est marqué, il exécute l'interaction, sinon il tire au hasard une autre interaction et recommence.

Le dernier protocole considéré est α-core qui construit un participant pour chaque processus et un coordinateur pour chaque interaction. Chaque coordinateur reçoit les offres des participants et tente de les verrouiller un par un, chaque participant n'acceptant d'être verrouillé que par un seul coordinateur à la fois. Tous les coordinateurs verrouillent les participants selon un ordre global commun, afin d'éviter les interblocages. Si un coordinateur parvient à verrouiller tous les participants de son interaction, il exécute cette dernière. Sinon, une autre interaction en conflit a été exécutée et le coordinateur relâche les participants déjà verrouillés.

2.4 Priorités

Pour diminuer le non-déterminisme du choix de la prochaine interaction à effectuer, on peut équiper le système de priorités. Ces priorités sont décrites par un ordre partiel sur les interactions, seule une interaction maximale pour cet ordre parmi les interactions actives peut être exécutée. Les règles de priorité peuvent être utilisées pour éviter d'atteindre certains états. Par exemple, l'interblocage atteint dans le modèle classique du dîner des philosophes où chacun prend la fourchette de gauche puis la fourchette de droite peut être évité en donnant plus de priorité à l'interaction qui permet de prendre la fourchette de droite. Une autre utilisation des priorités favorise l'exécution d'interactions "utiles". Par exemple, un système insérant un disque dans un lecteur puis l'éjectant, sans jamais lire le disque, n'exécute pas d'interaction "utile". Donner plus de priorité à l'interaction permettant de lire qu'à l'interaction permettant d'éjecter le disque évite de telles exécutions.

En général, les interactions multiparties sont présentées dans un cadre distribué où

la correction est exprimée par des propriétés sur les envois de messages. Dans un tel cadre, la notion d'état global du modèle centralisé sous-jacent n'est pas définie, ce qui empêche de définir des priorités. C'est pourquoi la littérature aborde peu le problème des interactions multiparties régies par des priorités.

2.5 Extensions et autres modèles distribués

La notion d'interaction multipartie présentée ci-dessus peut être étendue, en définissant une interaction comme un ensemble de rôles. L'interaction est alors active s'il existe pour chacun des rôles un processus prêt à prendre ce rôle.

D'autres modèles existent et proposent de générer du code distribué à partir d'un modèle haut niveau qui est généralement moins expressif que les interactions multiparties avec priorités.

3 Connaissance

Le terme connaissance, dans cette thèse, est la somme d'informations détenue par un agent ou une partie d'un système à propos de l'état global du système. Le concept de connaissance est utilisé afin de formaliser ce que chaque agent sait ou peut déduire à propos du système, en partant de ses observations.

Cette notion peut servir à formaliser le problème des "enfants boueux" [7] : Après avoir joué ensemble, certains enfants d'un groupe ont de la boue sur leur front et risquent d'être punis. Chaque enfant sait si les autres sont sales, mais pas s'il l'est lui-même, et aucun autre enfant ne lui communique cette information. Quand les enfants rentrent, le père leur dit qu'au moins l'un d'entre eux est sale, ce qui devient un fait connu de tous les enfants. Le père demande ensuite qui peut affirmer avec certitude qu'il est sale, les enfants répondent simultanément. Le père continue à poser la question et les enfants à y répondre. La question est de savoir ce qui va se passer, en supposant que les enfants ne mentent pas et qu'ils raisonnent parfaitement.

La question posée par le père est typiquement une question de connaissance où la réponse est soit positive, auquel cas un fait est connu (l'enfant est sale et le sait), soit négative auquel cas le fait peut être vrai ou faux (l'enfant ne sait pas s'il est sale). À noter que chaque enfant, voyant k enfants sales, sait qu'il y en a au total soit k, soit $k+1$, suivant que lui-même est sale ou non.

La réponse au problème est que si k enfants sont sales, après la $k$ième itération de la question, les enfants sales répondent oui. Cela se prouve par récurrence sur k. S'il y a un seul enfant sale, il ne verra pas d'autre enfant sale, déduira que c'est lui qui est sale et répondra "oui" à la première question du père. S'il y en a deux, chacun d'entre eux voit un enfant sale au début, et ne peut pas faire la différence avec le cas $k = 1$. Après les réponses (toutes négatives) à la première question du père, le cas $k = 1$ est éliminé par chacun des enfants sales, qui en déduit qu'il est lui-même sale et répond "non" à la deuxième question du père. De même, s'il y a $k+1$ enfant sales, chacun d'entre eux sait que le nombre d'enfants sales n'est pas k après la réponse à la $k$ième question du père, et, en déduisent qu'ils sont sales.

Dans ce problème, on raisonne sur la connaissance de chacun des enfants qui ont une observation limitée du système mais sont capables d'inférer ce qu'ils ne voient pas. Ce schéma s'applique aux systèmes distribués où, par définition, chacun des processus n'observe que son propre état qui dépend des messages reçus. Différents niveaux de connaissance sont définis. La plus faible est la connaissance distribuée, obtenue en combinant la connaissance de plusieurs processus. Ce genre de connaissance requiert une synchronisation pour être construite. La plus forte est la connaissance commune, connue par tous les processus et dont les processus savent qu'elle est connue par tous les processus.

La connaissance se base sur un ensemble d'univers, l'un d'entre eux étant celui dans lequel le système évolue. Chacun des agents observe une partie des faits de cet univers et peut en déduire que le système se trouve dans un univers cohérent avec les faits observés. Toutefois, il ne peut faire la différence entre deux univers différant uniquement sur des faits non observables.

3.1 Connaissance distribuée obtenue à partir de l'état local

Cette connaissance considère un sous-ensemble $\mathcal{L}$ des processus. On observe l'état local de ce sous-ensemble, c'est-à-dire les états des processus qui le composent. Deux états globaux dont la projection sur $\mathcal{L}$ donne le même état local sont dits indistinguables par $\mathcal{L}$.

Un invariant du système fournit une sur-approximation des états atteignables. Un invariant est une formule qui est toujours vraie durant l'exécution du système. En particulier, tout état ne satisfaisant pas l'invariant n'est pas atteignable. Ces invariants sont obtenus en utilisant la théorie des réseaux de Petri. Les trappes fournissent ce que l'on appelle des invariants booléens dans cette thèse. Les invariants linéaires sont obtenus en considérant une base du noyau de la matrice place-transition du réseau de Petri correspondant au système.

L'observation de l'état local de $\mathcal{L}$ indique que le système se trouve dans un état à la fois cohérent avec cette observation et satisfaisant l'invariant. Si une formule Φ est satisfaite par chacun de ces états globaux, alors l'état de $\mathcal{L}$, combiné avec l'invariant, permet d'assurer que Φ est vrai. On définit le prédicat de connaissance $K_{\mathcal{L}}\Phi$ qui est vrai quand l'état local de $\mathcal{L}$ permet d'assurer que Φ est vrai. Si ce prédicat est faux, on ne sait pas si Φ est vrai.

Par définition, on a $K_{\mathcal{L}}\Phi \implies \Phi$, c'est-à-dire que $K_{\mathcal{L}}\Phi$ est une sous-approximation de Φ. On obtient une sur-approximation en considérant la contraposée de $K_{\mathcal{L}}\neg\Phi \implies \neg\Phi$. On sait que Φ contient les états pour lesquels on le sait vrai et ne contient pas les états pour lesquels on le sait faux. Cet encadrement s'affine d'autant plus que le nombre de processus observés est grand et devient une égalité si tout les processus sont observés.

3.2 Connaissance obtenue à partir de l'historique

Ce type de connaissance observe les interactions visibles par un processus, c'est-à-dire les interactions dans lesquelles il est impliqué. Deux séquences d'exécution sont indis-

cernables par un processus si leurs restrictions aux interactions visibles sont les mêmes.

Dans le cas de processus ayant un ensemble d'états finis, cette connaissance se représente par un automate dont les états sont des ensembles d'états globaux et les transitions sont étiquetées par des interactions visibles par le processus. Chaque état de l'automate correspond à un ensemble d'états globaux. L'état initial de l'automate contient tous les états globaux accessibles depuis l'état initial du système en exécutant uniquement des interactions non visibles. L'état de l'automate atteint après une séquence d'interactions visibles contient tous les états accessibles par n'importe quelle séquence valide d'interactions indiscernable de cette séquence.

Après avoir vu une séquence σ d'interactions visibles, le processus sait que l'état actuel du système est contenu dans l'état de l'automate atteint après avoir joué σ. Si tous les états globaux potentiellement atteints satisfont un prédicat Φ, le processus sait que Φ est vrai. Cette construction donne plus d'informations que l'observation de l'état du processus.

3.3 Travaux liés à la connaissance

Différentes notions de connaissance sont formalisées et étudiées dans [40, 50, 51]. Les ensembles approximatifs (rough sets) [71] fournissent des approximations d'ensembles d'objets dont seulement certaines caractéristiques sont observables.

Une application de la connaissance au contrôle décentralisé d'un système est proposée dans [77]. D'autres travaux [8, 12, 19] proposent des contrôleurs distribués permettant de forcer une propriété sur l'exécution d'un réseau de Petri. Ces travaux supposent l'existence d'un mécanisme pour exécuter correctement les transitions du réseau de Petri.

4 Modèles BIP et BIC

Ce chapitre présente les modèles BIP (Behavior Interaction Priority) et BIC (Behavior Interaction Condition). Ces modèles sont décrits par des composants atomiques composés par des opérateurs.

4.1 Modèles abstraits

Les modèles abstraits ne contiennent pas de données, seulement des états de contrôle. Dans ce cadre un composant atomique est défini par un comportement et une interface. L'interface est un ensemble de ports utilisé pour la communication. Le comportement est un automate ou un réseau de Petri dont les transitions sont étiquetées par des ports. Un composant peut exécuter une transition étiquetée par un port si son état de contrôle active cette transition. Dans un tel état, le port est actif.

Le premier opérateur de composition est spécifié par des interactions. Une interaction est un ensemble de ports. La sémantique d'un ensemble de composants composé par des interactions est un système de transitions étiqueté par les interactions. Depuis un état global du système, formé par le produit des états de chacun des composants, une

interaction est possible si chacun des ports qui la composent est actif pour l'état courant. L'exécution d'une interaction ne change que l'état de ses composants.

On définit un opérateur de priorité en spécifiant, pour chaque état global, un ordre partiel entre les interactions. Une interaction plus faible pour cet ordre a moins de priorité que l'interaction qui la domine. Un tel opérateur inhibe l'exécution d'une interaction de faible priorité uniquement lorsqu'une interaction de plus haute priorité est active. En particulier, cet opérateur n'introduit pas d'interblocage car il bloque une interaction seulement si une autre peut être exécutée. Une priorité est dite statique si elle ne dépend pas de l'état global.

L'opérateur de condition associe un prédicat, dépendant de l'état global du système, à chaque interaction. Cet opérateur inhibe l'exécution d'une interaction si le prédicat associé n'est pas satisfait par l'état courant.

On peut construire un opérateur de condition dont le comportement est équivalent à l'opérateur de priorité. On associe à chaque interaction de faible priorité le prédicat qui est vrai uniquement quand toutes les interactions de plus haute priorité ne sont pas actives.

4.1 Modèles concrets

Le modèle concret étend le modèle abstrait en rajoutant des variables dans les composants. Chaque port est associé à un ensemble de variables. Ces variables peuvent être lues et écrites durant une interaction contenant ce port. Chaque transition comporte une garde, prédicat sur les variables qui doit être vrai pour autoriser la transition. Un état du composant comporte l'état de contrôle et une valuation des variables. Le fait qu'un port soit actif dépend de l'existence d'une transition depuis l'état de contrôle actuel et des valeurs des variables à travers la garde associée à la transition. Enfin, quand la transition est exécutée, les variables associées au port sont éventuellement modifiées par l'interaction, puis une fonction, associée à la transition, est exécutée. Cette fonction modifie localement les variables.

De même, les interactions sont étendues pour prendre en charge les données. Comme pour les transitions dans les composants, l'exécution d'une interaction requiert que sa garde soit vraie et entraîne l'exécution d'une fonction. La garde et la fonction sont définies sur l'ensemble des variables associées aux ports constituant l'interaction.

Les interactions peuvent être spécifiées en utilisant des connecteurs. Chaque connecteur est défini sur un ensemble de ports, son support, et autorise certaine interactions utilisant ces ports. Les interactions autorisées dérivent d'un typage de chacun des ports comme synchron ou trigger. Si le connecteur ne comporte que des synchrons, alors seule l'interaction contenant tous les ports est autorisée. Sinon toute combinaison contenant au moins un trigger est autorisée. Ces connecteurs peuvent être composés hiérarchiquement, dans ce cas un connecteur exporte un port qui est utilisé dans le support d'un autre connecteur. Les variables associées au port exporté sont calculées par une fonction de propagation dépendant des variables associées aux ports du support. Ces variables sont utilisées par le connecteur dominant pour décider si la garde est vraie. Si l'interaction s'exécute, une fonction de propagation vers le bas calcule les valeurs des

variables associées aux ports du support.

Les priorités sont définies par un ensemble de règles, comportant un prédicat et un couple d'interactions. Le prédicat dépend des variables visibles par les interactions et indique quand la règle doit s'appliquer. La première interaction du couple est celle de basse priorité et l'autre est celle de haute priorité. Pour chaque état, l'ensemble des règles s'appliquant doit donner un ordre partiel sur les interactions. Une priorité statique est obtenue en prenant des tautologies comme prédicats.

Les conditions sont exprimées en associant à chaque interaction un prédicat qui dépend de l'état de contrôle et des variables de certains composants. L'introduction des conditions permet un intermédiaire entre participation et non-participation à une interaction. Ce rôle particulier est tenu par les composants dont l'observation est nécessaire pour déterminer si le prédicat associé à une interaction est vrai, mais qui ne sont pas participants dans cette interaction. De tels composants sont observés par l'interaction.

Comme précédemment, les priorités peuvent se réécrire en utilisant des conditions. De plus, on peut transformer un modèle avec condition en un modèle utilisant uniquement des interactions. Cela nécessite d'ajouter un port à chaque composant observé afin d'y étendre les interactions qui l'observent. Le prédicat associé à l'interaction devient alors une garde. Toutefois, les interactions restent une colle plus faible que les interactions avec priorités ou conditions car il est nécessaire de modifier les composants pour avoir un comportement équivalent.

5 Rupture de l'atomicité des interactions: parallélisme entre les composants

Une interaction se décompose en une partie commune, matérialisée en BIP par la fonction associée à l'interaction et une partie locale à chaque composant, consistant principalement en l'exécution d'une fonction locale de mise à jour des variables. En exécutant les composants dans des processus séparés, le parallélisme entre ces dernières apparaît naturellement.

Dans un cadre distribué, il n'est pas possible d'observer de façon simultanée l'état de tous les composants, ni de déclencher simultanément des transitions dans des composants séparés. Chaque composant envoie la liste des ses ports actifs vers un engin qui est un composant particulier chargé de planifier l'exécution des interactions. L'engin calcule les interactions possibles d'après les offres qu'il a reçues, choisit l'une d'elle, exécute la partie commune et envoie une notification à chacun des composants participants pour qu'ils exécutent leur partie locale de l'interaction.

5.1 Restriction de modèles

Les modèles considérés en entrée doivent avoir des composants dont le comportement est décrit par un automate (et non par un réseau de Pétri). Les priorités doivent être préalablement transformées en conditions.

Les modèles obtenus après transformation sont dits Send/Receive au sens où ils ne

comportent que des interactions modélisant l'envoi de message. Chaque composant représente un processus autonome. Chaque port est soit un port d'envoi soit un port de réception. Chacune des interactions contient un port d'envoi et un port de réception. Un port d'envoi participe à exactement une interaction, c'est-à-dire est associé à exactement un port de réception (l'envoi est déterministe). Une interaction copie les variables associées au port d'émission vers les variables associées au port de réception. Enfin, si à un état donné un port d'envoi est actif, le port de réception correspondant doit impérativement devenir actif. Cette dernière restriction assure que chaque message envoyé est reçu.

Un modèle Send/Receive est utilisé pour générer du code distribué. Chaque composant devient un processus. Les transitions étiquetées par un port d'envoi déclenchent l'envoi d'un message vers le port de réception associé. Les transitions étiquetées par un port de réception sont exécutées à la réception du message correspondant.

5.2 Transformation vers le modèle distribué

La transformation est effectuée en transformant chaque composant atomique du modèle de départ en une version distribuée, puis en rajoutant un engin.

Étant donné un composant atomique centralisé, on obtient sa version distribuée en séparant une transition en deux parties. La première partie, exécutée avant d'atteindre un état "stable" consiste en l'envoi d'une offre, cette transition est donc étiquetée par un port d'envoi. Cette offre contient la liste des ports actifs dans l'état stable, ainsi que les variables associées. De cet état stable sont possibles des interactions étiquetées par des ports de réception, un pour chaque port possible depuis l'état correspondant dans le composant atomique original. La réception d'une notification est alors possible et entraîne l'exécution de la transition associée, ce qui amène le composant dans un état occupé, rajouté dans la construction. Depuis cet état, seul l'envoi d'une nouvelle offre est possible, permettant de rejoindre le prochain état stable.

Le modèle distribué est complété par un engin chargé de recevoir les offres et d'envoyer les notifications. Le comportement de cet engin est décrit par un réseau de Pétri. À chaque composant du système original est associé un jeton dans ce réseau. Ce jeton est soit dans une place d'attente, lorsqu'aucune offre n'a été reçue, soit dans une place indiquant qu'une offre a été reçue, soit dans une place indiquant qu'il faut envoyer une notification au composant. Le passage de la place d'attente à la place indiquant la présence d'une offre se fait lors de la réception de cette dernière. Le passage de la place indiquant la présence d'une offre à la place indiquant la nécessité d'envoyer une notification se fait par une transition interne, correspondant à une interaction. Pour chaque interaction du modèle, une telle transition déplace les jetons depuis les places indiquant la présence d'une offre vers les places indiquant la nécessité d'un envoi pour les ports constituant l'interaction. La garde associée à cette transition vérifie que l'interaction est bien active, c'est-à-dire que les offres contiennent bien tous les ports de l'interaction, et que la garde associée à l'interaction est vraie. Si l'interaction requiert l'observation de composants, la transition nécessite la présence des jetons correspondants dans les places indiquant la présence d'une offre et la garde vérifie également le prédicat associé

à l'interaction. L'exécution de cette transition déclenche le calcul de la fonction associée, c'est-à-dire de la partie commune de l'interaction. Le passage d'une place indiquant la nécessité d'envoyer une notification à la place d'attente du composant se fait par une transition étiquetée par un port d'envoi. Chaque port a une place dédiée, permettant de séparer les différentes notifications possibles pour chaque composant. Chaque port d'envoi correspondant à une notification transmet également les valeurs des variables associées au port, après leur mise à jour lors de l'interaction.

Les versions distribuées des composants atomiques et l'engin sont reliés par des interactions de type Send/Receive. Chaque port d'envoi d'offre d'un composant atomique est relié au port de réception d'offre correspondant de l'engin. Chaque port d'envoi de notification de l'engin (il y en a un pour chaque port) est relié au port de réception de notification correspondant dans le composant atomique distribué.

5.3 Correction du modèle distribué

Le modèle distribué est conforme aux restrictions imposées dans la première section de ce chapitre. Par construction, les interactions se conforment au modèle Send/Receive. De plus, l'envoi d'une offre active les ports de réception associés aux notifications correspondantes et, symétriquement, l'envoi d'une notification entraîne l'activation du port de réception pour l'offre suivante. La propriété concernant l'activation des ports de réception est donc vérifiée.

Le modèle distribué est observationnellement équivalent au modèle de départ. Pour chaque état du système distribué, on considère l'état stable obtenu en effectuant tous les envois de message possibles qui sont considéré comme invisibles. Dans un état stable, chaque composant atomique distribué est dans un état qui est également un état du composant atomique centralisé correspondant. Les états des composants atomiques distribués donnent alors un état global du système de départ considéré comme équivalent. On peut alors montrer que depuis un état distribué et son état équivalent, les mêmes transitions visibles, c'est-à-dire interactions, sont possibles et permettent d'atteindre des états équivalents.

5.4 Utilisation de la connaissance pour prendre des décisions plus tôt

Pour évaluer un prédicat associé à une interaction, l'engin doit attendre une offre de chacun des composants observés. Dans certains cas, en particulier lorsque la condition dérive d'une priorité, cette décision peut être prise plus tôt, en considérant uniquement un sous-ensemble de ces offres.

On spécifie arbitrairement, pour chaque interaction, un ensemble de composants à observer. En pratique, on cherche à minimiser ce nombre de composants, afin de prendre la décision le plus tôt possible. On considère ensuite la connaissance distribuée, au sens du chapitre 3, du prédicat de condition obtenue à travers l'observation des participants et des composants à observer. Si pour une observation donnée cette connaissance permet d'assurer que le prédicat est vrai, alors l'interaction peut être exécutée sans compromettre la correction de l'exécution globale du système. Dans le cas contraire, l'interaction

ne peut être exécutée. Ce comportement est obtenu en remplaçant le prédicat associé à une interaction par le prédicat de connaissance permettant de l'approcher, c'est-à-dire en construisant une nouvelle couche de condition.

En restreignant trop l'ensemble des composants à observer, on prend le risque de n'avoir pas assez d'information pour exécuter les interactions et d'introduire des interblocages dans le système. Afin d'éviter cela, on définit des niveaux de détection caractérisant le comportement du modèle utilisant la connaissance par rapport au modèle original. Le niveau de détection basique garantit que l'on introduit pas d'interblocage dans le système. Le niveau de détection complet garantit l'équivalence entre les deux modèles.

L'approche présentée ici est paramétrée par les composants à observer pour chaque interaction. En pratique, on cherche à minimiser ces ensembles tout en maintenant un certain niveau de détection. À cette fin, on propose un algorithme de recuit simulé permettant de minimiser les composants observés tout en assurant un niveau de détection basique ou complet.

6 Décentraliser l'engin : parallélisme entre les interactions

La solution utilisant un engin centralisé exécute toutes les interactions dans le même processus, ce qui interdit le parallélisme entre elles. Obtenir ce parallélisme requiert de décentraliser l'engin en plusieurs engins. On paramètre cette décentralisation par une partition des interactions, chaque classe de la partition étant gérée par un engin différent.

6.1 Conflits

De manière générale, un conflit se crée lorsque deux parties ont besoin de la même ressource. Un composant participant dans deux interactions différentes est une source potentielle de conflit. En particulier, si ces deux interactions sont exécutées dans des engins distincts et sans coordination, elles peuvent être exécutées simultanément. Cela contredit la sémantique des interactions multiparties où chaque composant ne participe qu'à une seule interaction à la fois.

Nous définissons deux types de conflits entre les interactions, suivant le rôle (participant ou composant observé) de chaque composant en lien avec l'interaction. Deux interactions sont en conflit d'interaction lorsque elles ont au moins un participant en commun. C'est le conflit classique entre des interactions multiparties. Considérons deux interactions a et b, telles que a observe le composant B pour évaluer le prédicat de condition et B participe à l'interaction b. Dans ce cas, l'interaction a est en conflit de condition avec l'interaction b. Ce conflit est asymétrique car l'exécution de a ne compromet pas l'exécution de b, alors que l'exécution de b modifie l'état de B et peut donc désactiver a.

Dans les deux cas, nous dirons que a et b sont en conflit. Notons que si a et b observent un composant commun, il ne s'agit pas d'un conflit. On retrouve le même comportement avec la mémoire transactionnelle : deux transactions accédant à la même

variable en lecture ne sont pas en conflit, mais dès que l'une des deux transactions accède à la variable en écriture, on a un conflit.

6.2 Partitions sans conflits

Dans la version avec un seul engin, présentée au chapitre précédent, les conflits sont résolus à l'intérieur de l'engin, par la sémantique même du réseau de Petri. En groupant toujours deux interactions en conflit dans le même engin, on obtient une partition sans conflits. Dans ce cas tous les conflits sont entre des interactions gérées par le même engins et peuvent être résolus localement.

Chaque offre envoyée par un composant ne peut concerner qu'un seul engin à la fois. En effet, une offre envoyée à deux engins différents serait une source de conflit potentiel. De ce fait, on peut modifier les composants atomiques en ajoutant un port d'envoi différent pour chaque engin. Chaque offre est alors envoyée uniquement à l'engin concerné. La construction d'un engin à partir d'un ensemble d'interactions est la même que précédemment.

Il est toujours possible de construire une partition sans conflit en groupant toutes les interactions ensemble, ce qui revient au cas précédent. Dans certains cas, cette partition ne peut être raffinée en restant sans conflit. Afin d'autoriser un partitionnement arbitraire des interactions dans les engins, il faut ajouter un protocole de résolution de conflits.

6.3 Modèle Send/Receive à 3 niveaux

On considère dans cette section un partitionnement arbitraire des interactions. Chaque ensemble d'interactions du partitionnement est géré par un engin dédié. Le modèle distribué obtenu contient alors 3 niveaux. Le premier est constitué par les composants atomiques modifiés. Le deuxième contient les engins. Enfin, le troisième est un protocole de résolution de conflits.

Le principe du protocole de résolution de conflits est basé sur les solutions de Bagrodia [6] utilisant des compteurs. Chaque composant compte les offres qu'il envoie et leur associe la valeur de ce compteur. Un conflit correspond à la situation où deux interactions peuvent utiliser la même offre, c'est-à-dire le même numéro. Résoudre les conflits revient à s'assurer que chaque numéro d'offre n'est utilisé qu'une seule fois.

À partir des offres reçues, chaque engin détecte les cas où l'une des interactions dont il est responsable peut s'exécuter. Une interaction dont tous les conflits sont internes à l'engin peut être exécutée directement par ce dernier. Dans le cas contraire, une requête contenant le numéro d'offre de chacun des participants est envoyée au protocole de résolution de conflit. Si ce dernier répond positivement, l'interaction est exécutée. En cas de réponse négative, l'engin essaye une autre interaction ou attend une nouvelle offre.

Pour chaque composant, le protocole de résolution de conflit conserve le numéro d'offre utilisé lors de la dernière participation du composant à une interaction. Cette information permet de rejeter toute offre d'un composant dont le numéro d'offre est plus petit

ou égal au numéro de dernière participation, puisque l'offre correspondante a déjà été utilisée. Le protocole de résolution de conflit répond positivement seulement si tous les numéros d'offre de la requête sont encore valides, sinon il répond négativement.

On propose trois versions du protocole d'interaction. Le point crucial pour la correction du protocole est d'assurer l'atomicité de l'opération qui lit les numéros de dernière participation pour vérifier la validité d'une requête et les modifie en cas de succès. La première version consiste en un unique composant qui est le seul à accéder à ces numéros, ce qui garantit l'atomicité. La seconde version utilise un jeton contenant les numéros. Ce jeton circule entre les composants du protocole de résolution de conflits. Seul le composant détenant le jeton peut modifier les numéros situés dessus, ce qui garantit l'atomicité. La dernière version, plus décentralisée, est basée sur une solution du problème du diner des philosophes. Chaque interaction correspond à un philosophe, implémenté par un composant. Deux composants correspondant à deux interactions en conflit s'échangent une fourchette qui contient les numéros de dernière participation des composants causant le conflit. Un composant doit récupérer toutes les fourchettes partagées avec d'autres composants pour modifier les numéros, ce qui assure l'atomicité.

6.4 α-core

Le protocole α-core est un autre protocole de résolution de conflit qui a été utilisé pour générer un modèle distribué à partir d'un modèle centralisé. Ce protocole se compose d'un processus pour chaque composant et pour chaque interaction. Nous n'avons pas étendu ce protocole pour supporter les modèles avec condition.

Le modèle BIP distribué embarquant ce protocole est construit de manière hiérarchique. Chaque composant du modèle original est obtenu comme composition de la version distribuée de ce protocole, d'un composant générique implémentant la partie composant d'α-core, et de quelques composants nécessaires à la communication. De façon similaire, chaque interaction est implémentée par la composition d'un composant exécutant la fonction de l'interaction, d'un composant générique implémentant la partie interaction d'α-core et de composants additionnels pour la communication.

6.5 Optimisation utilisant la connaissance avec mémoire parfaite

Le protocole α-core implémente l'exclusion mutuelle des interactions en conflit sans avoir recours à des compteurs. Lors de l'exécution d'une notification, chaque composant doit explicitement annuler toutes les offres envoyées aux autres interactions et attendre une confirmation pour chaque annulation. Dans ce cadre, il est intéressant de n'envoyer des offres qu'aux interactions qui sont réellement possibles.

En utilisant la connaissance avec mémoire parfaite, chaque composant peut éventuellement restreindre les offres qu'il envoie en se basant sur son état local, mais aussi sur l'historique des interactions. Ce genre d'optimisation est utile lorsqu'un contexte particulier induit un comportement précis à un composant générique. L'historique des interactions permet dans ce cas de limiter les prochaines interactions possibles, alors que l'état du composant ne le permet pas.

Le même principe peut être appliqué aux coordinateurs. Dans ce cas, l'historique utilisé est celui des offres reçues et des notifications envoyées. La connaissance obtenue permet de ne pas tenir compte de certaines offres, quand elles ne peuvent être utilisées dans l'interaction. De plus, on peut considérer certaines offres comme acquises, quand on sait qu'il n'y a pas d'autre interaction dans laquelle le composant correspondant peut participer.

6.6 Discussion

Ce chapitre présente plusieurs méthodes pour exécuter des interactions en parallèle, ce qui suppose de résoudre les conflits apparaissant lors de la décentralisation. La première méthode utilise des compteurs et peut être plus ou moins distribuée. Cette méthode permet de prendre en compte les modèles avec Condition.

La seconde méthode, α-core, utilise un processus par composant et un processus par interaction. Il serait intéressant d'étendre cette méthode afin qu'elle prenne en compte les modèles avec Condition.

De plus, l'optimisation proposée pour ce chapitre n'a été formalisée et expérimentée que dans le cadre du protocole α-core. Il serait intéressant de l'intégrer à la première méthode.

7 Implémentation

Ce chapitre présente l'implémentation de BIP, à travers le langage permettant de décrire les modèles BIP et les outils permettant de manipuler ces modèles.

Le langage BIP est utilisé pour décrire des modèles. Les briques de base sont les composants atomiques et les connecteurs qui sont décrit par leurs types. Une fois un type de composant déclaré, on peut en créer plusieurs instances. Ces composants sont alors connectés par des instances de connecteur qui sont définies à partir des ports exportés par les composants. Un ensemble de composants et de connecteurs crée un type de composant composé que l'on peut réutiliser pour former des composants hiérarchiques.

Un ensemble d'outils permet de générer, vérifier, transformer, optimiser et exécuter des modèles décrits dans ce langage. Une première catégorie d'outils permet de générer des modèles BIP à partir de modèles écrits dans d'autres langages. Les outils de vérification permettent de garantir qu'un système valide une propriété donnée, soit à l'aide d'invariants, pour des propriétés sur les états atteignables, soit à l'aide de model-checking stochastique, pour des propriétés sur les séquences d'actions exécutées.

Un modèle BIP ayant des composants hiérarchiques peut être transformé en un modèle équivalent ne comportant que des composants atomiques et des connecteurs. Une autre transformation permet de fusionner deux composants atomiques en un seul. En itérant cette transformation, on obtient un seul composant atomique équivalent au modèle d'origine.

Les méthodes pour décentraliser un modèle présentées dans cette thèses sont également implémentées comme des outils. Les tâches consistant à transformer des priorités en conditions, optimiser les conditions en utilisant la connaissance, générer un modèle distribué

utilisant les compteurs pour résoudre les conflits, générer un modèle distribué utilisant α-core sont implémentées par des outils dédiés.

L'exécution de modèles BIP peut se faire de manière séquentielle en utilisant un engin centralisé. Pour cela, on génère automatiquement du code C++ à partir du modèle à exécuter. Ce code fait appel à une librairie implémentant la sémantique de BIP au travers d'un engin chargé d'ordonnancer l'exécution du code généré. On peut également utiliser les méthodes décrites dans cette thèse pour générer un modèle distribué. À partir de ce modèle, on génère un programme pour chacun des composants. Ces programmes utilisent les primitives de communications disponibles sur la plateforme visée afin d'échanger des messages. Les plateformes pour lesquelles on peut générer du code incluent celles supportant les sockets POSIX, les thread POSIX ou MPI. Un autre prototype génère des scripts ASEBA permettant de programmer le robot Marxbot.

Pour l'instant, chacun de ces outils est implémenté de façon indépendante. On peut classer ces outils en 3 catégories : frontend, backend et middleend. Un flot de conception commence par un frontend, peut faire appel à plusieurs middleend et finalement produit une implémentation au moyen d'un backend. Cette approche requiert un outil permettant d'appeler ces différents modules avec les paramètres convenables.

8 Expérimentations

En utilisant les outils décrits dans le chapitre précédent, on propose d'évaluer les diverses implémentations obtenues. Ces évaluations portent sur l'influence des différents paramètres.

Une première série d'expériences compare les performances obtenues en prenant différentes partitions pour les interactions et différents protocoles pour la résolution des conflits. Certains temps d'attente sont insérés pour simuler des temps de calcul ou de communication, selon diverses configurations. Une même partition peut être optimale pour une configuration donnée et mauvaise pour une autre, ce qui suggère que le choix de la partition doit être effectué en fonction de la plateforme.

Une deuxième série d'expériences compare une implémentation obtenue par notre méthode avec une implémentation écrite à la main. Ces expériences sont effectuées sur des applications demandant beaucoup de calcul. On observe que la perte de performance par rapport à une version écrite à la main est assez faible. Dans certains cas, l'utilisation d'outils de transformations pour ajuster le nombre de processus générés au nombre de processeurs permet d'obtenir de meilleures performances.

Une troisième série d'expériences considère l'utilisation de l'opérateur de Condition et l'optimisation utilisant la connaissance qui lui est associée. Passer d'une implémentation n'implémentant que les interactions multiparties à une implémentation prenant en compte l'opérateur de Condition permet un gain très important de performance et limite le nombre de messages échangés. L'optimisation utilisant la connaissance se révèle efficace sur les modèles n'utilisant que les interactions multiparties. En revanche, pour les modèles utilisant la Condition, cette optimisation permet surtout de réduire le volume de communication nécessaire à l'exécution du système.

La dernière série d'expériences évalue le gain de performance obtenu en utilisant la connaissance avec mémoire parfaite. Suivant les modèles, la taille de l'automate encodant la connaissance peut varier. Les versions optimisées sont plus rapides que les versions originales.

9 Conclusion

Résultats

Flot de conception rigoureux pour une implantation distribuée

Dans cette thèse, nous présentons un flot de conception rigoureux, constitué d'une série de transformations. Chaque transformation s'occupe d'un aspect particulier et leur composition en chaîne permet de transformer un modèle haut niveau en un ensemble de programmes distribués. Le flot présente les caractéristiques suivantes.

Correction Les différentes transformations sont suffisamment simples pour être entièrement formalisées et prouvées. En particulier chaque transformation assure la préservation des propriétés fonctionnelles. De ce fait, vérifier la correction fonctionnelle du modèle haut-niveau de départ assure la correction de l'implantation distribuée générée en utilisant la chaîne d'outils.

Performance/Efficacité Les solutions présentées dans cette thèse permettent différents niveaux de décentralisation, notamment au niveau de la répartition des interactions dans les engins. Cela conditionne le nombre de composants dans le modèle distribué généré et donc le nombre de programmes dans l'implantation obtenue, permettant au concepteur d'adapter l'implantation pour la plate-forme visée.

De plus, plusieurs optimisations sont proposées. Ces optimisations s'appliquent à différents endroits du flot de conception et permettent d'améliorer la performance de l'implantation finale générée.

Productivité Le même modèle sémantique est commun à toutes les transformations, ce qui permet de les enchaîner. De plus, d'autres outils utilisant ce même modèle sémantique permettent d'accomplir d'autre tâches, telles que la vérification de certaines propriétés, l'aplatissement de la hiérarchie ou la fusion de deux composants.

Techniques d'optimisation fondées sur la connaissance

Deux optimisations sont présentées dans cette thèse. La première utilise la connaissance pour prendre une décision à partir d'une observation plus restreinte du système. Cette optimisation permet de réduire le volume de communication et dans certains cas d'améliorer les performances du système.

La deuxième optimisation intervient ultérieurement dans le flot de conception et permet de réduire le nombre de messages échangés pour résoudre les conflits. Dans certains

cas, la connaissance utilisée par cette optimisation permet de savoir qu'une seule interaction est possible parmi plusieurs potentiellement en conflit. Dans ce cas la version optimisée ne fait pas appel au protocole de résolution de conflit.

Implantation et évaluation sur des études de cas

Les transformations et optimisations présentées dans cette thèse ont été implémentées sous forme de prototypes. Cela a permis d'étudier les performances obtenues avec divers paramètres sur différents modèles.

Dans un premier temps, on a étudié l'influence du partitionnement des interactions sur les performances. La meilleure partition des interactions dépend fortement des caractéristiques de la plate-forme cible pour l'exécution.

En ce qui concerne les optimisations, le gain de performance le plus important est obtenu en passant d'un protocole de résolution de conflit classique à un protocole optimisé pour les Conditions. Les optimisations utilisant la connaissance permettent également d'augmenter les performances, mais de façon moins importantes. Cela prêche en faveur d'une extension des interactions multiparties prenant en compte la Condition, c'est à dire introduisant la notion de composant observé mais non participant dans une interaction. Une telle extension permet d'exprimer, entre autres, les priorités et est relativement peu coûteuse à implémenter.

Perspectives

Partitionnement et déploiement Choisir une partition revient à faire un compromis entre le niveau de parallélisme possible entre les interactions et le coût des communications pour résoudre les conflits. Une partition sans conflits minimise le coût de communication nécessaire pour résoudre les conflits, mais peut limiter de façon importante le niveau de parallélisme possible entre les interactions. D'une manière duale, on peut construire une partition préservant le parallélisme maximal autorisé par le modèle, mais au prix d'une charge importante pour résoudre les conflits.

En pratique, ce compromis doit être guidé par la plateforme de destination. Dans ce contexte, trouver une partition optimale peut être vu comme une extension du problème consistant a trouver un déploiement optimal d'une application donnée sur une plateforme donnée. Le partitionnement des interactions permet de moduler le nombre de processus, la charge de calcul correspondante et le volume des communications.

Systèmes temps réel Les modèle BIP peuvent être dotés d'une sémantique temps réel. Dans ce cas, les composants atomiques sont étendus par des horloges et chacune des transitions a une contrainte temporelle exprimée en fonction des horloges du composant. Chaque interaction doit alors s'effectuer dans la zone correspondant à l'intersection des contraintes de chacune des transitions qui la composent.

Les modèles BIP temporisés peuvent être exécutés soit par un seul processus ou par une implémentation multi-thread avec un engin centralisé. Dans ce cas, l'engin a une vue partielle du système, comme décrit au Chapitre 5. Avant d'effectuer une interaction,

il doit s'assurer qu'aucune interaction plus urgente n'est possible. On peut tirer un parallèle avec les priorités, où il faut s'assurer qu'aucune interaction plus prioritaire n'est possible. Dans le cas des priorités, il est toujours possible d'attendre que tous les composants aient envoyé une offre avant de prendre une décision. Pour les contraintes temporelles, cette stratégie peut entrainer le dépassement de certaines échéances. L'engin centralisé reste toutefois capable de détecter ces dépassements.

Décentraliser un engin distribué temps-réel est d'un niveau de difficulté supérieur. Avant d'exécuter une interaction, il faut s'assurer qu'aucune interaction en conflit n'est possible plus tôt. De plus, détecter les dépassements d'échéances a posteriori devient particulièrement délicat, car cela nécessite de pouvoir reconstruire l'ensemble des offres valides à un instant donné.

Exécutions distribuée stochastique On peut également considérer l'exécution distribuée de modèles stochastiques. Par exemple, un processus de décision markovien s'exécute en faisant à chaque étape un choix non-deterministe puis un choix probabiliste. Dans notre cas, on peut choisir de manière non-deterministe un composant, puis de manière probabiliste l'une des interactions auxquelles il participe. L'implantation distribuée d'un tel processus peut être parallélisée en choisissant plusieurs composants. Toutefois deux composants ne peuvent être choisis simultanément s'il existe deux interactions en conflit, impliquant chacune l'un des composants.

Alternative à la communication par envoi de message Dans cette thèse, nous avons considéré uniquement des systèmes distribués communicant par envoi de messages. Ce cadre est assez général, car la plupart des plateformes proposent des primitives permettant de communiquer par envoi de messages, même si d'autre primitives sont disponibles. Les plateformes massivement multicœurs proposent par exemple de la mémoire partagée à plusieurs niveaux. Ces plateformes sont traditionnellement divisées en unités synchrones, comportant plusieurs processeurs communicants par mémoire partagé ; la communication entre ces unités se fait par envoi de message asynchrones. Un piste possible est d'exploiter les plateformes proposant de la mémoire partagée, et, finalement, permettre de concevoir des implantations mélangeant divers types de communications.

1 Introduction

Systems designed today often require distributed computation. The main reason for considering distributed systems is undoubtedly the efficiency. Doubling the frequency of processors consumes twice as much energy as doubling the number of processors while achieving comparable speedups provided the software is efficiently distributed.

Another reason for considering distributed systems lies in the geographical location of the sensors and the actuators. Processing sensor data and controlling actuators requires dedicated computing units in specific locations.

When implemented, a distributed system software consists of a set of autonomous and independent processes. At each step, a process executes one of three different types of action: sending a message, performing a local computation and waiting for an incoming message. The decision of the next action to execute is taken locally by the process, depending on the messages so far received and the computation results.

Programing directly using Send/Receive primitives requires several difficult problems to be handled at the same time. The first problem is to decompose the application into independent parts. Next, the designer has to define how the different parts communicate to ensure correct execution of the application, that is to design a protocol. Finally, he has to implement this protocol, by taking into account all possible interleavings of messages. Each of these tasks is complex and tedious.

The implemented system must provide the functionality it was designed for, i.e. meet the specifications. Ideally, specifications are expressed over the final system. In practice, specifications are formalized using a model of the system. Reasoning in terms of processes communicating through asynchronous message-passing impedes the verification and even the formulation of these properties. Verification at this level becomes intractable owing to the considerable amount of possible message interleavings.

In this thesis, we propose a methodology relying on a design flow to avoid the above pitfalls. The design flow should provide significant help in addressing the following challenges that arise when designing a distributed system.

Correctness In order to define how the system should operate, the designer provides requirements and specifications. These specifications are often written in English; the designer then formalizes them into a set of properties that must be verified by the implementation. For the designer to have confidence in its formal specification, the model or language used must be high-level enough so that the formal properties are simple to write and understand.

In the context of distributed systems, a property may depend on several processes. Verifying a distributed system is very difficult owing to the size of the state space obtained when considering all message interleavings. Therefore, correctness cannot be

ensured a posteriori and must be considered from the beginning of the design process.

Performance/Efficiency The system should utilize resources in an optimal way to ensure the best performance at the lowest cost. The design flow should enable the designer to parametrize as much as possible the deployment of the application software over the hardware. In particular, the underlying model should provide methods for deploying the same application over different platforms. Ideally, ensuring optimal deployment of the software on a platform should be orthogonal to ensuring its functional correctness.

Productivity The design process should be focused on tasks where human intervention is needed. In particular, tedious implementation tasks should be automated as much as possible to allow designers to concentrate on important design choices.

The idea that message-passing primitives are of too low a level for programing is expressed in [47]. In that paper, the author advocates the use of Message-Passing Interface (MPI) [41] collective operations. A collective operation is a strong synchronization between a subset of the processes and may involve data transfer. A process locally chooses and commits to a single collective operation, then waits until the operation executes. All processes must make the same choice for the collective operation to take place.

We consider higher-level application models where each step is a multiparty interaction [56], that is a strong synchronization between processes. Contrarily to MPI collective operations, the next interaction is chosen non-deterministically among the enabled ones at the current global state. More precisely, at each state, a process proposes a set of interactions to execute. An interaction is enabled if all participant processes propose to execute it. At each step, an enabled interaction is chosen for execution.

These models are described within a component-based framework, namely Behavior Interaction Priority (BIP) [10]. Formal verification methods allow validation of functional properties of a system described using such a framework. Rigorous system design [80] allows the building of a distributed implementation preserving these properties by successive applications of transformations. We present the concepts behind rigorous system design, and then detail the design flow for obtaining a distributed implementation from a BIP model.

1.1 Rigorous System Design

Rigorous system design [80] relies on a sequence of transformations guaranteeing that essential requirements are met by the implementation. Figure 1.1 depicts an abstract design flow starting from an application software and leading to an implementation. A framework allowing rigorous system design exhibits the following characteristics.

Model Based. The design flow relies on a single semantics that is used consistently throughout the flow. For instance, in Figure 1.1, the application software and the intermediate model n are written using a common syntax and rely on the same semantic

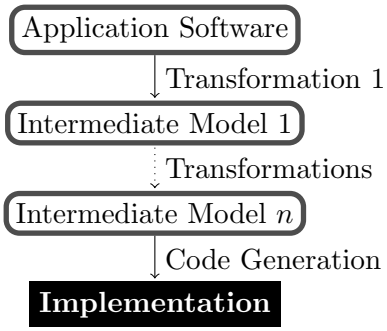

Figure 1.1: A design flow.

rules. Intermediate models have different levels of abstraction, suitable for performing various tasks such as verification, performance analysis and code generation. The last model before code generation represents exactly the software to be implemented. Code generation only implements the semantics of this model using a low-level programming language such as C++.

Component Based. Basic blocks of the systems are enclosed in components that have a well-defined interface. Components allow the embedding and the composition of heterogeneous pieces of software. Furthermore, built components can be reused in ulterior designs.

Correct-by-construction. Each transformation preserves the functional properties of the model. Correctness-by-construction guarantees that the intermediate model n is functionally equivalent to the application software. In our case, the equivalence is either trace equivalence or observational equivalence.

Finally, the design flow is implemented through a set of tools automatically performing the different transformations. The designer provides a high-level model of the system from which the implementation is generated. In particular, the mechanisms to handle messages are generated automatically during the transformations. Human intervention is required to choose the parameters of the transformations.

1.2 Design Flow for Building Distributed Systems

We focus on the part of the design flow that allows us to generate a distributed implementation from a high-level model of the application software. A distributed implementation is represented by a set of components communicating through interactions restricted to asynchronous message-passing. The challenge is to switch from the high-level model, where multiparty interaction is a primitive, to the distributed model, where only message-passing is allowed. We present each step of the flow and the optimizations that can be performed at these steps.

Input Model

Figure 1.2 depicts an abstract model made of 4 components ($B_1, \ldots, B_4$). Each component is described as an automaton or a Petri net equipped with guards and update functions on transitions. Components are composed using two composition operators, namely Interaction and Priority. The Interaction operator is parameterized by a set of interactions, which is $\{a, b, c\}$ in our example, that are synchronizations of components' transitions. The Priority operator is parameterized by a partial order on the interactions, which is $a < c$ on our example. Such a model has a global state semantics. A

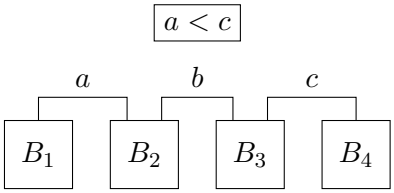

Figure 1.2: A simple example of input model.

semantic step is done through an interaction that changes atomically the states of all participants in the interaction. From a global state, an interaction can execute if:

- All its participants are ready to execute that interaction. For instance, executing interaction a requires that B_1 and B_2 are ready to execute a transition allowing interaction a. In that case, the interaction is *enabled*.
- No higher priority interaction is enabled at that global state. For instance, interaction a cannot execute if interaction c is enabled.

Interactions execute atomically; the state of all participants in the interaction is changed in a single execution step. The states of other components are not modified. From the reached global state, another enabled interaction is chosen to perform the next execution step.

We define another composition operator named Condition, which, like Priority, is defined over components composed by Interaction. Condition associates a predicate κ_a to each interaction a. The predicate must be true for the interaction to execute. The predicate depends on the states of some components. Components whose state is observed to decide whether an interaction can execute, but which are not participant in the interaction, are said to be observed by the interaction. Components observed by an interaction influence its execution, but are not modified during it. Using the Condition operator defines precisely the role of each component (participant or observed) in an interaction.

Priority forbids the execution of a low priority interaction if there is a higher priority interaction enabled. The condition operator encodes Priority by assigning, to a low priority interaction, the predicate that is true when no higher priority interaction is enabled. In our example, the predicate κ_a, stating that "c is not enabled", is assigned

to interaction a. For interaction a, components B_1 and B_2 are participants whereas components B_3 and B_4 are observed.

Breaking the Atomicity of Interactions

BIP semantics requires knowing the global state of the system to take a decision about the next interaction to execute. In a distributed setting, there is no instance knowing the global state of the system. Therefore, each component explicitly sends the set of interactions it can perform, which we call *offer*, to a centralized engine E, as shown in Figure 1.3. Initially, the engine has no information about the current state of the system, but builds a partial view of the state based on the offer received. When enough information is available, the engine selects a feasible interaction for execution and notifies the participating components to execute. After sending a notification to a component, the engine has no information about its state until the next offer.

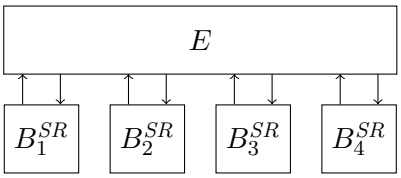

Figure 1.3: First step: breaking the atomicity of interactions.

After this step, the model contains only Send/Receive interactions between components, the original interactions of the model are transformed into actions in the engine. By considering actions in the engine as observable, and Send/Receive interactions as unobservable, the distributed model is equivalent to the original one. In a centralized context, the execution of an interaction consists in the sequential execution of the corresponding local computation in each component. In the distributed model, components run concurrently and execute these local computations in parallel.

Optimization: Taking Decision Earlier

Evaluating a Condition predicate in the engine requires knowing the states of the observed components, and therefore waiting for their offers. Note that the Condition predicate κ_a associated to interaction a in our example is "c is not enabled". Its evaluation requires waiting for offers from B_3 and B_4. Thus, for executing interaction a, the engine requires offers from all components of the model. The proposed optimization aims at reducing, for each interaction, the number of observed components.

The concept of knowledge [40] has been extensively studied for distributed systems with respect, in particular, to their ability to execute actions [50]. Distributed Knowledge [51] allows a set of components to "know" that a predicate holds based on a partial observation. We assume that each interaction a observes the states of a set of components L_a comprising participants in a. The knowledge predicate denoted $K_{L_a}\kappa_a$ characterizes the states where observing only components in L_a is sufficient to ensure that κ_a holds.

In other words, it characterizes states where the distributed knowledge of the set of components L_a is sufficient to safely execute a. Restricting too much the set L_A might lead to cases where the distributed knowledge is not sufficient to ensure that a can execute. We propose two detection levels that characterize how much of the original interactions are also allowed in the knowledge-based version. Basic detection ensures that no deadlocks are introduced. Complete detection ensures observational equivalence with the original model.

Decentralizing the Engine

With the centralized engine, no parallelism between interactions is allowed. Enabling parallelism between interactions requires their execution in separate engines. We decentralize the engine by partitioning the interactions and building one engine for each class of the partition. For instance, the partition $\{a, b\}, \{c\}$ yields a distributed implementation of our example with two engines, as depicted in Figure 1.4.

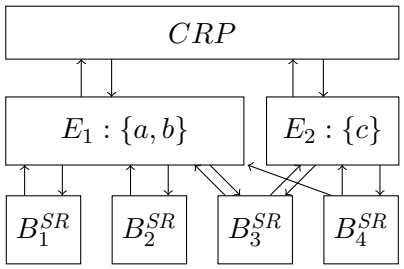

Figure 1.4: Second step: decentralizing the engine.

Having interactions executed in several engines creates conflicts. Generally speaking, a conflict occurs when two entities are competing for a common resource. In the present case, a conflict occurs when two interactions involving a common component are handled in separate engines. The conflicting resource is the common component that can take part in only one interaction. If two engines execute simultaneously two different interactions involving a common component, the semantics is broken because this behavior is not allowed in the centralized model.

We solve this problem by adding a conflict resolution protocol (see Figure 1.4). Before executing a conflicting interaction, the engine asks the conflict resolution protocol. The latter grants the execution only if it does not break the semantics. Our solution accommodates several implementations for the conflict resolution protocol, based on Bagrodia [6]. In particular, we consider distributed conflict resolution relying either on a token ring or on a solution to the dining philosophers problem.

Optimizing Conflict Resolution

In the above solution, an engine systematically calls the conflict resolution protocol before executing an externally conflicting interaction. However, there might be some

states where components causing conflicts have no alternative interaction from their local state. In that case, no conflict resolution is required for that component. This case is similar to the MPI collective operations where a component commits to a single interaction. A component can use a special message to notify that it commits to an interaction, allowing the receiving engine to efficiently handle this particular offer. The α-core protocol [73] implements such a mechanism.

There are some states where a component has locally a choice between two interactions, but where the global state actually only allows one interaction. A component augmented with knowledge with perfect recall can detect some of these states and send an offer only for the allowed interaction. Reducing the number of possible interactions in the offer leads to a more efficient implementation because it diminishes the number of actual conflicts to resolve.

Code Generation

The final step of our transformation generates distributed code for a given platform from a Send/Receive model. A Send/Receive model consists of a set of components communicating through interactions representing asynchronous message-passing. Each component is an automaton or a Petri net whose transitions correspond either to sending or receiving a message.

The generated code contains one standalone program for each component of the Send/Receive model. Such a program implements the Petri net or automaton representing the behavior of the component. Transitions corresponding to send actions are executed as soon as they are enabled. If no send action is possible, the component waits for a message that triggers a receive action.

Code generation is the same regardless of the role (component, engine, conflict resolution) of the distributed component considered. This scheme for generating code can be applied to any platform providing message-passing. We generate code for general purpose platforms (MPI, sockets) and code for domain specific platforms (ASEBA scripts [64] for the Marxbot robot [25]).

1.3 Organization

The Chapters 2, 3 and 4 introduce multiparty interactions, knowledge and the BIP framework,respectively , which constitute the prerequisites of this thesis. The main contribution consists of the design flow, presented in Chapters 5 and 6, the tools, presented in Chapter 7, and their evaluation in Chapter 8.

Chapter 2 formally presents the semantics of multiparty interaction, expressed as a global state semantics. When attempting to implement the latter in a distributed manner, conflicts between interaction appear. Therefore, distributed execution of multiparty interactions is controlled by a protocol. Several protocols are outlined in the chapter.

Chapter 3 presents the notion of knowledge, that is the set of facts known by a part of a system. This notion applies naturally to distributed systems, where, by definition, each

process acts upon the information it receives from other processes. The chapter presents two concrete examples of knowledge, their representation and their computation.

In Chapter 4 we provide the abstract and concrete models of the BIP framework. The BIP framework provide a single semantics used consistently throughout the design flow. We also introduce the Condition operator, which is an alternative to Priority.

The first step towards decentralization consists in letting components run in parallel. In Chapter 5, we present a decentralized solution relying on a centralized engine responsible for scheduling all interactions. The obtained distributed model is observationally equivalent to the original model. We provide a knowledge-based optimization that only modifies the Condition operator and reduces the number of exchanged messages.

The next step towards decentralization consists in splitting the centralized engine into several distributed engines. As explained in Chapter 6, partitioning interactions into engines creates conflicts. Thus, distributed models built in this chapter include conflict resolution protocols. We prove the correctness of the final distributed model through trace equivalence. We present another protocol for executing multiparty interaction in a distributed context, namely α-core, that can be embedded in a BIP model. Finally, we show another knowledge-based optimization that aims to reduce the number of messages exchanged to resolve conflicts.

Chapter 7 presents the set of tools involved in the BIP framework in general. It focuses on tools related to the distributed implementation of BIP. The methods for generating distributed code are explained for various platforms. The tools are evaluated in Chapter 8. In particular, we assess the actual performance gains induced by the aforementioned optimizations, the influence of partitioning, and the performance of the various platforms.

Finally, we conclude and outline some future works in Chapter 9.

2 Multiparty Interactions

Multiparty interactions provide a high-level description of a distributed system in terms of processes and interactions. An action of the system is an interaction, which is a coordinated operation between an arbitrary number of processes.

In [56], Joung and Smolka classified different types of multiparty interactions. According to their taxonomy, we focus only on *gate* or *multi-channel* interaction constructs, where each interaction has a fixed set of participant processes.

The problem of executing multiparty interactions in a distributed context has been studied extensively in [5, 6, 32, 33, 55, 61, 70, 73]. These works state the problem and establish correctness of protocols for multiparty interaction execution in a distributed context only, without assuming an underlying global state semantic of the system.

In this chapter, we propose a more holistic approach. We first present in Section 2.1 an abstract formalization of multiparty interactions, based on a global-state execution of the system. Section 2.2 present the traditional view of multiparty execution in a distributed context. In the presented approach, the correctness of a distributed implementation is obtained by comparing it with the previous formalization. We then present in Section 2.3 some of the existing protocols implementing unprioritized multiparty interactions. The Section 2.4 discusses an extension that adds priorities to multiparty interactions. Finally, we consider another possible extension to multiparty interactions and other frameworks for building distributed systems in Section 2.5.

2.1 Specification Model

Our specification model consists in a set of processes. In general, processes are computing artifacts that, at some point of their execution, need to coordinate one of their actions with some other processes of the system. In the sequel, we represent processes as Labeled Transition Systems (LTSs), where transitions are labeled by actions that need coordination. In particular, using LTSs abstracts away the inner computation of the processes and allows us to focus on the interaction scheme.

Definition 2.1 (Process)**.** A process $\mathcal{P}$ is defined by a tuple $(Q, A, \rightarrow)$, where:

- Q is a set of states.
- A is a finite set of actions.
- $\rightarrow \subseteq Q \times A \times Q$ is a set of transitions.

Given two states q, q' and an action a, we denote the transition $(q, a, q') \in \rightarrow$ by $q \xrightarrow{a} q'$. We also denote by $q \xrightarrow{a}$ (resp. $q \not\xrightarrow{a}$) if there exists (resp. doesn't exist) a transition

labeled by a outgoing from state q. Throughout this thesis, we assume that processes are deterministic. In the current context, deterministic means that given a state q there is at most one outgoing transition from q labeled by a.

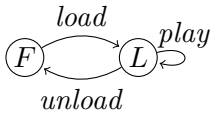

Figure 2.1: LTS representing a process.

Example 2.2. In Figure 2.1 we depict a process $\mathcal{P} = (Q, A, \rightarrow)$. This process represents a disc, whose state is either free (F) or loaded (L), i.e. $Q = \{F, L\}$. The set of actions of the process is $A = \{load, unload, play\}$. The transitions are $\rightarrow = \{(F, load, L), (L, unload, F), (L, play, L)\}$. At state L, the actions play and unload are enabled. If the process executes the action unload, it state changes to F.

An *interaction* is an action synchronized between several processes. In the adopted formalism, an interaction is denoted by a common action symbol occurring in several processes. That is, all processes that have a in their set of actions participate in interaction a. Interaction a is enabled only if all its participants are in a state allowing a transition labeled by a; its execution correspond to the synchronous execution of an a-labeled transition by all participant processes.

Definition 2.3. A distributed system is given by a set of processes $\mathcal{P}_1, \ldots, \mathcal{P}_n$, where $\mathcal{P}_i = (Q_i, A_i, \rightarrow_i)$, with pairwise disjoint state sets: $\forall i, j \ 1 \leq i, j \leq n \wedge i \neq j \implies Q_i \cap Q_j = \emptyset$.

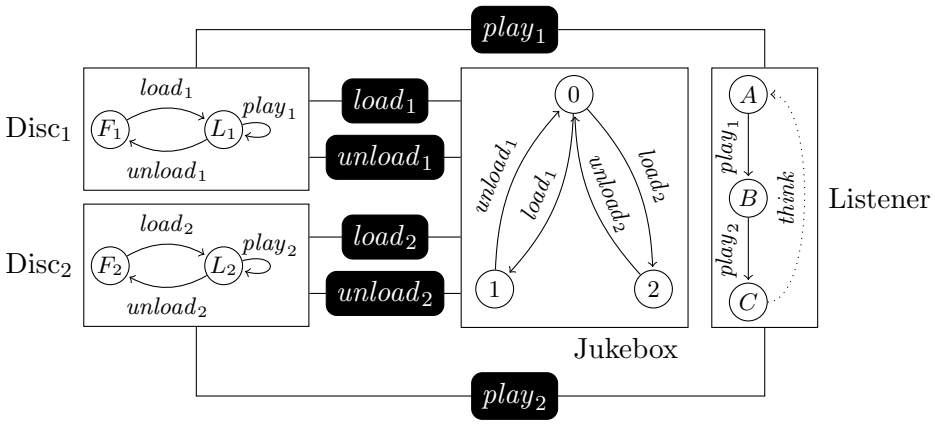

Figure 2.2: A model with multiparty interactions.

Example 2.4. Figure 2.2 depicts a system with 4 processes and 7 interactions. This system contains a Listener process, that plays Disc_1 then Disc_2, then thinks. Each Disc

is a separate process. Playing a disc is possible only if it has been loaded in the Jukebox. In order to make interactions visible, they have been added as black boxes on Figure 2.2. For each box, the set of bound processes denotes the participants in the corresponding interaction. We did not add a box for the unary interaction *think* as it involves only the Listener process.

Assume that initially both discs are free (places F_1 and F_2 are active), the Jukebox process is at state 0 and Listener at state A. From that state, interaction $load_1$ can take place since in both participants (Disc_1 and Jukebox) there is an interaction labeled by $load_1$ from the active place.

We now formally define the behavior of distributed system. We denote by $I_a = \{i | a \in A_i\}$ the indices of processes that participate in a.

Definition 2.5 (Global behavior of distributed systems)**.** The behavior of a distributed system $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$, with $\mathcal{P}_i = (Q_i, A_i, \rightarrow_i)$ is a LTS $(Q, \mathcal{I}, \rightarrow)$ where:

- $Q = Q_1 \times \ldots \times Q_n$: the set of global states is obtained by the cartesian product of the states of the processes.
- $\mathcal{I} = \bigcup_{i=1}^{n} A_i$: the interactions are the union of all processes actions.
- $\rightarrow \subset Q \times \mathcal{I} \times Q$ is the least set of transitions satisfying the rule:

$$\frac{a \in \mathcal{I} \qquad \forall i \in I_a \; q_i \xrightarrow{a}_i q_i' \qquad \forall j \notin I_a \; q_j = q_j'}{(q_1, \ldots, q_n) \xrightarrow{a} (q_1', \ldots, q_n')}$$

We say that an interaction a is enabled at global state q if all its participants are ready to perform an action labeled with a. As for processes, we use the notation $q \xrightarrow{a}$ to denote that a is enabled at *global* state q and the notation $q \overset{a}{\not\longrightarrow}$ to denote that a is not enabled at global state q. The definition of the transitions expresses that moving from one state to another is done by executing an enabled transition. It consists of synchronous execution by all participants in a of their a-labeled transition. Furthermore, this transition is atomic in the sense that non-participating processes do not alter their state during this interaction.

In addition to the operational semantics of the system given by the composed transition relation, we have to provide an initial state for each process.

In Figure 2.3, we present the global behavior of the system depicted in Figure 2.2. The global state of the system is defined by the local state of each individual process. For instance, the initial state is $(0, F_1, F_2, A)$, meaning that process Jukebox is in state 0, process Disc_1 in state F_1, process Disc_2 in state F_2 and process Listener in state A. Each arrow represents the execution of an interaction. The only modification between the source and target of an arrow is the state of processes participant in the corresponding interaction.

Example 2.6. Figure 2.4 presents the beginning of a global execution of the system depicted in Figure 2.2. On this Figure, each process corresponds to a vertical line,

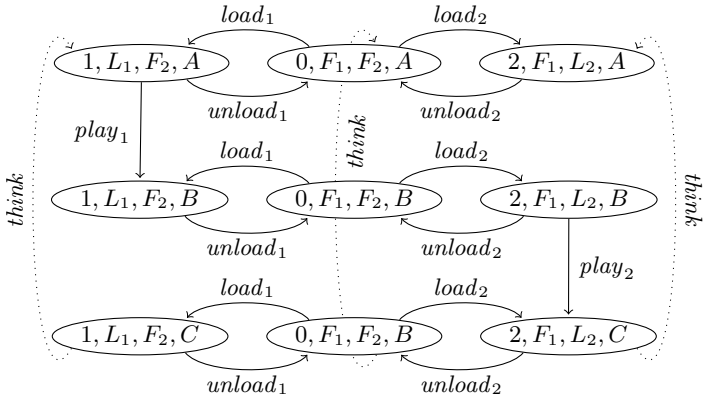

Figure 2.3: Global behavior of system from Figure 2.2.

that indicates the evolution of the process as the time passes. A global state execution consists in a sequence of global steps. At each state, the set of possible transitions for each process is represented by an outgoing arrow. From the initial state, both $load_1$ and $load_2$ are enabled. The interaction $play_1$ is not enabled because Disc_1 cannot execute a transition labeled by $play_1$. First, $load_1$ is executed, thus Disc_1 and Jukebox execute their corresponding transition. Disc_2 and Listener do not move because they did not take part in an interaction.

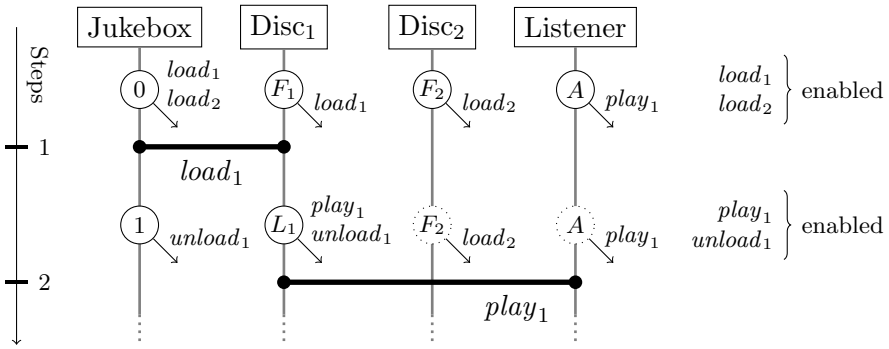

Figure 2.4: A centralized execution of the distributed system from Figure 2.2.

The global behavior of systems encompassing multiparty interactions is defined based on a global view of all the processes. This representation is useful to define the semantics of such systems. However, in a distributed setting, we cannot assume a synchronization between independent processes.

2.1.1 Link with Petri Nets

We recall here the definition of a Petri net. We show that the global behavior of a system encompassing multiparty interactions can be expressed as a Petri net.

Definition 2.7. A *Petri net* is defined by a triple $S = (L, \mathcal{I}, T)$ where L is a set of *places*, $\mathcal{I}$ is a set of labels, and $T \subseteq 2^L \times \mathcal{I} \times 2^L$ is a set of transitions. A transition τ is a triple $({}^\bullet\tau, a, \tau^\bullet)$, where ${}^\bullet\tau$ is the set of *input places* of τ and $\tau^\bullet$ is the set of *output places* of τ.

A Petri net is often modeled as a directed bipartite graph $G = (L \cup T, E)$. Places are represented by circular vertices and transitions are represented by rectangular vertices (see Figure 2.5). The set of directed edges E is the union of the sets $\{(\ell, \tau) \in L \times T \mid \ell \in {}^\bullet\tau\}$ and $\{(\tau, \ell) \in T \times L \mid \ell \in \tau^\bullet\}$.

We depict the state of a Petri net by *marking* its places with *tokens*. Formally, a *marking* is an application $m : L \to \mathbf{N}$ that indicates how many tokens are in each place. We say that a place is *marked* if it contains a token. A transition τ is enabled if all its input places are marked. Formally, τ is enabled if $\forall \ell \in {}^\bullet\tau \; m(\ell) > 0$.

At a given marking m, any enabled transition can be executed. Executing a transition τ corresponds to removing one token in each input place and adding one token in each output place. Formally, by executing τ at marking m, one reaches the marking m' characterized by:

$$\forall \ell \in L \quad m'(\ell) = m(\ell) - \tau^-(\ell) + \tau^+(\ell)$$

where

$$\tau^-(\ell) = \begin{cases} 1 & \text{if } \ell \in {}^\bullet\tau \\ 0 & \text{otherwise} \end{cases} \qquad \text{and} \qquad \tau^+(\ell) = \begin{cases} 1 & \text{if } \ell \in \tau^\bullet \\ 0 & \text{otherwise.} \end{cases}$$

We denote $m \xrightarrow{a}_S m'$ if the transition $\tau = ({}^\bullet\tau, a, \tau^\bullet)$ can be executed at marking m and reaches marking m'. We denote by $\to_S$ the set of triples (m, a, m') such that $m \xrightarrow{a}_S m'$. The behavior of a Petri net S can be defined as an (infinite) labeled transition system $(\mathbf{N}^L, \mathcal{I}, \to_S)$, where $\mathbf{N}^L$ is the set of states[1], A is the set of labels, and $\to_S$ is the set of transitions. The interested reader can find a survey about Petri Nets in [69].

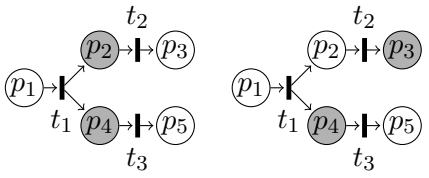

Figure 2.5: A simple Petri net in two successive markings.

Example 2.8. Figure 2.5 shows an example of a Petri net in two successive markings. It has five places $\{p_1, \ldots, p_5\}$ and three transitions $\{t_1, t_2, t_3\}$. The places containing a token are depicted with gray background. The marking on the right shows the resulting state after executing transition t_2.

Given a Petri net $S = (L, \mathcal{I}, T)$ and an initial marking m_0, the marking m is *reachable* if there exists a sequence of transitions $m_0 \xrightarrow{a_1}_S m_1 \xrightarrow{a_2}_S \ldots \xrightarrow{a_k}_S m$. We say that S

[1]In this thesis, we denote by Y^X the set of all applications from the set X to the set Y.

with the initial marking m_0 is *1-safe* if in all reachable markings there is at most one token per place. Note that a 1-safe Petri net has at most $2^{|L|}$ markings. In this thesis, we focus on *1-safe* Petri nets.

Given a system of processes $\mathcal{P}_1, \ldots, \mathcal{P}_n$, where $\mathcal{P}_i = (Q_i, A_i, \rightarrow_i)$, one can build a Petri net $S = (L, \mathcal{I}, T)$ which has the same semantics. This is done by considering the following:

- $L = \bigcup_{i=1}^{n} Q_i$: the set of places is the union of the processes states,
- $\mathcal{I} = \bigcup_{i=1}^{n} A_i$: the set of labels is the set of interactions in the system,
- For each interaction $a \in \mathcal{I}$, T contains the set of transitions

$$\left\{ \left(\bigcup_{i \in I_a} q_i, a, \bigcup_{i \in I_a} q_i' \right) \;\middle|\; \forall i \in I_a \; q_i \xrightarrow{a}_i q_i' \right\}.$$

  For each set of transitions $q_i \xrightarrow{a}_i q_i'$ involving exactly once each participant in a, T contains a Petri net transition moving tokens from places q_i to places q_i'.

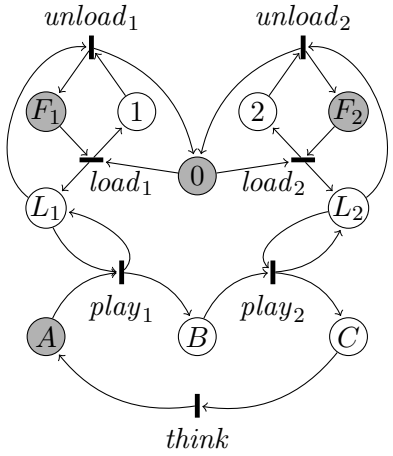

Figure 2.6: Petri net obtained from the example in Figure 2.2.

Example 2.9. The system of processes depicted in Figure 2.2 yields the Petri net shown in Figure 2.6. In this particular case, there is only one transition for each interaction. The initial state of the system $(F_1, F_2, 0, A)$ is encoded as the marking where the corresponding places contain one token. These places are depicted with a gray background on the figure. Consider the places corresponding to a given process, for instance the places $\{A, B, C\}$ corresponding to the process Listener. The LTS of the process can be reconstructed by transforming the Petri net transitions adjacent to the process places into labeled transitions.

In the previous construction, each process $\mathcal{P}_i$ corresponds to the subset L_i of places. Each transition either does not involve any place from L_i, or it removes and places exactly one token in L_i. Therefore, the number of tokens within L_i remains constant throughout execution. Intuitively, a state $(q_1, \ldots, q_n)$ of the system of processes corresponds to a marking where exactly one place of each subset L_i contains a token. Thus, starting from the marking corresponding to the initial state, one can only reach markings corresponding to global states of the system. In other words, with such an initial marking the obtained Petri net is 1-safe.

We compare the behavior of the system of processes $\mathcal{P}_1, \ldots, \mathcal{P}_n$ and the behavior of the corresponding Petri net. As stated before, we associate to each global state $q = (q_1, \ldots, q_n)$ the marking

$$m_q : \ell \mapsto \begin{cases} 1 & \text{if } \ell \in \{q_1, \ldots, q_n\} \\ 0 & \text{otherwise.} \end{cases}$$

Executing the transition $q \xrightarrow{a} q'$ in the global behavior of the system of processes changes only the states $(q_i)_{i \in I_a}$ of the participants in a to the states $(q'_i)_{i \in I_a}$ through the set of processes transitions $q_i \xrightarrow{a}_i q'_i$. The corresponding Petri net contains by construction a transition $\tau = (\bigcup_{i \in I_a} q_i, a, \bigcup_{i \in I_a} q'_i)$. We have equivalently $q \xrightarrow{a}$ and τ enabled at the corresponding marking m_q. The marking m' reached after executing τ removes tokens from places q_i and puts them in places q'_i therefore $m' = m_{q'}$. Thus, the behavior of the system of processes and the behavior of the built Petri net are the same.

Note that switching to Petri net formalism removes boundaries between processes, which are crucial for generating distributed systems. Furthermore, since the same interaction may correspond to several Petri net transitions, the representation with processes is more compact. In the sequel, we use the Petri net formalism to compute invariants of the system.

2.2 Distributed Execution

Distributed execution assumes a set of computational entities communicating through asynchronous message-passing. Each of these entities may decide either to execute internal actions, to send messages to other entities or to wait for incoming messages. In the global state model, the local execution of a transition labeled by a occurs only if interaction a executes globally. In contrast, distributed execution assumes that executing a local action (internal, send or wait) is decided based on the local state only, independently of other entities state.

2.2.1 Distributed Processes

In a distributed context, each process of the global state model becomes a distributed process that communicates with the environment to ensure correct execution with respect to the global state model. Thus, a reasonable assumption is that each distributed process publishes the list of available actions and waits for a decision on the interaction

to execute. Such a behavior is obtained by splitting each process transition in two parts; one part publishes the offer and the other part executes the chosen action. This transformation adds busy states as shown in Figure 2.7. Indeed, for each original state q of the process, there is a new busy state denoted $q^{\perp}$, that is reached before attaining q. From these busy states, the only possible action is to publish a list of actions indicating what is possible in the next state. Then, the process is in a stable state, waiting for the next interaction to happen.

The Figure 2.7 depicts the distributed version of the process from Figure 2.1. The distributed process starts in state $F^{\perp}$, from which it can only publish the interactions it can do from state F. In that case, there is only one possibility, that is *load*. After publishing its offer, i.e. $\{load\}$, the process reaches the state F where it waits for the message indicating to execute *load* before resuming execution.

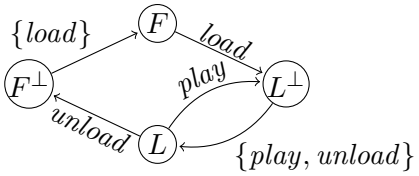

Figure 2.7: Distributed version of the process from Figure 2.1.

In most papers proposing protocols for multiparty interactions [5, 6, 55, 61, 73], a distributed process has only two states: idle and active. The transition to idle from active correspond to publishing the offer, the transition from idle to active correspond to starting an interaction. Thus, for the remaining of this section, we only specify the offers sent by the distributed processes.

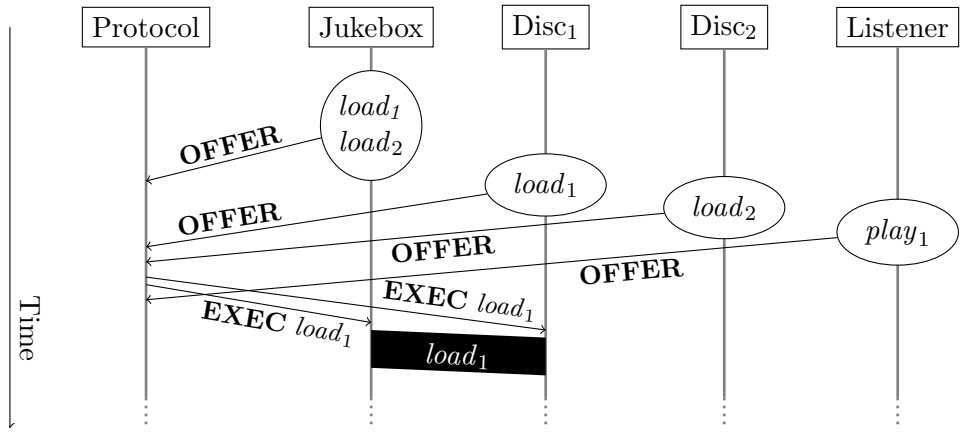

Figure 2.8: Beginning of a possible distributed execution of the model from Figure 2.2.

Example 2.10. Figure 2.8 shows the beginning of a distributed execution of the multiparty interaction system from Figure 2.2. Execution of interactions is done through a

protocol which listens to offers sent by the processes. On this figure, the protocol is represented as a single separate process. Alternative solutions provides protocol distributed among several processes or even embedded in each process of the system. Note that we do not assume a global view of the system since the protocol decides based on received messages. Furthermore, decision may be based on a partial view of the system, e.g. the execution of $load_1$ is decided before receiving the offers from Disc_1 and Listener.

2.2.2 Committee Coordination Problem and Conflict Resolution

The committee coordination problem, stated by Chandy and Misra in [33], describes the general problem that such a protocol has to solve. A set of professors need to attend a set of committees. Each committee requires full attendance to take place. Each professor cannot attend two meetings simultaneously. At some point, a professor indicates in which committees he wishes to participate next (i.e. his offer) and waits until one of the committee happens. The problem is to devise a protocol such that if all members of a committee are waiting to attend it, then at least one of these members will eventually take part in a committee.

Similarly to meetings involving a common professor, interactions that involve a common process are *conflicting*. Conflicting interactions cannot execute in parallel. This problem does not arise in the global behavior from Definition 2.5 because each interaction is executed atomically. However, the problem appears in a distributed execution context where separate entities are responsible for scheduling conflicting interactions. Consider conflicting interactions $load_1$ and $load_2$ that both involve process Jukebox. If the two interactions are simultaneously scheduled by two separate entities, then the process Jukebox will receive two different **EXEC** messages, one for executing $load_1$ and one for executing $load_2$. This clearly breaks the global state semantics where only one of the interactions can take place.

A *conflict graph* is a convenient representation of conflicts between multiparty interactions. Interactions are represented as nodes of the graph and edges show the conflicting processes. Figure 2.9 shows the conflict graph of the model presented in Figure 2.2. For instance, we see that $play_1$ and $load_2$ are not conflicting, and that $play_1$ and $play_2$ are conflicting because of the process Listener.

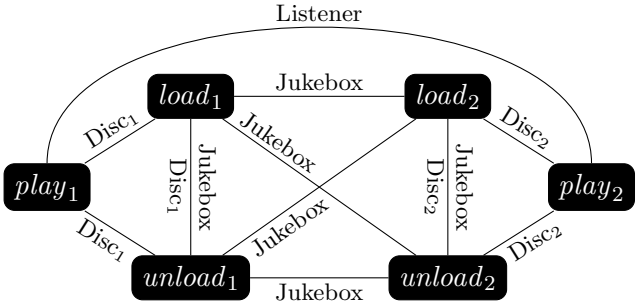

Figure 2.9: Conflict graph for the model in Figure 2.2.

2.2.3 Correctness

Many papers, e.g. [5, 6, 55, 61, 73] state the correctness of a distributed multiparty interaction protocol as the combination of the following properties:

- *Exclusion:* After scheduling the interaction a, all participant processes cannot execute another interaction until they publish a new offer.
- *Synchronization:* If some process starts to execute the interaction a then all processes involved in a will eventually execute a.
- *Progress:* If an interaction a is enabled, one of its participants will eventually execute an interaction.

In this thesis, we prove the correctness by comparing the distributed execution with the centralized execution. The following correctness statement relies on the notion of execution traces. Intuitively, a trace is a sequence of events (interactions, message exchanges, processes internal actions) that occur during an execution of a specification model or a distributed implementation. For instance, the trace obtained from Figure 2.4 is $load_1, play_1$, the trace obtained from Figure 2.8 could be is (Jukebox, **OFFER**, Protocol), (Disc_1, **OFFER**, Protocol), (Protocol, **EXEC**, Jukebox), ... Note that here we considered message-passing as an atomic action, we could also have considered a trace made of emission and reception of messages instead. Furthermore, the distributed setting does not allow for a canonical trace associated to an execution since for events are not comparable by Lamport's *happened-before* relation [62], the ordering is not well defined. Given a transition system, we denote by *Exec* the set of all valid execution prefixes, that is, any sequence of action that can prefix a trace.

Using the notion of trace, the correctness of a distributed implementation is stated as follows:

- There exists an application from traces of distributed executions to sequences of centralized interactions.
- The image of a distributed trace is a trace of the centralized execution.
- A distributed trace that that is mapped to a prefix of a centralized trace can be extended so that the length of its image increases.

Note that both correctness criteria imply deadlock-freedom preservation. A deadlock is a state where no interaction or message exchange is possible. In the specification, it corresponds to a state where no interaction is enabled. In a distributed context, it corresponds to a state where all processes are waiting for some message, and there is no pending message. A system is deadlock-free if no such state can be reached from the initial state. With the first correctness criterion, deadlock-freedom preservation is ensured by the *Progress* property. For the second correctness criterion, first remark that a trace leading to a deadlock is finite and thus cannot be the proper prefix of another trace. Therefore, a distributed execution reaching a deadlock that is not present in the specification is not correct, as the third point above does not hold.

Fairness is another property considered for distributed implementation of multiparty interactions. The notion of fairness is encountered in non-deterministic systems. Intuitively, an execution is fair when any choice resolving non-determinism is fair, in the sense that there is no privileged nor prejudiced choice. This intuitive definition has two variants: strong and weak fairness [3]. Weak interaction fairness ensures that "*any continuously enabled interaction is eventually executed*", strong interaction fairness guarantees that "*any infinitely often enabled interaction is eventually executed*". Fairness can also be refer to processes or groups of interactions.

2.3 Studied Protocols

In this section, we present different protocols that ensure correct execution of multiparty interactions in a distributed setting. Notations and terminology have been adapted to fit our previous definitions.

2.3.1 Bagrodia's EM and MEM

Bagrodia proposes a set of protocols implementing multiparty interactions in a distributed context [5, 6]. In these protocols, each process sends its offer to one or several entities called *managers*. Managers then select one of the enabled interactions, execute it, and send a message to involved processes indicating that the interaction executed.

Centralized Manager

The first version of the protocol consists of a single manager, that receives offers from processes. Whenever it detects enabled interactions, it selects and executes one of them by sending an **EXEC** message to all the involved processes.

In order to ensure mutual exclusion of conflicting interactions, the manager maintains two counters for each process $\mathcal{P}_i$:

- The offer-count n_i which counts the number of offers sent by the process. This counter increments each time the manager receives an offer from the process.
- The participation-count N_i which counts the number of interactions the process participated in. This counter increments each time the manager selects an interaction involving $\mathcal{P}_i$ for execution.

Initially, both counters are set to 0. Upon receiving the first offer from $\mathcal{P}_i$, n_i is incremented and we have $n_i = N_i + 1$. This means than one offer from n_i has been received by the manager and no corresponding interaction has been executed yet. If for all processes $\{\mathcal{P}_i\}_{i \in I_a}$ participating in interaction a, the equation $n_i = N_i + 1$ holds and offers indicate that the interaction is enabled, then the interaction a can execute. Upon execution of the interaction, all counters N_i of involved processes $\{\mathcal{P}_i\}_{i \in I_a}$ are incremented and we have $n_i = N_i$ for all of them. This ensures that involved processes do not participate in a new interaction before sending an offer.

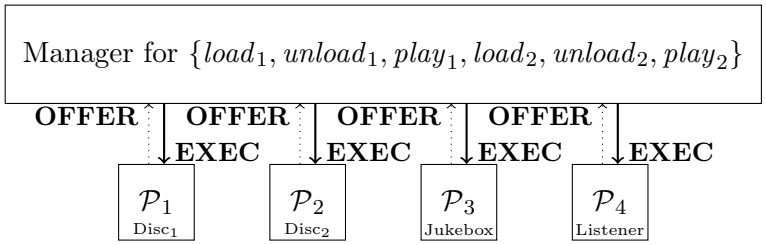

Figure 2.10: Bagrodia's solution with Centralized manager.

A global view of the solution applied to the instance from Figure 2.2 is depicted in Figure 2.10. Each process communicates exclusively with the Manager, by sending offers and waiting for **EXEC** messages. The manager performs two types of tasks:

- Receiving an offer: when an incoming offer from a participant $\mathcal{P}_i$ is received, the manager records the offer, i.e. the set of interactions in which $\mathcal{P}_i$ can take place, and increments variable n_i.
- Executing an interaction a. This task is possible only when interaction a is enabled according to up to date participants. Freshness of the offers is obtained by checking that $n_i = N_i+1$ for each participant $\mathcal{P}_i$ in a. The manager sends an **EXEC** message to each participant and increments the corresponding N_i variables.

Each one of these tasks executes atomically. When executing an interaction, the values of n_i and N_i variables are not modified by any other task.

Informally, the correctness of this protocol comes from two facts.

- Mutual exclusion of conflicting interactions is ensured by the counters. In particular, counters ensure that each offer cannot be consumed by more than one interaction.
- The protocol does not introduce deadlocks, i.e. cases where no action in the system is possible. Consider a deadlock for the system. We assume that all processes terminate their inner computation, otherwise the deadlock would also appear in the global state model. In a deadlock state, all processes have sent an offer, but no interaction is detected enabled. By replaying in the global state model the sequence of interactions that have been played by the manager (and assuming that processes are deterministic), the global behavior reaches a state where the deadlock is also present. Thus, this protocol does not introduce any deadlock.

This protocol relies on a centralized manager. Next we consider a protocol using several managers, in order to be able to execute in parallel non-conflicting interactions.

Token Ring: EM Algorithm

The token ring protocol is a decentralized variant of the centralized manager, in the sense that the latter is replaced by a set of managers $\{M_1, \ldots, M_\ell\}$. Each manager

M_k manages an arbitrary set of interactions $\mathcal{I}_k$. Whenever it is ready to interact, each process sends its offers to *all* managers that manage interactions in which it participates. Each manager M_k maintains an offer-count n_i^k for each process $\mathcal{P}$ from which it receives offers. Because of transmission delays, there might be some differences between n_i^k maintained by M_k and $n_i^{k'}$ maintained by $M_{k'}$.

In the previous version, incrementing the participation-count ensures mutual exclusion of conflicting interactions. Since these counters are stored inside the centralized manager, any modification is immediately visible. However, with multiple managers one needs to ensure that the participation-count do not change between the check for executing and the actual execution, that is, access to the participation-counts must be atomic for the manager executing the interaction.

The first solution provided by Bagrodia, named "Event Manager" (EM) embeds the participation-counts in a token. The token moves along a predefined cycle traversing all managers. When a manager detects an enabled interactions, it waits to receive the token and get the latest participation-count. If the latest values still allow the interaction, the token executes the interaction and modifies accordingly the participation-count in the token. Then it sends the token to the next manager in the cycle.

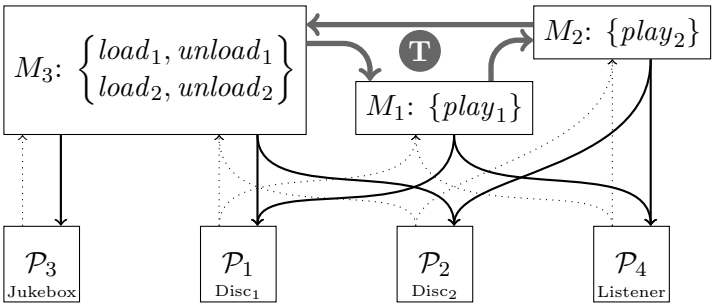

Figure 2.11: Bagrodia's EM with 3 managers: M_1 handles $play_1$, M_2 handles $play_2$ and M_3 handles all "load/unload" interactions.

Figure 2.11 represents a global view of the EM algorithm applied to the example from 2.2. In this example, we execute interactions $play_1$ within manager M_1, $play_2$ within manager M_2 and "load" and "unload" interactions within manager M_3. Other partitioning of interactions into managers are possible. A partitioning is valid as long as each interaction is executed by at least one manager. In our case, the process $\mathcal{P}_3$ (Jukebox) is involved only in the "load" and "unload" interactions, which are all handled by M_3. Therefore it does not communicate with M_1 and M_2. Other processes are involved in interactions handled by different managers, therefore they communicate with several of them. Finally, the token circulates between the three managers according the path depicted on the Figure.

Each manager shown in Figure 2.11 performs the same tasks (receives offers and executes interactions) as the centralized manager. The main difference is that an interaction executes only when its manager has the token. Upon reception of the token, the local participation-count variables are updated to the values of the token.

Dining Philosophers: MEM Algorithm

Similarly to the EM algorithm, the MEM algorithm involves several managers, each of them managing a subset of the interactions. MEM ensures atomicity of operations on participation-count in a different manner than EM. One of the drawbacks of EM is that a considerable number of token passings is needed, even in the absence of offers from the processes, to ensure that each manager does not wait "too long" before being able to schedule an interaction.

In MEM, instead of waiting for the token, each manager negotiates with other managers to get exclusive control of the participation-count of conflicting processes. Let us consider the example of Figure 2.2, with the same partitioning of interactions into managers as with the token ring protocol. Graphically, a manager corresponds to a group of nodes on the conflict graph in Figure 2.12. On this Figure, edges between two interactions

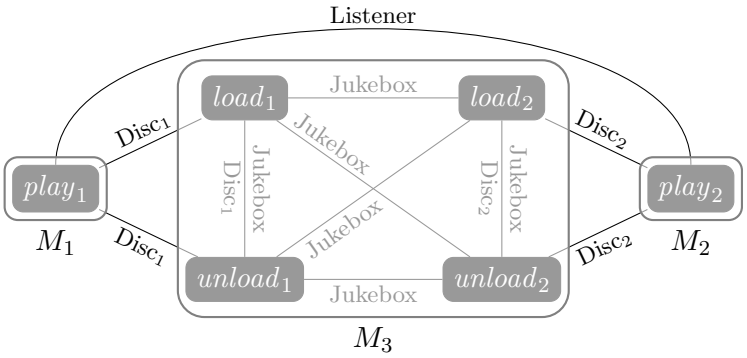

Figure 2.12: The conflict graph of example from Figure 2.2 with the managers depicted as set of nodes.

handled by the same manager do not require negotiation. Edges between interactions handled by two different managers implies a negotiation between the managers before executing one of these interactions. For instance, executing interaction $load_1$ requires that manager M_3 negotiates with manager M_1 to get access on the participation-count of Disc_1. Note that it does not require to negotiate for the participation-count of Jukebox, since only M_3 executes interactions involving Jukebox.

The problem of requesting access for each manager is equivalent to the dining philosophers problem where each philosopher has a given number of neighbors (with whom he shares a fork). At some point, a philosopher becomes hungry and tries to get the forks its neighbors. A philosopher can eat only if it has all the forks. The problem is to devise an algorithm for exchanging the forks so that no philosopher starves.

A so-called *hygienic* solution to this problem is proposed by Chandy and Misra in [32], and is reused by Bagrodia. Each fork is either clean or dirty. Initially, all forks are clean. Whenever some philosopher eats, all involved forks become dirty. The fork is cleaned when it is sent to another philosopher. At each point, one of the philosophers has the fork which his neighbor may request. Upon reception of a request, the philosopher having

the fork sends it only if it is dirty. Otherwise, it notes a request and gives the fork when he finishes eating.

Intuitively, a clean fork gives precedence to the philosopher owning it and a dirty fork gives precedence to the other philosopher. This precedence can be seen as an orientation of the conflict graph, each directed edge pointing towards the interaction/philosopher that has precedence. To avoid deadlocks, the obtained oriented graph must remain acyclic. Note that this precedence order is modified only when a philosopher uses the forks. On the graph, this operation corresponds to modifying the orientation of edges incident to the philosopher so that he becomes a source, that is a vertex with only outgoing edges. In particular, this operation cannot introduce cycles in the graph, since any introduced cycle would go through one of the modified edges and thus through a source.

A global view of the solution is presented in Figure 2.13. The interactions are grouped into managers as explained above. Note that the forks between M_1, M_2 and M_3 are deduced from the conflict graph depicted in Figure 2.12. In EM, a manager can execute an

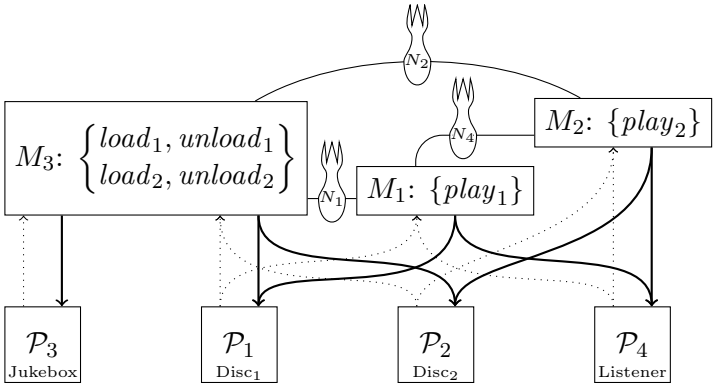

Figure 2.13: Solution obtained with MEM for the example from Figure 2.2, with the manager M_1 handling a and c, M_2 handling b and M_3 handling c.

interaction only if it has the token, which contains the participation-counts. In MEM, a manager executes only if it has all forks shared with its neighbors. Each fork contains participation-count of the participants causing the conflict between two neighbors. Unlike in centralized and token ring solutions, there is no single variable N_i holding the participation-count for a process $\mathcal{P}_i$. The actual value of the participation-count for a process is the maximum of participation-counts for that process in the forks. For instance, to execute $play_1$, manager M_1 has to get the forks corresponding to participation-counts N_1 (Disc_1) and N_4 (Listener). Indeed, $play_1$ is conflicting with both $load_1$ and $unload_1$ because of the Disc_1 process and it is conflicting with $play_2$ because of the Listener process.

Upon reception of a fork, the local values of each participation-count are updated if needed, that is if the values on the received fork are greater. Whenever an interaction takes place, the participation-counts on the forks are incremented. This coordination scheme ensures that when a manager get forks from all conflicting interactions, then at

least one of them has the latest value of the participation-count.

Counter Overflow Since the solutions proposed by Bagrodia use counters, there is a potential issue of counter overflow. In [6], Bagrodia proposes to extend the EM algorithm in order to prevent counter overflow by temporarily stopping execution to ensure that all counters have been reset. For MEM, he proposes to explicitly include a counter overflow check and reset mechanism in the original specification of the processes and interactions.

Another solution is to evaluate the lifetime of the system and select the counter size accordingly. As an example, a 64-bits counter that is incremented every 0.1 ns, that is at a frequency of 10 Ghz lasts

$$\frac{2^{64}}{10^{10}} \simeq 1.8 \times 10^9 \text{ s} \simeq 58 \text{ years.}$$

2.3.2 Kumar's Token

In [61], Kumar presented another approach for implementing multiparty interaction. The solution of Kumar does not require additional processes because it uses a protocol embedded in the original processes.

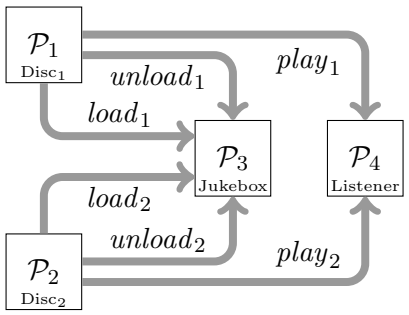

Figure 2.14: Global view of the solution proposed by Kumar.

The solution requires one token for each interaction. This token tries to progress from least to greatest process of the interaction, according to a fixed global order on the processes. On Figure 2.14, we show the path of each token, assuming that processes are ordered by their indices. Intuitively, a token can progress only if the visited process can commit to the corresponding interaction.

If the token traverses all processes in the interaction, then the interaction is safely executed. The last process is responsible for notifying all other involved processes to start the interaction. For instance, if the token for $load_1$ reaches $\mathcal{P}_3$ (Jukebox) and the latter commits to $load_1$, then $\mathcal{P}_3$ sends an **EXEC** $load_1$ message to $\mathcal{P}_1$.

Whenever a token (corresponding to an interaction) reaches a process, the process has 3 choices:

- propagate the token,

- hold the token, and wait, or
- hold the token and cancel the interaction.

Propagating the token is possible only if the process can execute the interaction corresponding to the token and has not committed to another interaction yet. Once a token is propagated, the component has committed to the corresponding interaction and waits until the latter succeeds of fails.

If the token arrives in a process that is not ready to perform the corresponding interaction, the latter is canceled. Canceling an interaction means sending a cancel message to each process that already propagated the token. After receiving a cancel message, a process is not committed to the interaction anymore.

If the token arrives in a process that has already committed to another interactions, it holds the newly received token until the other interaction succeeds or fails. In case of failure, that is if a cancel message is received, one of the held tokens is propagated. In case of success, for each token held by the process, the corresponding interaction is canceled.

Note that even when the interaction is canceled, the token remains in the process it was visiting. When the process finishes its local computation and reaches a stable state, it sends backs each token corresponding to an enabled interaction to the first process of its path.

Intuitively, the correctness of the protocol comes from the following facts:

- Mutual exclusion is ensured by the fact that each process allows only one token to traverse at a time, then waits until the corresponding interaction succeeds or fails. In particular, conflicting interactions involve at least one common process and this process ensures their mutual exclusion.
- Synchronization comes from the fact that the start message is sent by the last process on the interaction path to all the other involved processes.
- If an interaction is enabled, then either the corresponding token can travel along the whole path and the interaction executes, or the token is blocked at the first process shared with a conflicting interaction. The global order ensures that the first conflicting component is the same for both interactions. Progress is ensured by the fact that either the conflicting interaction succeeds or a cancel message allows the token to go further along the path.

2.3.3 Joung's Randomized Algorithm

In [55], Joung proposes a solution based on a randomized algorithm, that guarantees strong interaction fairness. He proposes two alternatives, one based on message-passing primitives, the other one uses shared memory. We focus only on the message-passing implementation.

Joung's solution is implemented through a protocol that is embedded in each process. This solution relies on an "attempt, wait and check" pattern.

Whenever a process reaches a stable state, the corresponding part of the protocol chooses randomly one of the locally enabled interaction. The protocol sends to each participant in the interaction a token indicating the interaction and the process identifier. The choice and the emission of the token constitutes the attempt part of the protocol.

Then, the process waits for a time Δ, during which it or another participant in the interaction may observe the establishment of the chosen interaction. An interaction is established if one process simultaneously holds one token from each participant, labeled by the interaction. In that case, the process adds a **SUCCESS** tag on each of these tokens.

After waiting Δ units of time, the process sends messages to retrieve all tokens sent during the attempt phase. Once it received back all tokens, it checks whether one of them is tagged with **SUCCESS**. In that case, it executes the corresponding interaction. Otherwise, it tries another "attempt, wait and check" cycle.

If processes have bounded computation time, then it is possible to compute a Δ that ensures progress. Otherwise, the algorithm provided in [55] includes an adaptive computation of Δ ensuring strong interaction fairness.

- Mutual exclusion is guaranteed since each process chooses one interaction, and an interaction can take place only if all processes have chosen it.
- Synchronization is guaranteed since the time elapsed between two processes starts executing the same interaction is at most Δ.
- Progress is ensured by the value of Δ, that is adaptively computed.

2.3.4 α-Core/Parrow-Sjödin Algorithm

This solution, proposed by Pérez et al. in [73] and by Parrow and Sjödin in [70], relies on two kinds of processes. There is one manager for each interaction and one participant for each process of the original model. Communication occurs only between managers and participants, as shown in Figure 2.15. This solution is used in [81] to provide a distributed implementation for the Language Of Temporal Ordering Specification (LOTOS) [53].

The main idea of this algorithm is that each manager tries to lock all participants involved in the corresponding interaction. To avoid deadlocks, locking is done according to a global order defined on the processes. The idea is similar to Kumar's Token algorithm (Subsection 2.3.2), except that the locking is done by an external manager instead of being propagated through a token.

The α-core protocol is detailed in Section 6.4.

2.4 Adding Priorities

With multiparty interactions, the next interaction is chosen non-deterministically. In some cases, the designer might want to enforce a given scheduling, without encoding it explicitly in the behavior of the processes. We provide below two motivating examples

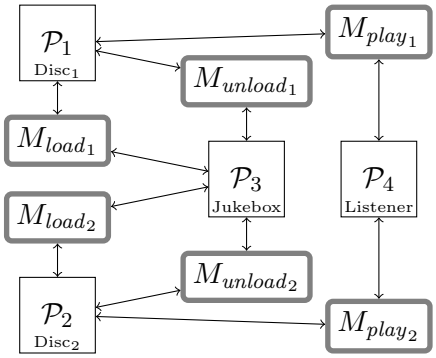

Figure 2.15: Distributed implementation obtained with α-core from the model in Figure 2.2.

where control of the scheduling is highly beneficial, and can be expressed with *priority* rules.

Avoid Deadlock States. Some systems have deadlock states that are avoided by a specific scheduling of the interactions. The dining philosopher example presents such an example. In Figure 2.16, we have 3 philosophers and 3 forks. In order to eat, each philosopher first grabs the fork to its left through a binary interaction *grabL*, then the one to its right (interaction *grabR*). Once a philosopher has the two forks, it can eat through a ternary interaction *eat*. If all philosophers grab the fork on their left, then the system reaches a deadlock since no one of the philosophers can grab the fork on its right.

Such a deadlock can be avoided by applying the following rule: "If at some global state two philosophers can grab the same fork, then only the philosopher on the left can grab it." For instance, if Philo_1 is at state 0 and Philo_2 is at state 1, then both can grab Fork_2. To avoid reaching the deadlock state only Philo_2 should take it. In other words, we give higher priority to interaction $grabR_2$ than to interaction $grabL_1$. Thus, the above rule can be stated through the three following priority rules: $grabL_1 < grabR_2$, $grabL_2 < grabR_3$ and $grabL_3 < grabR_1$. A method to automatically produce this kind of priorities is proposed in [34].

Enforce Progress. Some interactions are more rewarding than the others in terms of progress. Consider again the Jukebox example shown in Figure 2.2. In this model, interactions that make progress are the "play" interactions. In particular, we need to exclude executions where the jukebox always loads and unloads discs without playing any. These executions are avoided by enabling the "unload" interactions only if "play" interactions are not enabled. Formally, it is expressed by $unload_1 < play_1$ and $unload_2 < play_2$. Encoding this scheduling into the processes would require to know in advance the sequence of discs asked by the Listener.

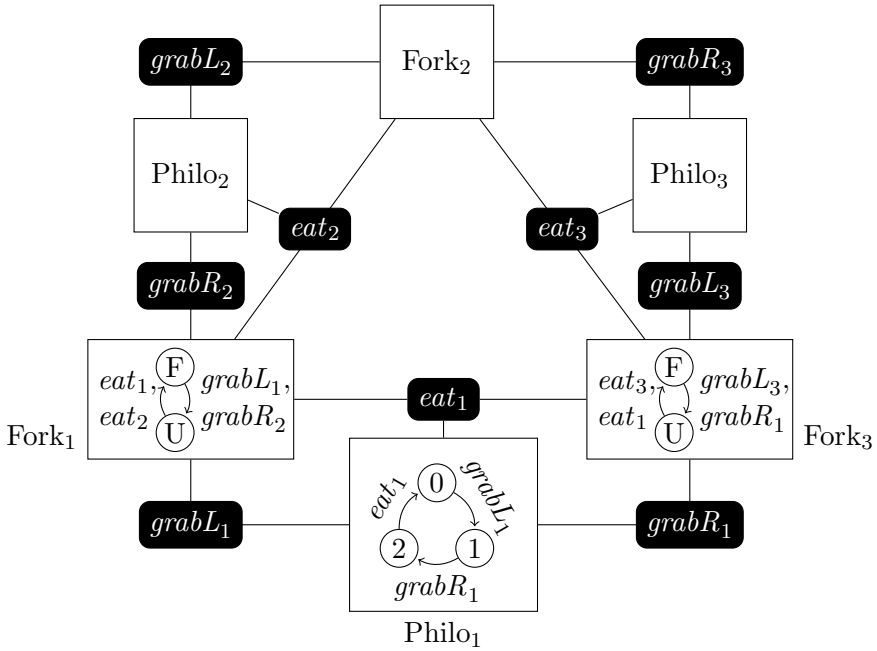

Figure 2.16: A model of the dining philosophers problem.

2.4.1 Extending Multiparty Interactions with Priorities

We now give a formal definition for the aforementioned priority rule.

Definition 2.11 (Priority)**.** Given a system of processes $\mathcal{P}_1, \ldots, \mathcal{P}_n$, where $\mathcal{P}_i = (Q_i, A_i, \rightarrow_i)$, a priority is a partial order $< \subset \mathcal{I} \times \mathcal{I}$ on the interactions of the system $\mathcal{I} = \bigcup_{i=1}^{n} A_i$.

The semantic of a system of processes communicating through prioritized multiparty interactions is based on the semantic of the system without priorities.

Definition 2.12 (Global behavior of processes communicating through prioritized multiparty interactions)**.** The behavior of a system of processes $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ equipped with a Priority $<$ is a LTS $(Q, \mathcal{I}, \rightarrow_<)$, where $(Q, \mathcal{I}, \rightarrow)$ is the unprioritized behavior and $\rightarrow_<$ is the least set of transitions satisfying the rule:

$$\frac{a \in \mathcal{I} \qquad q \xrightarrow{a} q' \qquad \forall b \; a < b, \; q \overset{b}{\not\longrightarrow}}{q \xrightarrow{a}_< q'}$$

Note that rule only excludes transitions for which a higher priority interaction is possible. In particular, it does not introduce deadlocks since an interaction is forbidden only if at least one other interaction (with higher priority) is possible.

To our knowledge, there are no protocols implementing prioritized multiparty interactions in a distributed context. This is mainly because in the literature multiparty

interactions are specified in a distributed context and not as the distributed implementation of a global state model. Indeed, in a distributed context it is unintuitive to express the fact that an interaction is not possible, and what should be a correct execution of a prioritized system. On the other hand, our approach allows us to state the correctness of the distributed implementation by comparing it to the global state model.

Note that [49] provides a solution for building a distributed implementation of BIP with priorities. However, the proposed solution does not directly handle models with *confusion*, that is, where conflicts due to interactions and conflicts due to priorities are mixed.

2.5 Other Extensions and other Distributed Models

In this section, we present related works regarding multiparty interaction and models for generating distributed implementation in general. The first subsection discusses a more dynamic version of multiparty interaction, that we do not consider. Finally, Subsection 2.5.2 presents some other frameworks that either rely on multiparty interactions or allow distributed code generation.

2.5.1 Multiparty Interactions Extension

In this chapter, we define multiparty interactions, and present in Section 2.3 some protocols implementing them. Our definition assumes that each interaction has a statically fixed set of components. Other definitions allow a more open behavior, to encompass dynamic creation and deletion of processes. An extension of BIP [29] allows specifying such constructs.

In [57], an interaction is specified as a set of roles. The interaction can take place whenever each role is fulfilled by a distinct process. In that context, a single interaction, that is a set of roles, defines several combinations of actual processes. Detecting the sets of processes that can interact from a given state is therefore more difficult than in the static case, where each interaction involves a fixed set of processes. The paper proposes an algorithm where each process can coordinate the execution of an interaction. Another solution, proposed in [72], separates detection of enabled interactions from selection of the next interaction to execute. The focus is on the detection of enabled interactions. The solution supports any selection algorithm provided it meets some criteria, allowing, for instance, the use of a selection algorithm focused on fairness.

2.5.2 Other Frameworks

German's Framework

In [46], German presents a framework providing multiparty interactions with priorities as primitives. The processes are described using a notation similar to the Calculus of Communicating Systems (CCS). Interactions are specified as composite action labels made of conjunctions and negations of actions. For instance, the interaction $unload_1$ of

the example from Figure 4.6 could be encoded as the action label $un_1 \wedge d_1 u \wedge \neg(rec_1 \wedge pl_1)$, assuming that BIP ports are mapped to simple actions. Encoding an interaction requires an additional process, that contains a single rule with the corresponding action label.

The idea is to provide a language suitable for specifying distributed systems, with a high-level description that can be executed for rapid prototyping. This framework was used to model and verify a telephone switching application. To our knowledge, there is no distributed implementation for this framework, although it was apparently one of the goals in [46].

Reo

Reo [4] is a framework where components communicates using a basic set of dataflow connectors that are combined to form a complex connector. At each round, each component enables a set of input and output ports. In Reo, components are black boxes, only their interface is known at each state. A Reo connector defines a set of allowed dataflow interactions for each configuration of the enabled ports. A round consists of executing such an interaction, which transfers data. Furthermore, Reo basic connectors include FIFO_1 connectors that can store one data item, allowing the composed connector to save some data between two rounds. The FIFO connectors introduce a control state in the composed connector, which differs from BIP where interactions have no memory.

The Dreams framework [74, 75] provides a distributed implementation for Reo. Each basic connector is implemented as an actor. A round synchronizes the actors through a consensus algorithm, in order to choose the next interaction. In order to achieve a more decentralized behavior, a Globally Asynchronous Locally Synchronous (GALS) architecture is obtained by cutting the complex connector into synchronous regions. Two regions can be separated only if their only connections consist of FIFO connectors.

I/O Automata

I/O automata [63] were introduced to formally model distributed systems. In this framework, each process is represented by an automaton whose transitions are labeled by actions, similarly to processes from this chapter. As for systems of processes, each interaction is represented through a common label that is used in several processes to denote synchronization. However, I/O automata clearly distinguish between input (uncontrollable) actions and output (controllable) actions. Given an interaction label, there is exactly one process for which this label is an output action, in other processes it can appear only as an input action. Furthermore, from every state of an automaton, all its input actions are required to be enabled. With these restrictions, an interaction is completely controlled by the process for which it is an output action.

I/O automata interactions are similar to Send/Receive interaction as they are completely controlled by the sender and the receiver should not block the sender. In particular, the fact that an interaction is enabled or not is local to the process that controls the corresponding output action. In that sense, there is no conflict between interactions. However, if two interactions a and b are scheduled simultaneously by two separate pro-

cesses, the order should be consistent among all common participants in a and b. The solution proposed in the first sketch of a distributed implementation [44] is to require that each automaton reaches the same state for both orderings. In a later solution [45, 82], this problem is solved by adding a handshake protocol.

Synchronizers

In [42, 39], *Synchronizers* are used to filter incoming messages for a set of actors. A message is delivered if no synchronizer prevents it through a *disable* construct and the message matches an enabled pattern. Such patterns include atomic synchronization of a set of messages, that requires all involved messages to be pending before granting their transmission. According to [39], synchronizers are implemented through dispatchers located on the target actors, that is the actors for which incoming messages are filtered. Upon reception of an incoming message, the dispatcher is responsible for checking whether the message is allowed for transmission according to the synchronizers. In case of atomic synchronization, this requires a protocol similar to the one for multiparty interactions. The actors communicate through asynchronous message-passing, which makes it difficult to exploit the synchronization of messages for verification purposes. This framework is mainly concerned with providing practical constructs for programming with actors.

Behavioral Programming

Another model of programming interactions between processes is described in [65]. In that model, at each global state, each process provides three sets of actions: requested actions, watched actions and blocked actions. A (centralized) scheduler selects an action that is requested by at least a process and not blocked by any process. The selected action is executed by all processes that requested it and all processes that watched it. The system reaches the next global state by executing the selected action.

This model differs from multiparty interactions as the set of participants in the common action is not fixed, but depends on the state. Decentralizing the scheduler while preserving centralized semantics requires to solve problems similar to Interaction and Condition conflicts resolution. In particular, scheduling an action based on a partial set of offers requires to ensure that this action will not be blocked by a subsequent offer.

Guesstimate

A framework for programming collaborative distributed model is proposed in [76]. This framework allows several processes to work on a common set of objects. Each process works with a local copy of the common objects and execute locally a series of actions on them. A periodic synchronization gathers the sequences of actions performed by all processes and interleaves them within a global sequence. Each process then replays these actions from the last synchronization point. In case of conflicts between two actions, that is, if executing one disables the other one, the last one in the global sequence is simply discarded.

Such frameworks are suitable for systems where processes are always computing, and may alter their states when taking the global execution sequence into account. Only the synchronized part of the history correspond to a correct execution of the system. The local actions executed in each process cannot, in general, be combined to obtain a valid execution of the system.

3 Knowledge

In this thesis, knowledge refers to the knowledge of an agent, that is a part of a global system, about the global system. Since the agent has only a limited view of the global system, it cannot directly assess the truthfulness of properties about the global system. However, assuming an agent can reason, it might infer more facts than those it observed. Knowledge allows reasoning about what each agent knows about the global system. In order to illustrate this kind of reasoning, let us consider the muddy children puzzle, which is also known as the "cheating husband" or "unfaithful wives" puzzle. The following text is quoted from [7].

> Imagine n children playing together. The mother of these children has told them that if they get dirty there will be severe consequences. So, of course, each child wants to keep clean, but each would love to see the others get dirty. Now it happens during their play that some of the children, say k of them, get mud on their foreheads. Each can see the mud on others but not on his own forehead. So, of course, no one says a thing. Along comes the father, who says, "At least one of you has mud on your head," thus expressing a fact known to each of them before he spoke (if $k > 1$). The father then asks the following question, over and over: "Can any of you prove you have mud on your head?" Assuming that all the children are perceptive, intelligent, truthful, and that they answer simultaneously, what will happen?

Here, the agents are the children. They can observe the forehead of other children, as well as the questions asked by the father and their answers. At the beginning, the knowledge of each child is the number q of other children that have mud on their forehead. A child seeing q other muddy children knows that either $k = q$ or $k = q + 1$, depending whether it is itself muddy or not. After the father's first announcement, it is common knowledge that there is at least one muddy child. Furthermore, the announce being public, each child knows that each child knows that there is at least one muddy child.

The progress of the situation is the following: the first $k - 1$ questions of the father will be answered "no" by all the children and the kth question will be answered "yes" only by the muddy children. This property can be proved by induction on the number k of muddy children.

- If $k = 1$, the only muddy child sees no other muddy children, so it can deduce that it is muddy and answer "yes" to the first question.
- If $k = 2$, each muddy children sees that there another one child who is muddy. Thus, it cannot rule out the possibility that there is only one muddy child. Consequently, it will answer "no" to the first question of the father. Since every child is

intelligent, it can do the same reasoning about the case where $k = 1$. This case is ruled out as the other muddy child also answered "no", to the first question of the father. Thus, both of the muddy children will answer "yes" to the second question of the father.

- If there are $k + 1$ muddy children, they cannot rule out the possibility that there is only k muddy children. This possibility is actually ruled out when the other muddy children answer "no" to the kth question of the father, since they would have answered "yes" according to the induction hypothesis. Therefore, the $k + 1$ muddy children will answer "yes" to the next question of the father.

This puzzle and its variants have been studied and formalized in [50, 51]. In [50], a similar problem is modeled as a knowledge-based protocol for a distributed system. The framework provided by the concept of knowledge can be applied to distributed systems, by considering each process as an agent. Each process sees a part of the system and may infer a part that it does not see. In [51] Halpern and Moses define different levels of knowledge among a group of processes.

The weakest level, distributed knowledge, is obtained by combining the knowledge of all processes in the group. With this kind of knowledge, a fact can be known by the group even if no process knows it. For instance, at the beginning of the muddy children problem, distributed knowledge allows knowing which children are muddy, although each children does not know whether it is muddy. Actually, the distributed knowledge obtained by combining the knowledge of any two children is sufficient to know the global state. Indeed, by taking the knowledge of one child, the only missing information is whether this child is muddy, which is completed by adding the knowledge of a second child. In practice, building this kind of knowledge implies a synchronization for processes to share their observations.

Higher levels include facts that are known by at least one member of the group, facts that are known by all members of the group and then facts that are known to be known by all members of the group. The highest level of knowledge among a group of processes is called common knowledge. It correspond to facts that are publicly announced, such as "At least one of you has mud on your head." in the muddy children puzzle.

Knowledge assumes that a set of universes is defined. In the case of the muddy children puzzle, a universe is defined by specifying which children have mud on their head. The global system is always in one single coherent universe. Each agent knows the set of possible universes and the observed facts in the current one. In other words, for each agent, the knowledge corresponds to the set of universes that are consistent with its observations. In particular, this introduces, for each agent, an equivalence relation between universes, that is called *undistinguishability* relation. Two universes are said to be undistinguishable for an agent if its observations are coherent for both universes.

An important parameter to define the knowledge for a given process is to define which observed facts determine the knowledge. For instance one can consider only the local state of the agent. A stronger knowledge is obtained by assuming that each agent remembers the history of all interactions it has participated in. This is called knowledge

with perfect recall [84]. Note that for the muddy children, one has to remember how many times the father asked the question, that is, to remember history.

In this chapter, we present two concrete constructions of knowledge. Firstly, we focus on distributed knowledge obtained by considering only the current state of a group of processes. Secondly, we present how to build history with perfect recall for one process. Finally, we discuss how knowledge has been used to actually implement distributed systems.

3.1 Distributed Knowledge based on Local State

We assume a system of processes $\mathcal{P}_1, \ldots, \mathcal{P}_n$ as described in Section 2.1. In that case, the set of possible universes is the set of *reachable* global states. Consider the global behavior $(Q, \mathcal{I}, \rightarrow)$ of the system. We say that a global state q is *reachable* if there exists a sequence of interactions $a_1, \ldots, a_n$ such that $q_0 \xrightarrow{a_1} q_1 \rightarrow \cdots \xrightarrow{a_n} q$, where q_0 is the initial state. We denote by $\mathcal{R}$ the set of reachable states of the system. For our Jukebox example depicted in Figure 2.2, the reachable states are the states of the global behavior depicted in Figure 2.3.

In this section, we focus on knowledge deduced by observing the local state of a subset of processes $\mathcal{L}$. This corresponds to the distributed knowledge of [51]. Given a subset $\mathcal{L} = \{\mathcal{P}_{i_1}, \ldots, \mathcal{P}_{i_k}\}$ of processes, we define the projection $q|_{\mathcal{L}}$ of a global state q on $\mathcal{L}$ as the local states of processes in $\mathcal{L}$ when the system is at global state q. Formally, we have:

$$.|_{\mathcal{L}} : \begin{array}{ccc} Q_1 \times \ldots \times Q_n & \longrightarrow & Q_{i_1} \times \ldots \times Q_{i_k} \\ (q_1, \ldots, q_n) & \mapsto & (q_{i_1}, \ldots, q_{i_k}) \end{array}$$

Two global states that have the same projected state on $\mathcal{L}$ are said to be *undistinguishable*. A global state is fully determined by its projection on $\mathcal{L}$ and its projection on the complementary of $\mathcal{L}$, that is $\mathcal{L} \setminus \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$.

Definition 3.1 (Undistinguishability relation)**.** Two global states q and q' of the system are said to be *undistinguishable* by a set of processes $\mathcal{L}$, denoted $q \sim_{\mathcal{L}} q'$, iff $q|_{\mathcal{L}} = q'|_{\mathcal{L}}$.

Given a set of processes $\mathcal{L}$, the undistinguishability relation for $\mathcal{L}$ is an equivalence relation on the global states of the system. A class of this equivalence relation corresponds to the set of global states that extend a given local state of $\mathcal{L}$. By combining the undistinguishability relation $\sim_{\mathcal{L}}$ and the reachable states, we obtain the set of reachable states that are undistinguishable for a given local state of $\mathcal{L}$. This corresponds to the distributed knowledge of the processes in $\mathcal{L}$ at that local state.

Example 3.2. Consider again the example from Figure 2.2. As said above, the reachable states are the states of the LTS in Figure 2.3. In Figure 3.1, we assume that only the Jukebox process is observed, that is $\mathcal{L} = \{\text{Jukebox}\}$. Each square corresponds to a possible global state, characterized by two coordinates: a local state (the global state projected on $\mathcal{L}$) and a possible complement of the local state. If the local state is 0, the corresponding global state is in the column corresponding to 0. In that case, considering

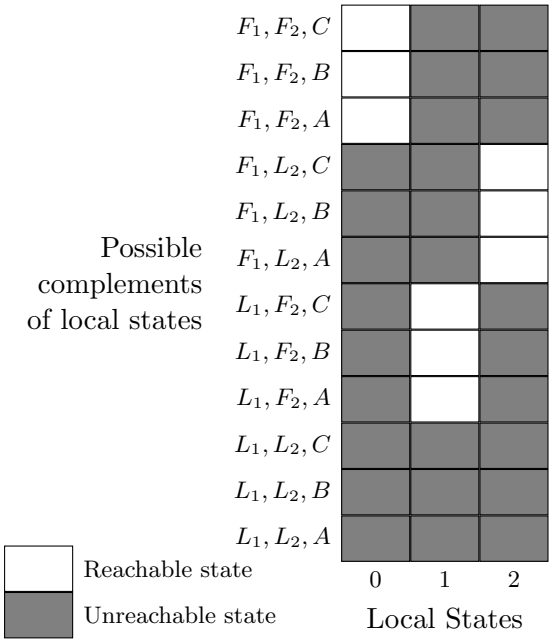

Figure 3.1: Global states of the example from Figure 2.2, decomposed by observing the Jukebox process.

only the reachable states (white squares) 3 candidate global states remain. In each one of them, the state of the discs is F_1, F_2, which is actually *known* by observing only the jukebox and not the discs. In Figure 3.2, we assume that the Jukebox and Reader processes are observed. In that case, for each local state, there is only one indistinguishable reachable state. That is, observing only these two processes is sufficient to know the complete global state.

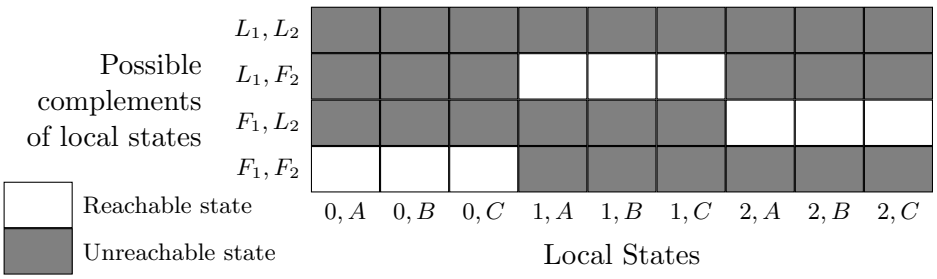

Figure 3.2: Global states of the example from Figure 2.2, decomposed by observing Jukebox and Reader processes.

The facts that we can express about the system are encoded using the local states of the processes. If $\mathcal{P}_i = (Q_i, A_i, \rightarrow_i)$, we define for each local state $\ell \in \bigcup_{i=1}^{n} Q_i$ of a process a predicate at_ℓ which is true whenever ℓ is active. This predicate is defined over

the set of global states, that is $Q = Q_1 \times \ldots \times Q_n$. In this thesis, we denote by $\mathcal{B}$ the boolean domain, i.e. $\{True, False\}$.

$$at_\ell : \begin{array}{ccc} Q & \longrightarrow & \mathcal{B} \\ (q_1, \ldots, q_n) & \mapsto & \exists i \; q_i = \ell \end{array}$$

Since we assume that local states are pairwise disjoint, there is at most one i such that $\ell \in Q_i$. In the sequel, we denote by ϕ a formula obtained using $\neg, \wedge, \vee$ and at_ℓ predicates, i.e. $\phi : Q \to \mathcal{B}$.

Intuitively, a set of processes $\mathcal{L}$ knows a formula ϕ if ϕ holds in all reachable states indistinguishable from the current local state of $\mathcal{L}$. In other words, a set of processes knows a formula when the current local state ensures that no reachable state falsifies the formula. Note that an over approximation $\tilde{\mathcal{R}}$ of the reachable states $\mathcal{R}$ is sufficient to define knowledge. Indeed, ensuring that no state in $\tilde{\mathcal{R}}$ indistinguishable from the local state falsifies the formula ϕ also ensures that no reachable state indistinguishable from the local state falsifies ϕ. We represent this by a new predicate, that is defined over the global states.

Definition 3.3 (Knowledge predicate)**.** Given a system of processes $\mathcal{P}_1, \ldots, \mathcal{P}_n$, such that Q is the set of its global state, $\tilde{\mathcal{R}} \supseteq \mathcal{R}$ is an over approximation of the reachable states, and a subset $\mathcal{L} \subseteq \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, we define the knowledge predicate $K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\phi$ as follows:

$$K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\phi : \begin{array}{ccc} \tilde{\mathcal{R}} & \longrightarrow & \mathcal{B} \\ q & \mapsto & \forall q' \in \tilde{\mathcal{R}} \;\; q' \sim_{\mathcal{L}} q \implies \phi(q') \end{array}$$

By definition, if q is a reachable global state, we have $K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\phi(q) \implies \phi(q)$. In other words, the distributed knowledge is truthful. Another consequence of this definition is that $K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\phi$ depends only on the state of $\mathcal{L}$. Indeed, the set $[q] = \{q' \in \tilde{\mathcal{R}} \mid q' \sim_{\mathcal{L}} q\}$ depends only on the local states $(q_{i_1}, \ldots, q_{i_k})$ of the processes in $\mathcal{L}$ at global state q, and $K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\phi$ holds if ϕ holds in all states in $[q]$.

On our example, for the case where only the Jukebox process is observed ($\mathcal{L} = \{\text{Jukebox}\}$), the predicate $K_{\mathcal{L}}^{\mathcal{R}} at_{F_1}$ holds at global state $q = (F_1, F_2, 0, A)$. Indeed, at_{F_1} holds in all reachable states that are undistinguishable from q by $\mathcal{L}$, namely $(F_1, F_2, 0, B)$ and $(F_1, F_2, 0, C)$. In that case, observing that Jukebox is at state 0 is sufficient to ensure that at_{F_1} holds.

The predicate $K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\phi$ tells whether observing the set $\mathcal{L}$ of processes is sufficient or not to ensure truthfulness of ϕ at a given global state. Whenever this predicate is false, there is no guarantee that ϕ is also false. For instance, consider the case where only the Jukebox process is observed. If the Jukebox process is at state 0, the predicate $K_{\mathcal{L}}^{\mathcal{R}} at_A$ is false, because the state of the Reader process can be different than A. In other words, $K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\phi$ gives a lower bound of ϕ. We can obtain an upper bound by considering the complementary of $K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\neg\phi$, that is the states where the local observation does not ensure that ϕ is false. This is stated in Proposition 3.4.

Proposition 3.4. $\forall q \in \tilde{\mathcal{R}} \;\; K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\phi(q) \implies \phi(q) \implies \neg K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\neg\phi(q)$.

Proof. According to the definition 3.3, $K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\phi \implies \phi$ holds for all states in $\tilde{R}$. The second implication is the contraposition of $K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\neg\phi \implies \neg\phi$, that holds for all states in $\tilde{\mathcal{R}}$. □

Finally, when observing more processes, the knowledge predicate is more precise. For instance, between the situation in Figure 3.1 and the situation in Figure 3.2, the Reader process is also observed, completing information about the global state. Note that this was a "clever" choice, as adding Disc_1 or Disc_2 would not have given any additional information.

Proposition 3.5 (Monotonicity)**.** *The predicate $K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\phi$ is monotonic with respect to $\mathcal{L}$, i.e. $\mathcal{L} \subseteq \mathcal{L}'$ implies $K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\phi \implies K_{\mathcal{L}'}^{\tilde{\mathcal{R}}}\phi$.*

Proof. First, remark that if $\mathcal{L} \subseteq \mathcal{L}'$, then $\{q' \in \tilde{\mathcal{R}} \mid q' \sim_{\mathcal{L}'} q\} \subseteq \{q' \in \tilde{\mathcal{R}} \mid q' \sim_{\mathcal{L}} q\}$, since states undistinguishable for $\mathcal{L}$ might differ on $\mathcal{L}' \setminus \mathcal{L}$. Then, $K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\phi(q)$ means that $\forall q' \in \tilde{\mathcal{R}}\ \ q' \sim_{\mathcal{L}} q \implies \phi(q')$ and by the above remark $\forall q' \in \tilde{\mathcal{R}}\ \ q' \sim_{\mathcal{L}'} q \implies \phi(q')$, that is, $K_{\mathcal{L}'}^{\tilde{\mathcal{R}}}\phi(q)$. □

Notice that observing the whole system, i.e. taking $\mathcal{L} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, fully characterizes any predicate ϕ. In that case indeed the relation $\sim_{\mathcal{L}}$ becomes the equality on Q. Therefore, for every global state q, $K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\phi(q)$ is equal to $\phi(q)$.

3.1.1 Representation and Computation

In practice, the set $\mathcal{R}$ of reachable states or its over approximation $\tilde{\mathcal{R}}$ is represented through a predicate $\mathfrak{I}$ defined of the global states. More precisely, if $q = (q_1, \ldots, q_n)$, $\mathfrak{I}(q)$ evaluates to true only if $q \in \tilde{\mathcal{R}}$.

Given a subset $\mathcal{L} = \{\mathcal{P}_{i_1}, \ldots, \mathcal{P}_{i_k}\}$ of processes whose state is known, $\mathfrak{I}$ characterizes the possible states of the remaining processes $\overline{\mathcal{L}} = \{\mathcal{P}_{j_1}, \ldots, \mathcal{P}_{j_{n-k}}\}$. Assuming that processes in $\mathcal{L}$ are at state $(q_{i_1}, \ldots, q_{i_k})$, the possible states for remaining processes $\overline{\mathcal{L}}$ is $\{(q_{j_1}, \ldots, q_{j_{n-k}}) \mid \mathfrak{I}(q_1, \ldots, q_n)\}$. The global states obtained by combining the known states of $\mathcal{L}$ and a possible state of its complementary $\overline{\mathcal{L}}$ constitute the states indistinguishable from $\mathcal{L}$ still validating $\mathfrak{I}$.

Given a predicate ϕ, the knowledge predicate $K_{\mathcal{L}}^{\tilde{\mathcal{R}}}\phi$ is encoded as

$$(q_{i_1}, \ldots, q_{i_k}) \mapsto \forall q_{j_1} \ldots q_{j_{n-k}}\ \mathfrak{I}(q_1, \ldots, q_n) \implies \phi(q_1, \ldots, q_n).$$

The input range can be extended to the whole set of global states Q by simply ignoring the states of processes in $\overline{\mathcal{L}}$. This formula gives valid results only if the input $(q_{i_1}, \ldots, q_{i_k})$ is the projection of a state in $\tilde{\mathcal{R}}$ on $\mathcal{L}$.

Computing Invariants

We detail below two kinds of invariants that yield suitable $\mathfrak{I}$ predicates. The approximation provided by these invariants is obtained with a much lower computational complexity than the exact set of reachable states. To describe how interaction invariants are computed, we use Petri net formalism, relying on the equivalence described in Subsection 2.1.1.

Boolean Invariants correspond to traps or siphons in Petri nets [69, 36]. A trap is a set of places F such that $F^\bullet \subseteq {}^\bullet F$, where $F^\bullet$ (resp. ${}^\bullet F$)is the set of transitions that have an input place (resp. an output place) in F. Any transition that might remove tokens from F, that is a transition in $F^\bullet$, is also a transition in ${}^\bullet F$, that is a transition that adds at least one token in F. Therefore, any trap initially containing a token will contain at least one token in each state and provides an invariant. In the Petri net depicted in Figure 2.5, the set $\{p_1, p_2, p_3\}$ is a trap. Indeed, if p_1 initially contains a token, any marking reached during further execution contains at least one token in the places $\{p_1, p_2, p_3\}$. The invariant corresponding to this trap is $at_{p_1} \vee at_{p_2} \vee at_{p_3}$.

Boolean Behavioral Constraints (BBC) [15] encode the condition $F^\bullet \subseteq {}^\bullet F$. These constraints involve boolean variables inF_ℓ, where ℓ is a place. Any boolean assignment that satisfies the BBC provides a trap, by taking the places ℓ for which inF_ℓ is true. The BBC state that if a place ℓ is in the trap, then for each global transition τ that takes a token from ℓ, there is at least one output control location of τ that is also in F. Using Petri net notations, the BBC obtained with ℓ is

$$inF_\ell \implies \bigwedge_{\tau \in \ell^\bullet} \bigvee_{\ell' \in \tau^\bullet} inF_{\ell'}$$

For the example of Figure 2.5, the BBC would be

$$\begin{array}{rcl} inF_{p_1} & \implies & inF_{p_2} \vee inF_{p_4} \\ inF_{p_2} & \implies & inF_{p_3} \\ inF_{p_3} & \implies & true \\ inF_{p_4} & \implies & inF_{p_5} \\ inF_{p_5} & \implies & true \end{array}$$

Note that $\{p_1, p_2, p_3\}$ actually corresponds to a solution of this system. The Boolean invariant $\mathfrak{I}$ obtained is equivalent to the conjunction of all solutions of the BBC that correspond to initially non-empty traps. Efficient methods to compute incrementally this invariant are provided in [15].

Linear Invariants are also originated from Petri net theory. A linear constraint is denoted as a linear combination of at_ℓ predicates that is equal to a constant. For instance, a linear constraint of the Petri net from Figure 2.5 is $2at_{p_1} + at_{p_2} + at_{p_3} + at_{p_4} + at_{p_5} = 2$. Here, to evaluate whether the constraint is satisfied at a given state, we assume that at_ℓ takes the value 1 if ℓ is active and 0 otherwise. A linear invariant is a conjunction of linear constraints.

Methods for computing linear invariants are described in [60]. These methods are based on linear algebra, and more precisely on the Place-Transition matrix. The latter is a matrix M where each line corresponds to a transition and each column corresponds to a place. The element M_{ij} at line i and column j contains the effect of the transition i on place j. For 1-Safe Petri nets, transition i either adds a token in place j ($M_{ij} = +1$), removes a token from j ($M_{ij} = -1$), removes then adds a token ($M_{ij} = 0$), or does not

involve place j ($M_{ij} = 0$). We provide below the Place-Transition matrix for the Petri net in Figure 2.5.

$$\begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \end{array} \begin{array}{c} \begin{array}{ccccc} p_1 & p_2 & p_3 & p_4 & p_5 \end{array} \\ \begin{bmatrix} -1 & 1 & 0 & 1 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix} \end{array}$$

The Place-Transition matrix can be used to compute the state reached after executing a sequence σ of interactions. To perform this computation, Petri nets markings are represented as column vectors, whose ith element indicates the number of tokens in the place i. For 1-Safe Petri nets, this number is either 0 or 1. For instance, the marking for the Petri net of the left of Figure 2.5 would be:

$$m = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{array}{l} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \end{array}$$

Similarly, we introduce a firing vector whose ith element indicates how many times transition i is executed during the sequence. The order is not specified and it is not checked that the sequence is a valid one. The firing vector for the sequence $\sigma = t_1 t_2$ is

$$V(\sigma) = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \begin{array}{l} t_1 \\ t_2 \\ t_3 \end{array}$$

Using these notations, computing the marking m reached after executing the sequence σ from initial marking m_0 is a simple matrix computation:

$$m = m_0 + M^T V(\sigma)$$

A linear constraint is obtained from a vector I that cancels M, that is such that $MI = 0$. By multiplying the previous equation by I^T, we obtain:

$$I^T m = I^T m_0 + (MI)^T V(\sigma) = I^T m_0$$

The value K of the scalar product $I^T m$ does not depend on σ, thus holds for any marking reachable from m_0. At global state q, the marking is $m = [at_{\ell_1}(q) \ \ldots \ at_{\ell_n}(q)]^T$. By denoting k_i the ith element of I, the previous equation is rewritten

$$\sum_{i=1}^{n} k_i at_{\ell_i}(q) = K$$

The above equation holds for each reachable state q and thus is an invariant.

It is sufficient to find a basis $(I_1, \ldots, I_k)$ of the kernel of M to be able to generate all such constraints. Given such a basis, the obtained predicate $\mathfrak{I}$ is $\bigwedge_{i=1}^{k} I_i^T m = I_i^T m_0$.

3.2 Knowledge with Perfect Recall

In this Section, we focus on the knowledge for only one process $\mathcal{P}_i$. However, instead of only considering its own state to compute the knowledge, we assume that the process remembers the history of all interactions in which it participated. This kind of knowledge has been studied in [84].

The process $\mathcal{P}_i = (Q_i, A_i, \rightarrow_i)$ observes and remembers only the interactions in which it is involved, that is the interactions in A_i. We denote by σ a prefix of a global execution trace. We denote by *Exec* the set of all prefixes of traces of valid executions. Such a prefix is a word of $\mathcal{I}^*$. We denote by σ_j the jth interaction of the word and by ϵ the empty string. Furthermore, if σ is a valid global execution prefix, then there is a sequence of transitions from the initial state $q_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_k} q$, reaching a state q. We denote by $q_0.\sigma$ the state reached after executing σ from the initial state q_0. We define the projection of a global execution prefix on A_i as follows:

$$\begin{array}{rccl} . |_{\mathcal{P}_i} : & \mathcal{I}^* & \longrightarrow & A_i^* \\ & \sigma_1\sigma_2\ldots\sigma_k & \mapsto & \sigma'_1\sigma'_2\ldots\sigma'_k \text{ where } \sigma'_j = \left\{ \begin{array}{ll} \sigma_j & \text{if } \sigma_j \in A_i \\ \epsilon & \text{otherwise} \end{array} \right. \end{array}$$

As for the global states, we say that two global execution prefixes are undistinguishable by $\mathcal{P}_i$ if they have the same projection on A_i.

Definition 3.6 (Undistinguishability relation (for traces))**.** Two global execution prefixes σ and σ' of the system are said to be *undistinguishable* by the process $\mathcal{P}_i$, denoted $\sigma \sim_i \sigma'$, iff $\sigma|_{\mathcal{P}_i} = \sigma'|_{\mathcal{P}_i}$.

The undistinguishability relation is an equivalence relation. The equivalence classes of this relation correspond to set of execution prefixes that cannot be distinguished by $\mathcal{P}_i$.

Example 3.7. For our example, the two execution prefixes $load_1, play_1$, $unload_1$ and $load_1, unload_1$ are undistinguishable by the Jukebox process, since it is not involved in the interaction $play_1$.

As for the previous case, we define a knowledge predicate in the case of perfect recall.

Definition 3.8 (Knowledge with perfect recall)**.** Given a process $\mathcal{P}_i$, a state predicate ϕ, we define the knowledge with perfect recall $K^{PR}_{\mathcal{P}_i}\phi$ as follows:

$$\begin{array}{rccl} K^{PR}_{\mathcal{P}_i}\phi : & Exec & \longrightarrow & \mathcal{B} \\ & \sigma & \mapsto & \forall \sigma' \in Exec \;\; \sigma' \sim_{\mathcal{P}_i} \sigma \implies \phi(q_0.\sigma') \end{array}$$

The main difference with the knowledge based on local state is that this new knowledge predicate depends on the execution sequence leading to the current state. As for the knowledge based on local state, the knowledge with perfect recall is truthful, i.e. if $K^{PR}_{\mathcal{P}_i}\phi(\sigma)$ holds, then by definition $\phi(q_0.\sigma)$ also holds. The Proposition 3.4 can be adapted for the knowledge with perfect recall.

Proposition 3.9. $\forall \sigma \in Exec \;\; K^{PR}_{\mathcal{P}_i}\phi(\sigma) \implies \phi(q_0.\sigma) \implies \neg K^{PR}_{\mathcal{P}_i}\neg\phi(\sigma)$.

Proof. According to the definition 3.8, $K^{PR}_{\mathcal{P}_i}\phi(\sigma) \implies \phi(q_0.\sigma)$ holds for all execution prefixes in *Exec*. The second implication is the contraposition of $K^{PR}_{\mathcal{P}_i}\neg\phi(\sigma) \implies \neg\phi(q_0.\sigma)$, that holds for all execution prefixes in *Exec*. □

Since we assume here that only one process is observed, we cannot observe more processes to precise the above approximation. However, the knowledge with perfect recall of the process $\mathcal{P}_i$ after executing a sequence σ is more precise than the knowledge obtained by observing the same process at the state $q_0.\sigma$, as shown in the next proposition.

Proposition 3.10. *Given a process $\mathcal{P}_i$, a global execution prefix $\sigma \in Exec$, and a state predicate ϕ, we have $K^{\mathcal{R}}_{\{\mathcal{P}_i\}}\phi\ (q_0.\sigma) \implies K^{PR}_{\mathcal{P}_i}\phi\ (\sigma)$.*

Proof. Consider the set $G = \{q \in Q \mid \exists \sigma' \sim_{\mathcal{P}_i} \sigma \;\; q_0.\sigma' = q\}$ of global states that are reached after executing any sequence undistinguishable from σ by $\mathcal{P}_i$. By definition, all states in G are reachable, i.e. $G \subset \mathcal{R}$. Furthermore, in all states of G, the process $\mathcal{P}_i$ has the same state since it played the same sequence $\sigma|_{\mathcal{P}_i}$ from the initial state (recall that processes are deterministic). Therefore we have $G \subset \{q \in \mathcal{R} \mid q \sim_{\{\mathcal{P}_i\}} q_0.\sigma\}$. If $K^{\mathcal{R}}_{\{\mathcal{P}_i\}}\phi\ (q_0.\sigma)$ holds, then ϕ holds in all states of $\{q \in \mathcal{R} \mid q \sim_{\{\mathcal{P}_i\}} q_0.\sigma\}$ and by the previous inclusion ϕ also holds for all states in G. Thus we have $K^{PR}_{\mathcal{P}_i}\phi\ (\sigma)$. □

3.2.1 Representation and Computation

In order to compute the knowledge with perfect recall of the process $\mathcal{P}_i$, we build its support automaton $\mathcal{K}_i$ as in [8, 16]. The support automaton $\mathcal{K}_i$ will follow the execution of observable interactions for $\mathcal{P}_i$, that is, all interactions in A_i. The remaining interactions in $U_i = \mathcal{I} \setminus A_i$ are not observable by $\mathcal{K}_i$. Informally, the state reached in $\mathcal{K}_i$ after any sequence $\sigma \in \mathcal{I}^*$ summarizes all the global states that can be reached after any sequence $\sigma' \in \mathcal{I}^*$ such that σ and σ' are undistinguishable by $\mathcal{P}_i$. Formally, $\mathcal{K}_i$ is defined as the LTS $(S_i, A_i, \rightarrow^{PR}_i)$ where:

- The set of states $S_i = 2^Q$ correspond to subsets of the global states Q.
- Given a state s and an interaction a, the transition relation $\rightarrow^{PR}_i$ contains one transition $s \xrightarrow{a}{}^{PR}_i s'$, where $s' = \{q' \in Q \mid \exists q \in s,\ \exists \sigma\ \sigma \sim_{\mathcal{P}_i} a \wedge q.\sigma = q'\}$. Informally, for any state s, its successor s' through interaction a contains the set of global states q' that are reached from global states q in s by executing any sequence of unobservable interactions and exactly one a.
- The initial state is $s_{0i} = \{q \in Q \mid \exists \sigma \in (U_i)^*,\ q_0.\sigma = q\}$. Informally, s_{0i} contains all global states reachable by executing any sequence of unobservable interactions starting from the initial global state q_0.

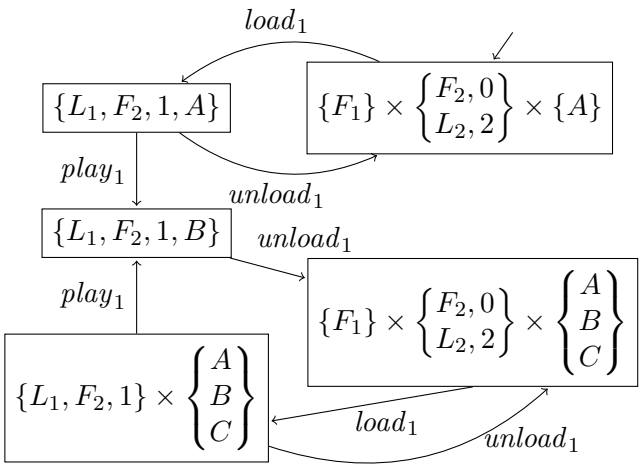

Figure 3.3: Support automata for the process Disc_1.

Example 3.11. The support automaton for the process Disc_1 is presented in Figure 3.3. In the initial state of the system, the interaction $load_2$ can take place. Since this interaction is not visible to Disc_1, it cannot distinguish between the two global states $F_1, F_2, 0, A$ and $F_1, L_2, 2, A$. Therefore the initial state of the support automaton is the set formed by these two global states. The process Disc_1 knows that until the first $play_1$ interaction happens, the state of the Listener process has to be A. After this first occurrence, the process Disc_1 loses track of the other processes state, except immediately after an occurrence of $play_1$, where it knows the global state of the system. This is an example where the knowledge with perfect recall gives more information than the state-based knowledge, which cannot distinguish after and before executing the interaction $play_1$. In particular, the process Disc_1 knows that an execution sequence containing two successive occurrences of $play_1$ cannot happen.

3.3 Related Works about Knowledge

The formalization of different kinds of knowledge and the obtained logic have been intensively studied [40, 50, 51].

A kind of sets for which only an upper and a lower bound are known are called *rough sets* [71]. In rough sets theory, objects are defined through a set of attributes. Intuitively, an attribute can be the shape, the color, the weight ... of the object. Each object is fully identified by the definition of its attributes. Given a subset of the objects, deciding whether a given object is part of that subset is always achieved by observing all attributes. Restricting the set of attributes that are observed creates a rough set, that approximates the subset. In that case indeed, there could be some objects whose membership in the subset depends on an unobservable attribute. One of the question in this theory is to find a minimal set of attributes whose observation is sufficient to

distinguish any two objects [85].

In [77], Knowledge is applied to decentralized control of a plant. A plant is an automaton whose interaction are labeled by actions, some of them being forbidden. Multiple decentralized controllers are in charge of controlling the plant, through allowing or not a given subset of the action. Each controller is defined by the set of actions it can observe and the set of actions it can control (i.e. execute). Knowledge is applied to allow each controller to infer which actions are legal from the current state and thus can be executed. An extension to distributed knowledge is proposed, whenever the information available to only one controller is not enough to decide. A criterion, called "Kripke observability" decides whether the extension to distributed knowledge is enough to control the plant.

In [12, 19, 8], the focus is on distributed controllers for executing Petri nets constrained by a given property. An example of such a constraining property is a priority order. Processes are defined as sets of Petri nets transitions. A transition can be common to several processes, which describes a synchronization. Each process can observe its neighborhood, that is the places that are adjacent to its transitions. In [12, 8], Knowledge is used to build a support table for each process. This table indicates, for each local configuration, which interaction can be safely executed. Knowledge based on the state of the neighborhood is not always sufficient, two possible extensions are proposed. The first one consists in using knowledge with perfect recall. The second one, also proposed in [48] consists in accumulating knowledge through additional synchronizations between processes. This additional synchronization is handled by a multiparty interaction protocol; α-core [73] is proposed. In [19], an optimization is proposed by considering only executions satisfying the constraining property in order to build the knowledge. This approach allows reducing the state space and possibly increase the knowledge of each process.

4 High-level Models: BIP and BIC

The BIP –Behavior/Interaction/Priority– and BIC –Behavior/Interaction/Condition– frameworks [10] aim at rigorous design, analysis and implementation of complex systems. Such systems are described as a set of *atomic components*, composed by a layered application of *glue operators*.

BIP provides two glue operators, namely Interaction and Priority. Interaction describes multiparty interactions between atomic components, as defined in 2.1, although in a more compact way. Priority is a partial order between interactions, as defined in 2.4.1. Therefore, the BIP framework encompasses prioritized multiparty interactions as presented in Chapter 2.

BIC provides a variant of BIP, where Priority is replaced by another operator, Condition. Condition expresses influence of non-participating components on the execution of an interaction. More precisely, an interaction can take place only when non-participating components satisfy a given state predicate.

We show that Priority rules can be rewritten using Condition. Therefore we can use BIC to represent BIP models.

This Chapter is structured as follows. The abstract model of BIP and BIC are described in Section 4.1 as an abstract formalization of the layers of Behavior, Interaction, Priority and Condition. Section 4.2 describes the concrete model of BIP extended with data.

4.1 Abstract Models of BIP and BIC

We provide a formalization of the BIP and BIC frameworks focusing on their individual layers. For Behavior, Interaction and Priority, the abstract model is very similar to the one presented in Chapter 2. In this section, we introduce formally each layer and then give their semantics.

4.1.1 Modeling Behavior

An atomic component is the most basic BIP or BIC entity, which represents behavior. A behavior is very similar to a process, as defined in Definition 2.1 from Section 2.1. A formal definition for the behavior of a BIP atomic component is given below:

Definition 4.1 (Abstract Behavior)**.** A behavior B is a labeled transition system represented by a triple $(Q, P, \rightarrow)$, where:

- Q is a finite set of control states,

- P is a finite set of communication ports,
- $\rightarrow \subseteq (Q \times P \times Q)$ is a set of transitions, each labeled by a port.

For a pair of states $q, q' \in Q$ and a port $p \in P$, we write $q \xrightarrow{p} q'$, if and only if $(q, p, q') \in \rightarrow$ and we say that p is enabled at state q. If such q' does not exist, we write $q \overset{p}{\not\longrightarrow}$ and we say that p is disabled at state q.

This definition differs from the Definition 2.1 in two aspects. First, the set of states is required to be finite. Second, the labels on the transitions are ports and not interaction labels. These ports constitute the interface of the behaviors and are used to specify their composition.

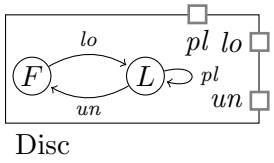

Figure 4.1: An example of abstract behavior.

Example 4.2. Figure 4.1 depicts a simple behavior, which corresponds to the example from Figure 2.1. Formally this behavior is defined as $(Q, P, \rightarrow)$, where $Q = \{F, L\}$, $P = \{lo, un, pl\}$ and $\rightarrow = \{(F, lo, L), (L, pl, L), (L, un, F)\}$. The squares represent ports that constitute the interface of the component.

4.1.2 Modeling Glue

A glue is a set of operators composing behaviors. Throughout this subsection, we consider n behaviors $B_1, \ldots, B_n$, where for each $i \in \{1, \ldots, n\}$, $B_i = (Q_i, P_i, \rightarrow_i)$. We assume that their respective sets of ports and sets of states are pairwise disjoint, i.e., for all $i \neq j$, we have $P_i \cap P_j = \emptyset$ and $Q_i \cap Q_j = \emptyset$. We define the set $P = \bigcup_{i=1}^{n} P_i$ of all ports in the system, and the set of global states $Q = Q_1 \times \ldots \times Q_n$. We consider three composition operators, namely Interaction, Priority and Condition. Priority and Condition assume that an Interaction operator has already been defined.

Interaction

In the BIP framework, interactions are explicitly defined as sets of ports. This differs with the model from Chapter 2, where interactions are implicitly defined using a common label.

Definition 4.3 (Interaction)**.** An interaction a is a non-empty subset $a \subseteq P$ of ports, such that $\forall i \in \{1, \ldots, n\}$, $|a \cap P_i| \leq 1$. We often denote $a = \{p_i\}_{i \in I}$, where $I \subseteq \{1, \ldots, n\}$ contains the indices of components participating in a and p_i is the only port in $P_i \cap a$.

An interaction operator is specified by a set of interactions $\gamma \subseteq 2^P$.

Priority

As motivated in Section 2.4, priority rules are expressed as a partial order on the interactions.

Definition 4.4 (Priority)**.** A priority operator is defined by a relation $\pi \subset \gamma \times Q \times \gamma$, such that $\forall q \in Q$, the relation $\pi_q = \{(a, a') \in \gamma \times \gamma \mid (a, q, a') \in \pi\}$ is a partial order.

If $a \pi_q a'$, interaction a has less priority than interaction a' at state q.

This definition differs from Definition 2.11 since the partial order over the interactions depends here upon the current global state. Priorities as in Definition 2.11, i.e. such that the partial order is the same at each global state, are called *static* priorities.

Condition

A condition operator associates to each interaction a state predicate that must be true for the interaction to execute.

Definition 4.5 (Condition)**.** A condition operator κ associates a state predicate to each interaction. Formally, $\kappa : \gamma \to \mathcal{B}^Q$.

If $a \in \gamma$, we denote by κ_a the predicate $\kappa(a)$. Intuitively, κ_a must hold to authorize execution of a.

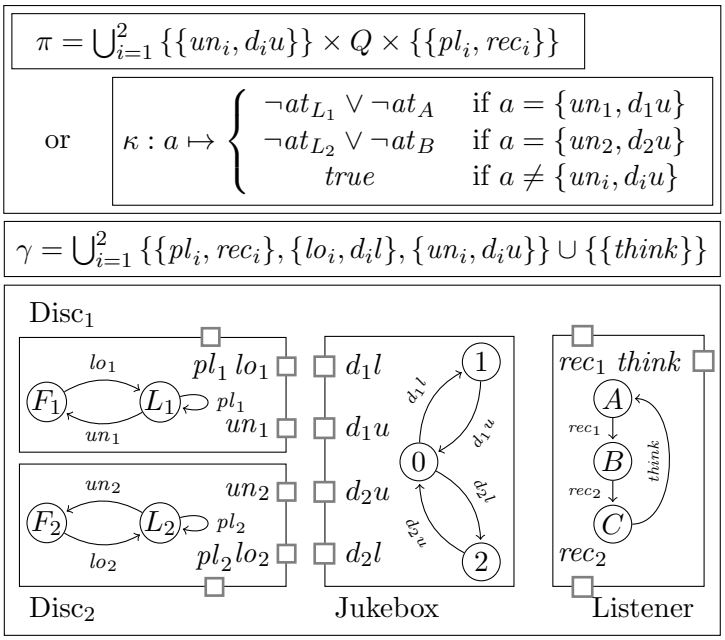

Figure 4.2: An example of abstract composition of 4 components using Interaction and either Priority or Condition.

Example 4.6. Figure 4.2 presents an example of an abstract model. It corresponds to the system of processes presented in Figure 2.2. Note that the last layer can either be a priority π or a condition κ and both possibilities are represented. This model contains 4 atomic components and 7 interactions. The interaction $\{think\}$ is a unary interaction involving only one port. Note that by naming the interactions, $play_i = \{pl_i, rec_i\}$, $load_i = \{lo_i, d_il\}$ and $unload_i = \{un_i, d_iu\}$, we obtain the same interaction labels as in Figure 2.2.

The priority π states that at each state $q \in Q$, the interaction $unload_i$ has less priority than the interaction $play_i$. This implements the static priority provided in Section 2.4 for this Jukebox example. The condition κ associates a non trivial predicate to the corresponding low priority interactions, i.e. $unload_i$ interactions.

4.1.3 Composition of Abstract Models

A composite component is obtained by application of a glue GL on components $B_1, \ldots, B_n$, which is denoted $GL(B_1, \ldots, B_n)$. We consider three different glues, namely Interaction γ, Interaction subject to Priority $\pi\gamma$ and Interaction subject to Condition $\kappa\gamma$. For each of them, we define the semantics below.

Definition 4.7 (Composition with Interaction γ)**.** The composition of the atomic components $B_1, \ldots, B_n$, parameterized by a set of interactions $\gamma \subseteq 2^P$, is a transition system $B = (Q, \gamma, \rightarrow_\gamma)$, where:

- $Q = Q_1 \times \ldots \times Q_n$ is the set of global states,
- $\rightarrow_\gamma$ is the least set of transitions satisfying the rule:

$$\frac{a = \{p_i\}_{i \in I} \in \gamma \qquad \forall i \in I \ q_i \xrightarrow{p_i}_i q_i' \qquad \forall j \notin I \ q_j = q_j'}{(q_1, \ldots, q_n) \xrightarrow{a}_\gamma (q_1', \ldots, q_n')}$$

Consider the model presented in Figure 4.2, without taking priority into account. The semantics of this component is the LTS depicted in Figure 2.3.

Definition 4.8 (Composition with Interaction γ subject to Priority π)**.** Let $B = (Q, \gamma, \rightarrow_\gamma)$ be the behavior of the composite component $\gamma(B_1, \ldots, B_n)$. We define the behavior of $\pi\gamma(B_1, \ldots, B_n)$ as the LTS $B' = (Q, \gamma, \rightarrow_\pi)$, where $\rightarrow_\pi$ is the least set of transitions satisfying the rule:

$$\frac{a \in \gamma \qquad q \xrightarrow{a}_\gamma q' \qquad \forall a' \ a \pi_q a' \implies q \overset{a'}{\not\longrightarrow}_\gamma}{q \xrightarrow{a}_\pi q'}$$

Note that the two above semantics are slight generalizations of the ones defined in Chapter 2. However, the composition with Interaction subject to Condition is new.

Definition 4.9 (Composition with Interaction γ subject to Condition κ)**.** Let $B = (Q, \gamma, \rightarrow_\gamma)$ be the behavior of the composite component $\gamma(B_1, \ldots, B_n)$. We define the behavior of $\kappa\gamma(B_1, \ldots, B_n)$ as the LTS $B' = (Q, \gamma, \rightarrow_\kappa)$, where $\rightarrow_\kappa$ is the least set of transitions satisfying the rule:

$$\frac{a \in \gamma \qquad q \xrightarrow{a}_\gamma q' \qquad \kappa_a(q)}{q \xrightarrow{a}_\kappa q'}$$

As said before, a Condition operator simply associates a predicate to each interaction. This predicate depends on the global state of the system, and not only on the states of the participants in the interaction. For the interaction to execute, this predicate must be true, which is stated in the last inference rule.

4.1.4 Priority vs. Condition

We show that given a Priority π one can obtain a Condition κ^π such that the behaviors of the composite components with priority and observation are identical. This is done by forbidding execution of an interaction a if a higher priority interaction is enabled at the current state.

Using at_ℓ predicates, we define the predicate EN_a stating whether the interaction a is enabled. First, we define the predicate $EN^i_{p_i}$ characterizing enabledness of port p_i in a component $B_i = (Q_i, P_i, \rightarrow_i)$. Formally, ${EN_{p_i}}^i = \bigvee_{\{q_i | q_i \xrightarrow{p_i}_i\}} at_{q_i}$. Then, the predicate EN_a can be defined by: $EN_a = \bigwedge_{p_i \in a} EN^i_{p_i}$. Given a global state $q = (q_1, \ldots, q_n) \in Q$, we denote by at_q the predicate $at_{q_1} \wedge \ldots \wedge at_{q_n}$.

Definition 4.10 (Condition obtained from Priority)**.** Given the composite component $\pi\gamma(B_1, \ldots, B_n)$, we define the *Condition obtained from Priority* κ^π as follows. For each interaction $a \in \gamma$, we define

$$\kappa^\pi_a = \bigwedge_{a \, \pi_q \, a'} at_q \implies \neg EN_{a'}$$

The predicate κ^π_a associated to the interaction a is false whenever there is a priority rule for the current state q where a has low priority and the higher priority interaction a' is enabled. The predicate prevents executing a whenever it is forbidden according to the priority π.

Example 4.11. Consider the model from Figure 4.2. We explicit the predicates associated to low priority interactions in the Condition κ^π obtained from Priority π. Since the priority expressed in this model is a static priority, i.e. $unload_i$ has less priority than $play_i$, the predicate associated to $unload_i$ can be simplified into $\neg EN_{play_i}$. Consider the interaction $play_1$. It is enabled whenever both ports pl_1 and rec_1 are enabled, that is when both places L_1 and A are active. Thus $EN_{play_1} = at_{L_1} \wedge at_A$. Similarly $EN_{play_2} = at_{L_2} \wedge at_B$. Finally, the predicate associated to $unload_1$ is $\kappa^\pi_{unload_1} = \neg at_{L_1} \vee \neg at_A$. Similarly $\kappa^\pi_{unload_2} = \neg at_{L_2} \vee \neg at_B$.

Assume that the global state is $(L_1, F_2, 1, A)$. The enabled interactions are $unload_1$ and $play_1$. In the model with Priority, the execution of interaction $unload_1$ is forbidden as $play_1$ is enabled. In the model with Condition, the predicate $\kappa^{\pi}_{unload_1}$ is not satisfied, which also forbids execution of $unload_1$.

Proposition 4.12. *The composite components* $\pi\gamma(B_1, \ldots, B_n)$ *and* $\kappa^{\pi}\gamma(B_1, \ldots, B_n)$ *have the same behavior, that is* $\rightarrow_{\pi} = \rightarrow_{\kappa^{\pi}}$.

Proof. Let $q \in Q$ be a global state and $a \in \gamma$ be an interaction. If a can be executed from state q according to $\rightarrow_{\pi}$, then for all a' such that $a\,\pi_q\,a'$, a' is not enabled. Therefore the predicate $\neg EN_{a'}$ holds. Thus, the predicate κ^{π}_{a} also holds, and we have $q \xrightarrow{a}_{\kappa^{\pi}}$. Similarly, if $q \xrightarrow{a}_{\kappa^{\pi}}$, then all interaction a' such that $a\,\pi_q\,a'$ are disabled and $q \xrightarrow{a}_{\pi}$. Finally, the two transitions have the same destination state q' which is given by $q \xrightarrow{a}_{\gamma} q'$. □

4.2 Concrete Model of BIP

The abstract models from the previous section focus on control. We now present how data is handled in BIP (and in BIC). Data allows more succinct representation for complex behavior like *guards* expressed over variables to prevent transitions and interactions. Data is modified through *update functions*. The mechanisms to handle data added on top of the abstract model constitute the concrete model of BIP.

4.2.1 Atomic Components

In BIP, *atomic components* are Petri nets equipped with a set of ports and a set of variables. Each transition is guarded by a predicate on variables, triggers an update function, and is labelled by a port. Ports are used for communication among different components and are associated with some variables of the component.

Definition 4.13. An *atomic component* B is defined by $B = (L, P, T, X, \{X_p\}_{p\in P}, \{g_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T})$ where:

- (L, P, T) is a *Petri net* (Definition 2.7).
- X is a set of variables.
- For each port $p \in P$, $X_p \subseteq X$ is the set of variables exported by p (i.e., variables visible from outside the component through port p).
- For each transition $\tau \in T$, g_τ is a predicate defined over X and f_τ is a function that updates the set of variables X.

It is required that the Petri net with its initial marking is 1-safe. As in the abstract model, ports represents the interface of the component. However, the interface also includes variables associated to ports.

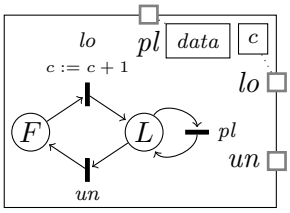

Figure 4.3: An atomic component.

Example 4.14. Figure 4.3 shows a concrete atomic component. The 1-safe Petri net is actually an automaton, namely the one from Figure 2.1. It has been extended with 2 variables, *data* and c. The variable *data* is associated to the port *play*, that is the value of *data* may be read and modified by an interaction involving the port *play*. This variable is not read or modified locally. The variable c is associated to the port *load*. The update function associated to the only transition labeled by *play* increments the value of c. Note that the value incremented is actually the value after the interaction has taken place.

Defining a semantic for the atomic components requires a notion of state. The state of an atomic component is described in two parts: the control state and the state of the variables. The *control state* is a marking of the Petri net. Since the Petri is 1-safe, a marking m is a subset of the places, i.e. $m \in 2^L$. The state of the variables is given by a valuation. We assume that all variables are defined over a data domain $\mathcal{D}$. Given a set X of variables, we denote by $\mathcal{D}^X$ the set of valuations defined on X. Formally, $\mathcal{D}^X = \{\sigma : X \to \mathcal{D}\}$. If $X' \subseteq X$, and $v \in \mathcal{D}^X$, $v' \in \mathcal{D}^{X'}$, we denote by $v[X' \leftarrow v']$ the valuation of X defined as

$$v[X' \leftarrow v'] : x \mapsto \begin{cases} v'(x) & \text{if } x \in X' \\ v(x) & \text{otherwise} \end{cases}$$

Formally, a guard g over a set of variables X is a function $\mathcal{D}^X \to \mathcal{B}$. Similarly, an update function f over a set of variables X is a function $\mathcal{D}^X \to \mathcal{D}^X$.

Definition 4.15. The semantic of an atomic component $B = (L, P, T, X, \{X_p\}_{p \in P}, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$ is defined as the labeled transition system $S_B = (Q_B, P_B, \to_B)$ where

- $Q_B = 2^L \times \mathcal{D}^X$, where $\mathcal{D}^X$ denotes the set of valuations on X.
- $P_B = P \times \mathcal{D}^X \times \mathcal{D}^X$ denotes the set of labels, that is, ports augmented with valuations of variables.
- $\to_B$ is the set of transitions defined as follows. Let (m, v) and (m', v') be two states in $2^L \times \mathcal{D}^X$, p be a port in P, and v_p^{up}, v_p^{dn} be two valuations in $\mathcal{D}^{X_p}$. We write $(m, v) \xrightarrow{p(v_p^{up}, v_p^{dn})}_B (m', v')$, iff $\tau = (m, p, m')$ is a transition of the behavior of the Petri net (L, P, T), $g_\tau(v)$ is true, $v_p^{up} = v|_{X_p}$ and $v' = f_\tau(v[X_p \leftarrow v_p^{dn}])$, (i.e., v' is obtained by applying f_τ after updating variables X_p by the values v_p^{dn}). In this case, we say that p is *enabled* in state (m, v).

In the LTS describing the semantics of an atomic component, each transition is labeled by a port and 2 valuations of variables exported by that port. During an interaction, these variables may be read and modified. The valuation v_p^{up} describes the values of the variables exported by the port p before the interaction. These values may be read during an interaction involving p. These values may also be modified during the interaction. The valuation v_p^{dn} returns these modified values. Note that the LTS representing the semantics of an atomic component can be seen as the abstract component from Definition 4.1 with an infinite set of states and an infinite set of ports.

4.2.2 Interactions and Connectors

As for atomic components, we extend the definition of interactions to handle variables. The support of an interaction is a set of ports. Furthermore, each interaction can read and write the set of variables exported by its support. More precisely, a predicate on these variables, called guard, has to be true for the interaction to be enabled. A data transfer function modifies the variables upon execution of the interaction.

In the sequel, we assume a set of atomic component $\{B_1, \ldots, B_n\}$, where for each $i \in \{1, \ldots, n\}$, B_i is defined as $(L_i,\ P_i,\ T_i,\ X_i,\ \{X_p\}_{p \in P_i},\ \{g_\tau\}_{\tau \in T_i},\ \{f_\tau\}_{\tau \in T_i})$ and its semantics is given by $(Q_i, P_{B_i}, \rightarrow_{B_i})$. We require that the sets of control locations, ports and variable of these components are pairwise disjoint (i.e., $i \neq j$ implies that $L_i \cap L_j = \emptyset$, $P_i \cap P_j = \emptyset$ and $X_i \cap X_j = \emptyset$). We denote by $P = \bigcup_{i=1}^{n} P_i$ the set of all ports, $X = \bigcup_{i=1}^{n} X_i$ the set of all variables and $Q = Q_1 \times \ldots \times Q_n$ the set of global states.

Since two distinct components have disjoint sets of variables, there is no shared variable. Each variable is local to a component. However, the values of the variables may be exchanged between components through an interaction.

Definition 4.16 (Interaction)**.** An *interaction* a is a triple (P_a, G_a, F_a), where:

- $P_a \subseteq P$ is a set of ports such that $\forall i \in \{1, \ldots, n\}, |P_a \cap P_i| \leq 1$. We denote $P_a = \{p_i\}_{i \in I}$, where $I \subseteq \{1, \ldots, n\}$ is the indices of participants in a and p_i is the only port in $P_a \cap P_i$.
- G_a is a predicate defined over the set $X_a = \bigcup_{i=1}^{n} X_{p_i}$ of variables involved in a.
- F_a is a data transfer function that modifies the variables X_a.

We denote by $\mathit{participants}(a)$ the set of components that have ports involved in a. Formally, $\mathit{participants}(a) = \{B_i \mid P_i \cap a \neq \emptyset\}$.

Connectors As in the abstract model, interactions are used to parameterize the first layer of glue. In order to avoid an explicit enumeration of all possible interactions, BIP introduces the notion of *connector*. Each connector Γ is defined over a given set of ports P, that forms its *support* and defines a set of interactions γ_Γ, that is, a subset of 2^P. A connector exports a port that can be reused in another, higher-level, connector. This export mechanism allows construction of hierarchical connectors.

A connector is defined by typing each port of its support as a *trigger* or as a *synchron*. BIP connectors form an algebra which is defined and studied in [22]. We do not define formally this algebra, nor explain how the typing of the support port defines the set of interaction of a connector. However, we give some intuitive examples of hierarchical connectors.

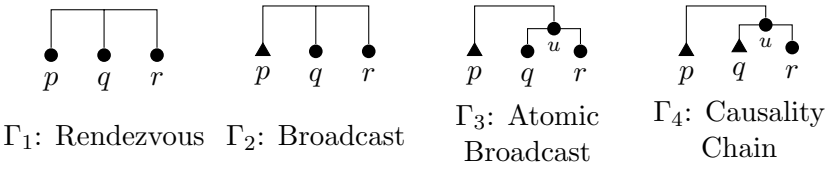

Figure 4.4: Examples of connectors and hierarchical connectors.

Example 4.17. In Figure 4.4, we graphically depict 3 connectors defined on the same support $\{p, q, r\}$. The connectors Γ_1 and Γ_2 are basic patterns.

- The connector Γ_1 is called rendezvous, or strong synchronization. Here, all ports are synchrons. The intuitive meaning of the synchron is that it has to wait for other ports in order to execute the interaction. This connector defines only one interaction: $\gamma_{\Gamma_1} = \{pqr\}$ (to lighten the notations, we write pqr instead of $\{p, q, r\}$).
- The connector Γ_2 is called broadcast. It includes one trigger (port p) and two synchrons. The intuitive meaning of a trigger is that it can initiate the interaction, even if all other ports are not enabled. This connector describes the set of all interactions that contains at least p, that is $\gamma_{\Gamma_2} = \{p, pr, pq, pqr\}$.

The two following connectors are hierarchical connectors obtained by combining broadcast and rendezvous. The combination of connectors is obtained by including in the support of a *top* connector Γ_t the exported port of a bottom connector Γ_b. Thus in the following, the bottom connector Γ_b refers to the one exporting the port u, and the top connector Γ_t refers to the other one. Intuitively, the set of interactions allowed by the hierarchical combination of the connectors is obtained by "replacing" each occurrence of u in γ_{Γ_t} with the set of interactions allowed by Γ_b. More precisely, if γ_{Γ_t} contains an interaction $p_1 \dots p_k u$, then the hierarchical connector allows all interactions formed as the union of $p_1 \dots p_k$ with an interaction allowed by Γ_b.

- The connector Γ_3 is called atomic broadcast. The bottom connector Γ_b is a rendezvous with support $\{q, r\}$ and exports a port u. This connector allows only interaction qr. The top connector Γ_t is a broadcast defined on the support $\{p, u\}$ and thus allows interactions p and pu. Therefore Γ_3 allows both p and pqr.
- The connector Γ_4 is called causality chain. The top connector allows the same interactions as previously: $\gamma_{\Gamma_t} = \{p, pu\}$. The bottom connector is a broadcast allowing both q and qr. Therefore, Γ_4 allows p, pq and pqr. In particular, for r to interact, it requires that both p and q interact as well.

For each example presented above, the corresponding set of interactions is defined formally from the basic typing of ports using the algebra from [22].

Connectors provide a mechanism for hierarchically handling variables. Intuitively, the mechanism implies two phases, an upward propagation, to decide whether the guard of the interaction is true, and a downward propagation, in case of execution, to compute the values to return. The full definition of a connector associates two functions to each interaction (instead of a single data transfer function). The first one, called U, compute the values to export for the top port of the connector during the upward propagation. The second one, called D, compute the values to return in the support ports during the downward propagation.

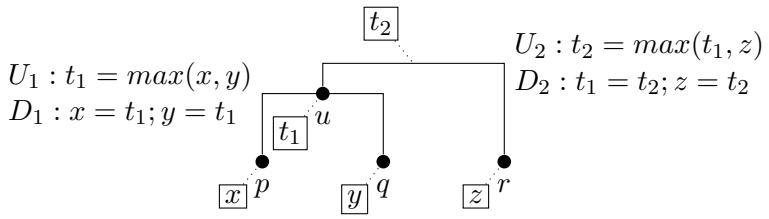

Figure 4.5: A hierarchical connector computing the maximum of exported values.

As an example, consider the connectors depicted in Figure 4.5. The support of this set of connectors is made of ports p, q and r, exporting respectively variables x, y and z. The connector exporting the port u computes the max of x and y and this value is exported by u, since t_1 is bound to u. This constitutes its upward propagation function U_1. The top connector then computes the maximum of t_1 and z and stores this value into t_2, as specified in U_2. This terminates the upward propagation. At any level, a guard prevents further upward propagation if it evaluates to false. If the interaction is selected for execution, then the downward propagation first sets both z and t_1 to the maximum value that is stored in t_2, according to D_2. Then D_1 sets both x and y to this maximum value. These propagations happen atomically, and t_i variables are not remembered between two interaction executions. The connector from Figure 4.5 behaves "as if" x, y and z where set to $max(x, y, z)$ in one atomic step.

In [30], the authors show that hierarchical connectors can be replaced by a set of "flat" interactions as defined in Definition 4.16. This is done by composing upward propagation functions guards, to obtain a guard for the interaction. Similarly, the update function is obtained by composing the downward propagation functions. More details about hierarchical connectors can be found in [11, 28].

4.2.3 Priority and Condition

Recall that a priority operator assigns a partial order between interactions to each state. However, since the state space Q may be infinite, priority operators are specified as a set of rules including a state predicate and an order between two interactions.

Definition 4.18 (Priorities)**.** Given a set γ of interactions defined over a set $\{B_1, \ldots, B_n\}$

of components, we define a priority as a relation $\pi \subseteq \mathcal{B}^Q \times \gamma \times \gamma$, such that $\forall q \in Q,\ \pi_q = \{(a, a') \in \gamma \times \gamma \mid C(q) \wedge (C, a, a') \in \pi\}$ is a partial order. We add $(C, a, a') \in \pi$ to express the fact that a has less priority than a' whenever the predicate C holds. We require that C depends only on variables visible by a or a'.

The predicate C depends on the variables exported by the participants in some interactions, allowing the corresponding priority rule to be dynamically enabled. A static priority is expressed by having $C = True$ for all $(C, a, a') \in \gamma$.

As for the abstract model, Condition is expressed by associating a state predicate κ_a to each interaction a in γ.

Definition 4.19 (Condition)**.** Given a set γ of interactions defined over a set $\{B_1, \ldots, B_n\}$ of components, we define a Condition as a function $\kappa : \gamma \to \mathcal{B}^Q$. This function associates a state predicate κ_a to each interaction a in γ.

The notation remains the same as for the abstract model. However, a global state $q \in Q$ contains both the control state and a valuation of the variables. Therefore, a predicate κ_a is expressed using at_ℓ predicates as well as predicates depending on the variables in X. Given a predicate ϕ defined over a set of variables X, we denote by $usedin(\phi)$ the set of variables appearing in ϕ. In particular, $usedin(\kappa_a)$ is the set of control locations and variables that appear in κ_a. This is useful to define the set of variables that have to be known to decide whether κ_a holds.

Expressing Condition by referring to the inner state of the components breaks the encapsulation principle, where only the interface (ports and exported variables) are visible outside the component. This implies that Condition cannot be used in a source model, but rather in an intermediate model, as it facilitates further implementation. Therefore, we do not provide a specific syntax for expressing Condition in a concrete model.

4.2.4 Composition of Components

We compose the atomic components using the same three glues as for the abstract model. The main difference is that a concrete model handles data within interactions.

Given a set of components $\{B_1, \ldots, B_n\}$ and a set of interactions γ, we denote by $\gamma(B_1, \ldots, B_n)$ the composition of these components using the set of interactions γ. Similarly, given a Priority π and a condition κ, we denote by $\pi\gamma(B_1, \ldots, B_n)$ and $\kappa\gamma(B_1, \ldots, B_n)$ the behaviors obtained by applying the corresponding glues.

Definition 4.20 (Composite Component: Semantics)**.** The behavior of a composite component $\gamma(B_1, ..., B_n)$, where B_i is an atomic component with semantics $S_{B_i} = (Q_i, P_{B_i}, \to_i)$, is a labeled transition system $(Q, \gamma, \to_\gamma)$, where

- $Q = Q_1 \times \ldots \times Q_n$,

- $\rightarrow_\gamma$ the least set of transitions satisfying the rule:

$$\frac{a = (\{p_i\}_{i\in I}, G_a, F_a) \in \gamma \qquad G_a(\{v_i^{up}\}_{i\in I}) \qquad \{v_i^{dn}\}_{i\in I} = F_a(\{v_i^{up}\}_{i\in I}) \qquad \forall i \in I\ q_i \xrightarrow{p_i(v_i^{up}, v_i^{dn})}_i q_i' \qquad \forall i \notin I.\ q_i' = q_i}{(q_1, \ldots, q_n) \xrightarrow{a}_\gamma (q_1', \ldots, q_n')}$$

Intuitively, this inference rule specifies that a composite component $B = \gamma(B_1, ..., B_n)$ can execute an interaction $a \in \gamma$, iff (1) for each port $p_i \in P_a$, the corresponding atomic component B_i allows a transition from the current state labeled by p_i (according to its semantics), and (2) the guard G_a of the interaction evaluates to true with the current values of the exported variables. If these conditions hold for an interaction a at global state q, a is *enabled* at that state, denoted by $q \xrightarrow{a}_\gamma$. The execution of a modifies the variables of participating component by first applying the data transfer function F_a on exported variables and then update functions inside each interacting component (according to their semantics). The (local) states of components that do not participate in the interaction remain unchanged.

We define the behavior of the composite component $\pi\gamma(B_1, \ldots, B_n)$ with priority, as the labeled transition system $(Q, \gamma, \rightarrow_\pi)$ where $\rightarrow_\pi$ is the least set of transitions satisfying the rule:

$$\frac{q \xrightarrow{a}_\gamma q' \qquad \forall C, a'\ s.t.\ (C, a, a') \in \pi \quad C(q) \implies q \not\xrightarrow{a'}}{q \xrightarrow{a}_\pi q'}$$

The inference rule filters out interactions which are not maximal with respect to the priority order, according to the current state. In particular, if a priority rule (C, a, a') holds at the current state, that is the corresponding state predicate C is verified, then execution of a requires that a' is not enabled.

We define the behavior of the composite component $\kappa\gamma(B_1, \ldots, B_n)$ with Condition as the labeled transition system $(Q, \gamma, \rightarrow_\kappa)$ where $\rightarrow_\kappa$ is the least set of transitions satisfying the rule:

$$\frac{q \xrightarrow{a}_\gamma q' \qquad \kappa_a(q)}{q \xrightarrow{a}_\kappa q'}$$

This rule is actually the same as for the abstract model, since the state of the components now includes the variables.

Example 4.21. Figure 4.6 shows a graphical representation of a composite component in BIP. It is a variation of the example from Figure 2.2. Variables have been added in components: each disc has its own counter c_i and some data $data_i$, the jukebox has a counter C and the listener has one variable *sample* for each disc. Each variable c_i is incremented whenever the corresponding disc is loaded; the variable C counts the total number of loads. The guard on each *load* interaction (between brackets) balances the number of loads for each disc, i.e. a disc can be loaded only when its own counter does

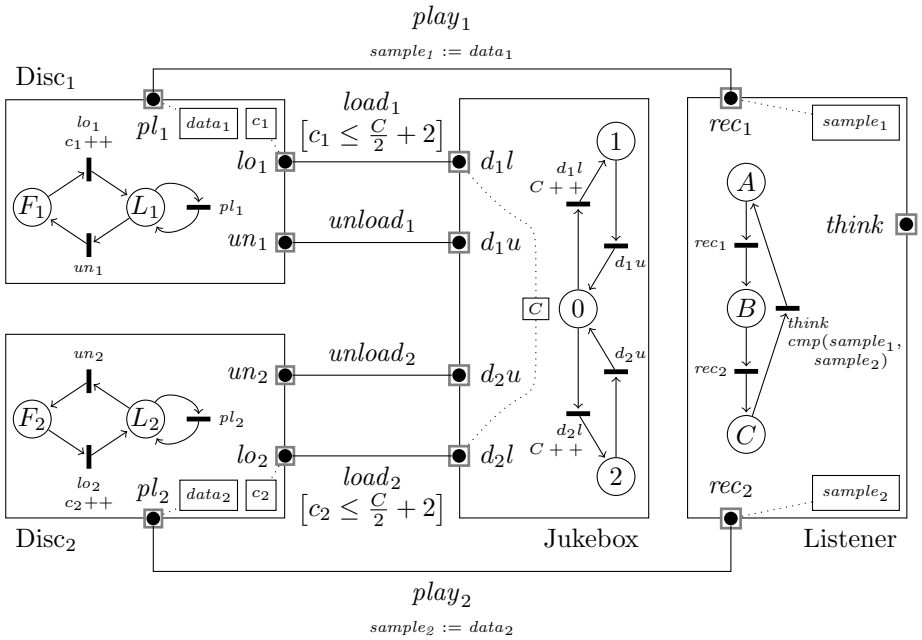

Figure 4.6: A composite component.

not exceed too much the average. During a *play* interaction, the data from the disc is copied in the variable *sample* of the listener.

Recall that the static priority operator described in Section 2.4 gives more priority to the *play* interaction than to the *unload* interaction. Such a priority is described as $\pi = \{(true, unload_1, play_1), (true, unload_2, play_2)\}$. Since none of these rules depends on variables, the Condition predicates defined in Subsection 4.1.4, namely $\kappa^{\pi}_{unload_1} = \neg at_{L_1} \vee \neg at_A$ and $\kappa^{\pi}_{unload_2} = \neg at_{L_2} \vee \neg at_B$, are still implementing this priority.

For a multiparty interaction, a component is either participant or non-participant. Condition was introduced in [17] to allow for an intermediate between participation and non-participation. This intermediate role is held by components that are observed by the interaction to decide whether the Condition predicate holds, but do not participate in the interaction. We denote by $observed_{\kappa}(a)$ the set of such components. Formally, given a BIC model $\kappa\gamma(B_1, \ldots, B_n)$, $observed_{\kappa}(a)$ is the set of atomic components $\{B_i \mid (L_i \cup X_i) \cap usedin(\kappa_a) \neq \emptyset\} \setminus participants(a)$. We denote by $invl_{\kappa}(a)$ the set of components that are involved in the interaction either as a participant or as an observed component. Formally, $invl(a) = participants(a) \cup observed_{\kappa}(a)$.

4.2.5 Discussion

In [23], the authors compare the expressiveness of glue operators found in different models, e.g. Calculus of Communicating Systems (CCS) [67], Communicating sequential processes (CSP) [52], Synchronous Calculus of Communicating Systems (SCCS) [68]

and BIP. Intuitively, a glue operator gl_1 is strongly more expressive than another one gl_2 if any composition using gl_2 can be rewritten in an equivalent composition using gl_1, without modifying components nor adding new components. A glue operator gl_1 is weakly more expressive than another one gl_2 if any composition using gl_2 can be rewritten in an equivalent composition using gl_1, by adding coordination components. It has been shown that the BIP glue without priority is strongly more expressive than the other models cited above. Furthermore, the BIP glue with priority is strongly more expressive than BIP glue without priority, and BIP glue without priority is not weakly more expressive than BIP glue with priorities.

Rewriting Priority Operators as Condition Operators

The definition of the Condition κ^π obtained from a Priority π has to be adapted to handle variables as well. Adding variables changes the definition of the EN_a predicates, as they must take into account guards on transitions in atomic components as well as guards on interactions. The predicate indicating whether a port p_i of B_i is enabled becomes

$$EN_{p_i}: \begin{array}{lcl} Q_i & \rightarrow & \mathcal{B} \\ q_i & \mapsto & \exists q_i' \exists v_{p_i}^{up} \exists v_{p_i}^{dn} \quad (q_i, (p_i, v_{p_i}^{up}, v_{p_i}^{dn}), q_i') \in \rightarrow_i \end{array}$$

where $\rightarrow_i$ is the transition relation of B_i semantics. The predicate EN_a takes into account the predicates EN_{p_i} and the guard G_a of the interaction. Assume that $P_a = \{p_i\}_{i \in I}$.

$$EN_a: \begin{array}{lcl} Q & \rightarrow & \mathcal{B} \\ ((m_1, v_1), \ldots, (m_n, v_n)) & \mapsto & G_a\left(\{v_i|_{X_{p_i}}\}_{i \in I}\right) \wedge \forall i \in I \quad EN_{p_i}((m_i, v_i)) \end{array}$$

Recall that a priority rule is active if the associated predicate C holds. The Condition κ^π obtained from Priority π is, for each interaction a:

$$\kappa_a^\pi = \bigwedge_{(C,a,a') \in \pi} C \implies \neg EN_{a'}$$

The resulting semantic $\rightarrow_{\kappa^\pi}$ is equal to $\rightarrow_\pi$. Indeed, κ_a^π is equivalent to $\forall C, a'$ $s.t.$ $(C, a, a') \in \pi \;\; C(q) \implies q \overset{a'}{\not\longrightarrow}$ and replacing κ_a^π by the latter expression yields the rule for prioritized execution.

Encoding Condition Operators with Interaction Operators

A model with Condition can be transformed in an equivalent model expressed using only Interaction, provided that atomic components are modified. This transformation implements Condition predicates from the original model as guards on the interactions of the built model. To this end, each interaction a is extended to form a new interaction a' such that components originally observed in a become participants in a'. This requires a special port in observed components, that is used for exporting the state of the component. The control state of the component is stored in an additional variable, denoted

@, that is updated on every transition. As the control state is the marking of a 1-safe Petri net, @ contains the set of currently active places. This variable is exported by all ports of the component along with the data needed to evaluate Condition predicates.

Formally, let $\kappa\gamma(B_1, \ldots, B_n)$ be a BIC model. We transform each atomic component $B_i = (L_i, P_i, T_i, X_i, \{X_p\}_{p\in P_i}, \{g_\tau\}_{\tau\in T_i}, \{f_\tau\}_{\tau\in T_i})$ into an observable component $B'_i = (L_i, P'_i, T'_i, X'_i, \{X_p\}_{p\in P'_i}, \{g'_\tau\}_{\tau\in T'_i}, \{f'_\tau\}_{\tau\in T'_i})$ where:

- $L'_i = L_i$ the control locations are not changed,
- $P'_i = P_i \cup \{obs_i\}$, with obs_i being an additional port,
- $T'_i = T_i \cup \bigcup_{\ell\in L}\{(\{\ell\}, obs_i, \{\ell\})\}$, for each control location ℓ, τ_{obs_ℓ} is an additional transition,
- $X'_i = X_i \cup \{@_i\}$, with $@_i$ being an additional variable to store the current control state of the component,
- Let $X^\kappa_i = \{@_i\} \cup \bigcup_{a\in\gamma}(usedin(\kappa_a) \cap X_i)$ be the set of variables from B_i that are needed to evaluate the Condition predicates. The port obs_i exports the variables $X_{obs_i} = X^\kappa_i$, and each original port $p \in P_i$ exports the variables $X'_p = X_p \cup X^\kappa_i$,
- for each $\tau \in T_i$, $g'_\tau = g_\tau$, the guards are not modified. For each ℓ, the new transition τ_{obs_ℓ} is guarded by $g_{\tau_{obs_\ell}} = true$,
- for each ℓ, the update function of τ_{obs_ℓ} is the identity function. For each transition $\tau \in T_i$, the new update function f'_τ first applies f_τ and then updates $@_i$ as follows: $@_i := (@_i \setminus {}^\bullet\tau) \cup \tau^\bullet$.

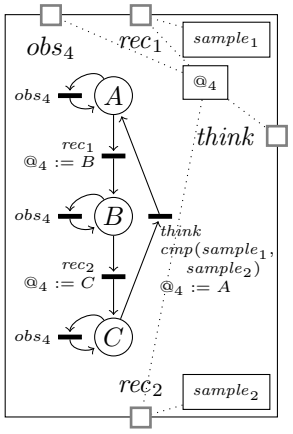

Figure 4.7: Observable version of the Listener component from Figure 4.6.

The variable @ initially contains the initial marking of the atomic component. Note that if the underlying Petri net is an automaton, the transformed component is also an automaton. In that case, updating the @ variable falls back to simply assigning it the

destination control state of the transition. For instance, in Figure 4.7, the control is described as an automaton. The added variable $@_4$ is exported by all the ports.

Given a BIC model $\kappa\gamma(B_1, \ldots, B_n)$, the equivalent model using only Interaction operator is $\gamma'(B'_1, \ldots, B'_n)$, where B'_i is the observable version of B_i and γ' is defined as follows. For each interaction $a \in \gamma$, γ' contains the interaction a' such that:

- $P_{a'} = P_a \cup \bigcup_{B_i \in observed_\kappa(a)} obs_i$ contains original ports in a as well as *obs* ports of the components observed by a,
- $G_{a'} = G_a \wedge G_{\kappa_a}$, where G_{κ_a} is obtained by replacing each at_ℓ predicate occurring in κ_a by the predicate $\ell \in @_i$, where i is the index such that $\ell \in L_i$,
- $F_{a'} = F_a$ is not modified.

Again, if atomic components are actually automata, an at_ℓ predicate can simply be written as $@_i = \ell$. Consider the interaction $unload_1$ from the example in Figure 4.6. The associated Condition predicate is $\kappa_{unload_1} = \neg at_{L_1} \vee \neg at_A$, meaning that the Listener component is observed by $unload_1$. Consequently, the set of ports involved in $unload'_1$ additionally contains obs_4. The port un_1 of the observable version of Disc_1 exports the state variable $@_1$, which indicates the current control state of Disc_1. The guard $G_{unload'_1} = \neg(@_1 = L_1) \vee \neg(@_4 = A)$ holds in states equivalent to the ones where κ_{unload_1} holds.

A global state q of $\kappa\gamma(B_1, \ldots, B_n)$ can be obtained from a global state q' of $\gamma'(B'_1, \ldots, B'_n)$ by ignoring the valuation of the $@_i$ variables. We denote $q = equ(q')$ in that case. By construction, in every reachable state q' of $\gamma'(B'_1, \ldots, B'_n)$, each state variable $@_i$ contains the set of control locations that are active at the current marking m_i. From a global state q of $\kappa\gamma(B_1, \ldots, B_n)$, one builds a state q' of $\gamma'(B'_1, \ldots, B'_n)$ by assigning to each variable $@_i$ the set of active places at marking m_i. We denote $q' = equ'(q)$ in that case. It always holds that $q = equ(equ'(q))$. Furthermore, for every reachable state q' of $\gamma'(B'_1, \ldots, B'_n)$, we have $q' = equ'(equ(q'))$. We use this bijection defined on the reachable states to compare $\rightarrow_\kappa$ and $\rightarrow_{\gamma'}$.

If $q \xrightarrow{a}_\kappa r$ then we have $q \xrightarrow{a}_\gamma r$ and $\kappa_a(q)$ holds. Since all ports *obs* are always enabled, all ports of the interaction a' are equivalently enabled at state $equ'(q)$. Since $q \xrightarrow{a}$, G_a evaluates to true at q and thus at $equ'(q)$. Since $\kappa_a(q)$ holds, $G_{\kappa_a}(equ'(q))$ holds. Therefore at state $equ'(q)$ all ports of a' are enabled and the guard of a' hold, thus $equ'(q) \xrightarrow{a'}_{\gamma'} r'$. Since the effect of executing a from q and a' from q' is the same except on $@_i$ variables, we have $r = equ(r')$ and thus, $r' = equ'(r)$.

Similarly, if $q' \xrightarrow{a'}_{\gamma'} r'$, we have $equ(q') \xrightarrow{a}_\kappa equ(r')$. Therefore, $\rightarrow_{\gamma'}$ and $\rightarrow_\kappa$ are equivalent.

This construction proves that from a BIP model with priority, one can build an equivalent model without priority that has the same behavior. However, this result does not contradicts the greater expressiveness of the glue with priority, as building the equivalent model requires to modify the components, and not only the glue.

5 Breaking Atomicity of Interactions: Parallelism Between Components

A BIP port could be defined as a basic synchronization unit, an interaction being formed of several such units. Parallelism between components stems from independent computations between two successive synchronizations. Independent parallel computation requires to distribute components execution in different processes.

As we assume that distributed processes can either send a message, perform internal computation or wait for a message, there is no direct distributed implementation for ports and interactions. Therefore, we implement them through a two-way handshake protocol. First, each component sends an offer to a centralized engine and then the engine replies by indicating which interaction (and thus which port) to execute.

This distribution scheme separates the choice of the next interaction to execute from the inner computations in the components. The centralized engine is a dedicated process responsible for the first task. The second task is distributed, by creating a new process for each atomic component.

In this Chapter, we start by enumerating in Section 5.1 the restrictions that we impose on the source and target BIP/BIC models that we consider. Then, we define formally the transformation from the centralized to distributed model in Section 5.2, and prove its correctness in Section 5.3. Finally, we optimize the number of messages needed to implement Condition in Section 5.4.

5.1 Model Restrictions

The restrictions that we impose on the source (centralized) model aim at simplifying the transformation towards a distributed model. The restrictions on the target model aim at implementability on a distributed platform. More precisely, the target model must be simple enough to be implemented using basic (i.e. Send/Receive) message-passing primitives.

The restrictions on the source model are as follows:

- We assume that our source model is flat, i.e. does not have hierarchical connectors. These restriction can be met by using the flattening tool from Subsection 7.2.3, which replaces all hierarchical constructs from a BIP model by an equivalent set of flat constructs. Furthermore, we require that only synchrons are used to describe connectors. This can be obtained by replacing a connector with a trigger by a set of connectors implementing the same set of interactions.

- We assume that the Petri nets defining transitions between control locations in atomic components are restricted to automata. Given a Petri Net with an initial state, one can consider its behavior, as defined in Subsection 2.1.1. This behavior is actually an automaton whose states are markings of the Petri net, and there is a transition between two states if a Petri Net move is allowed between the corresponding markings. In the worst case, the automaton representing the global behavior of a Petri net has a number of states exponential in the number of places in the Petri net.

  Furthermore, we require that the automaton is deterministic. For each port, there is at most one enabled transition labeled by this port at each state. In particular, two transitions outgoing from the same control state and labeled with the same port must have mutually exclusive guards.

- We assume that the source model is written using the Condition operator, not the Priority operator. As explained in Subsection 4.1.4, Priority operators can be rewritten using Condition operators.

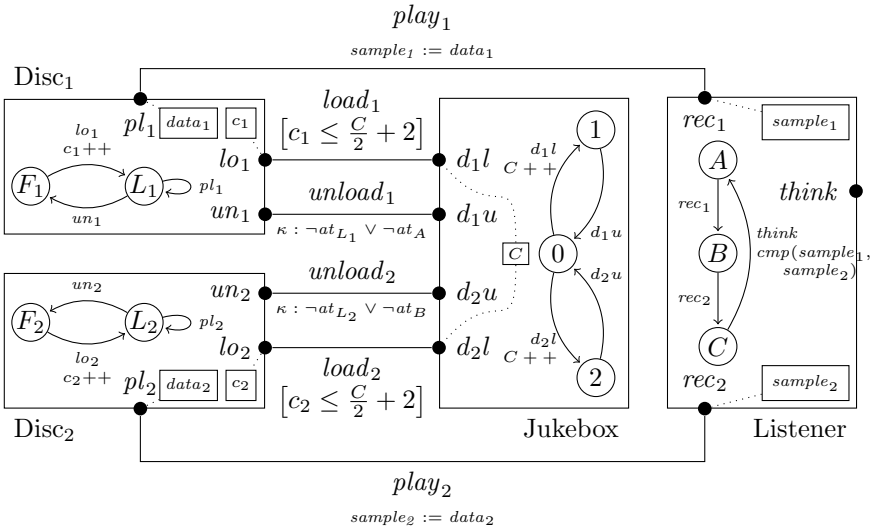

Figure 5.1: Version of the model from Figure 4.6 taking restrictions into account.

Example 5.1. In Figure 5.1, we present an equivalent version of the jukebox composite component presented in Figure 4.6, which meets all the restrictions. Petri nets in atomic components have been replaced by automata – in this particular case, it correspond to simply replacing transitions by arrows. We will use this model as running example, in particular for Condition obtained from Priority. Priority rules for the model from Figure 4.6 are $unload_1 \,\pi\, play_1$ and $unload_2 \,\pi\, play_2$. As explained in Subsection 4.1.4, these priority rules can be rewritten using Condition as follows: $\kappa^{\pi}_{unload_1} = \neg at_{L_1} \vee \neg at_A$ and $\kappa^{\pi}_{unload_2} = \neg at_{L_2} \vee \neg at_B$.

The target model aims to be implementable using basic message-passing primitives. The execution of a distributed process is a sequence of actions that are either message emission, message reception or internal computation. Consequently our target model includes three types of ports: send ports, receive ports and unary ports. Unary ports correspond to internal computation. They can only appear in unary interactions, that is interactions involving only one component. Send and receive ports appear only in message-passing interactions. Such an interaction has no guard, and the update function copies variables exported by the send port to variables exported by the receive port. In a canonic message-passing environment, each send action has a well-defined recipient. Therefore, we require that each send port participates in exactly one Send/Receive interaction. The latter ensures that for each send port there is a unique corresponding receive port. Finally, the target model is described using synchronizations, whereas message-passing is sender-initiated. Therefore, to the above syntactic restrictions, we add a semantic restriction: at each state where a send port is enabled, the corresponding receive port will become enabled unconditionally within a finite number of transitions in the receiver component. This condition guarantees that the built model does not rely on the sender for waiting until the receiver is ready to receive, which is not the most general case.

The class of BIP models satisfying the above restriction are called Send/Receive BIP models.

Definition 5.2. We say that $B^{SR} = \gamma^{SR}(B_1^{SR}, \ldots, B_n^{SR})$ is a *Send/Receive* BIP composite component iff we can partition the set of ports of B^{SR} into three sets P_s, P_r, and P_u that are respectively the sets of *send ports*, *receive ports*, and *unary ports*, such that:

- Each interaction $a = (P_a, G_a, F_a) \in \gamma^{SR}$, is either (1) a Send/Receive interaction with $P_a = \{s, r\}$, $s \in P_s$, $r \in P_r$, $G_a = true$ and F_a copies the variables exported by port s to the variables exported by port r, or, (2) a unary interaction $P_a = \{p\}$ with $p \in P_u$, $G_a = true$, F_a is the identity function.

- If s is a port in P_s, then there exists one and only one Send/Receive interaction $a = (P_a, G_a, F_a) \in \gamma^{SR}$ with $P_a = \{s, r\}$ and the port r is a receive port. We say that r is the receive port associated to s.

- Let $a = (P_a, G_a, F_a)$ with $P_a = \{s, r\}$ be a Send/Receive interaction in γ^{SR}. Let B_i^{SR} be the component exporting r. If s is enabled at some reachable state of B^{SR}, then B_i^{SR} will reach a state where s is enabled in a finite number of transitions labeled by either send or unary ports.

Our target model contains only interactions and no priorities or conditions. We graphically denote a send port by a trigger (▲) and a receive or a unary port by a synchron (●).

5.2 Transformation from Centralized to Distributed Model

We present here the formal definition of the distributed Send/Receive model obtained from a centralized BIC model $\kappa\gamma(B_1, \ldots, B_n)$. We first explain how atomic components

are transformed in Subsection 5.2.1. Then we build a centralized engine that coordinates distributed atomic components in Subsection 5.2.2. Finally, we define the connections between these newly built Send/Receive atomic components in Subsection 5.2.3.

5.2.1 Breaking Atomicity in Components

We transform an atomic component B of a BIC model into a Send/Receive atomic component B^{SR} by decomposing each "atomic" synchronization into a send and a receive action. The idea is the same as in [9] and in Subsection 2.2.1. The synchronization between participants is implemented as a two-phase protocol between the atomic components and the engine. First B^{SR} sends an *offer* through a dedicated send port, then it waits for a notification arriving on a receive port. The offer contains the information to determine whether an interaction is enabled and whether the Condition predicates of observing interaction hold. An offer includes the set of enabled ports of B^{SR} through which the component is currently ready to interact and the values of relevant variables. Relevant variables include variables exported by the enabled ports, as they may be read and written during an interaction. Furthermore, a Condition predicate may depend on the variables and control location of any atomic component. Consequently, relevant variables also include the current control location of the component and the variables that are needed to evaluate the Condition predicates. Since these additional relevant variables depend on the whole model, we add a parameter, denoted X_i^κ for the component B_i, indicating which variable values are to be sent. Formally, given a full BIC model $\kappa\gamma(B_1, \ldots, B_n)$, the set of variables to send for Condition is $X_i^\kappa = X_i \cap \bigcup_{a\in\gamma} \mathit{usedin}(\kappa_a)$.

We encode enabled ports and current control state as follows. For each port p of the transformed component B^{SR}, we introduce a Boolean variable x_p. We add a state variable @ that contains the current control location. This variable ranges over the set L of control locations of the atomic component. The newly added variables are modified by an *update* function when reaching a new state. The variable x_p is then set to true if the corresponding port p becomes enabled, and to false otherwise. Similarly, before reaching the control state ℓ, the variable @ is set to ℓ.

Since each notification from the engine triggers an internal computation in a component, following [9], we split each control location ℓ into two control locations, namely, ℓ itself and a *busy control location* $\perp_\ell$. Intuitively, reaching $\perp_\ell$ marks the beginning of an unobservable internal computation. We are now ready to define the transformation from B into B^{SR}.

Definition 5.3. Let $B = (L,\ P,\ T,\ X,\ \{X_p\}_{p\in P},\ \{g_\tau\}_{\tau\in T},\ \{f_\tau\}_{\tau\in T})$ be an atomic component, and $X^\kappa \subseteq X$ be a set of variables needed by the Condition layer. The corresponding Send/Receive atomic component is $B^{SR} = (L^{SR},\ P^{SR},\ T^{SR},\ X^{SR},\ \{X_p^{SR}\}_{p\in P},\ \{g_\tau\}_{\tau\in T^{SR}},\ \{f_\tau\}_{\tau\in T^{SR}})$, such that:

- $L^{SR} = L \cup L^\perp$, where $L^\perp = \{\perp_\ell \mid \ell \in L\}$.
- $X^{SR} = X \cup \{x_p\}_{p\in P} \cup \{@\}$, where each x_p is a new Boolean variable, and @ is a control location variable.

- $P^{SR} = P \cup \{o\}$, where the *offer* port o exports the variables $X_o^{SR} = \{@\} \cup \bigcup_{p \in P}(\{x_p\} \cup X_p) \cup X^\kappa$, that is the state variable, the new Boolean variables as well as the exported variables associated to each port, and the variables needed for the Condition layer. For all other ports $p \in P$, we keep $X_p^{SR} = X_p$.

- For each location $\ell \in L$, we include an offer transition $\tau_\ell = (\bot_\ell, o, \ell)$ in T^{SR}. The guard g_{τ_ℓ} is true and the update function f_{τ_ℓ} is the identity function.

- For each transition $\tau = (\ell, p, \ell') \in T$, we include a response transition $\tau_p = (\ell, p, \bot_{\ell'})$ in T^{SR}. The guard g_{τ_p} is true. The function f_{τ_p} first applies the original update function f_τ, then sets the state variables to the next control location (i.e. $@ := \ell'$) and finally updates the Boolean variables:

$$\text{for all } r \in P \quad x_r := \begin{cases} g_{\tau'} & \text{if } \exists \tau' = (\ell', r, \ell'') \in T \\ \mathit{false} & \text{otherwise} \end{cases}$$

As said before, control locations that are in $L^\bot$ are called *busy control locations*. Such control locations are transient as the only outgoing transition, labeled by a send port, are controlled by the distributed atomic component. Control locations L from the original atomic components are said to be *stable*. Indeed, once it reached such a control location, a distributed atomic component will remain in that state until it receives a notification. In general, we say that a distributed atomic component is in a stable state if its current control location is stable and in a busy state if its current control location is busy.

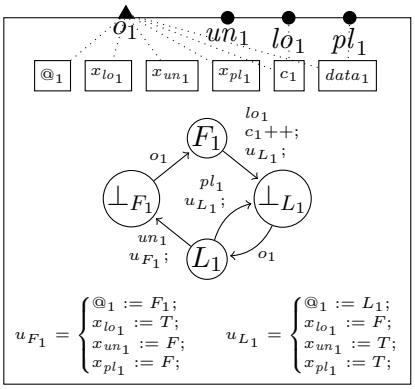

Figure 5.2: Distributed version of the Disc_1 atomic component from Figure 5.1.

Example 5.4. Figure 5.2 shows the distributed version of the atomic component Disc_1 from Figure 5.1. It corresponds to the BIP version of the distributed process depicted in Figure 2.7. Only the Condition predicate associated to $unload_1$ depends on Disc_1. Since this predicate only requires to know the state of Disc_1, we have $X^\kappa = \emptyset$. For each original control location ℓ, a busy control location $\bot_\ell$ has been added, with a transition labeled by the offer port o_1 from $\bot_\ell$ to ℓ. Before executing these transitions, the update functions u_{L_1} or u_{F_1} are called. Here, when reaching $\bot_{F_1}$, the variable $@_1$ is set to F_1

and the only guard variable set to true is x_{lo_1}. The next offer transition ($\perp_{F_1} \xrightarrow{o_1} F_1$) sends these refreshed values to the engine through the port o_1, so that the engine has up-to-date informations from the component Disc_1.

5.2.2 Implementing the Engine in BIP

As explained in [9], the engine works with a partial view of the global state. Initially, all components are doing their initial computation and the engine does not know their state until they send offers. Each offer received indicates to the engine the state of the sender component, and thus gives more information about the global state. If the engine does not take any decision, it will eventually receive all offers and know the global state.

For the sake of distributed implementation, it is worth taking a decision as soon as possible. In the absence of priority, it is always correct to execute an interaction when all participating components have sent an offer and are ready to execute the interaction. Executing a low priority interaction requires to check that all higher priority interaction are not currently enabled and won't be enabled if the engine waits until it knows the global state. Using Condition, the negative premise corresponding to the priority rule is replaced by a predicate. If all components that are involved in the predicate have sent an offer and the predicate holds, then it is safe to execute the interaction, since new offers cannot disable it.

To summarize, the engine has to listen to the offers, to take decisions based only on the offers received so far and to notify components when an interaction has taken place. We recap here the construction of the engine as initially described in [24], and add support for Condition. We represent the engine as a Petri net and associate each atomic component to a token of this Petri net. From the engine point of view, a component can have three different status, thus the associated token cycles between three kinds of place, as illustrated on Figure 5.3:

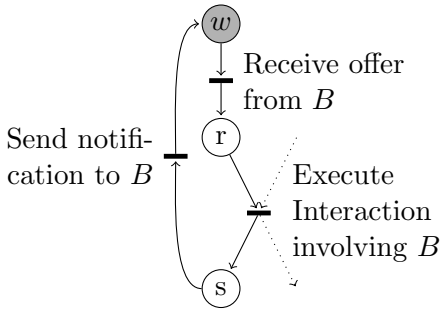

Figure 5.3: Circuit of a single token, corresponding to a component B, in the engine.

- *waiting* place: Initially, and after each notification, the engine does not know the state of the component until it sends an offer. In that case the engine is waiting for the component offer. There is one waiting place for each component. In Figure 5.3, the waiting place is labeled by w.

- *received* place: The token is there when an offer has been received, and the engine has not scheduled any interaction involving the component. There is one received place for each component. In Figure 5.3, the received place is labeled by r.

- *sending* place: The token is there when an interaction involving the component has been scheduled. There is one such place for each port of the atomic components, thus indicating which port is involved in the scheduled interaction. In Figure 5.3, there is a single sending place labeled by s.

Each token starts in the waiting place. It moves to its received place when an offer from the corresponding atomic component is received. An interaction involving this component yields a Petri net transition taking several tokens from the receive places and putting them in the send places corresponding to the ports involved in the interaction. From the send place, the token moves back to the waiting place when sending the notification to the atomic component.

Before defining the engine, we recall the different degrees of involvement for a given component B_i in a given interaction a. A component B_i is a participant in a if it exports a port that is part of a. We denote $\mathit{participants}(a)$ the set of such components. A component B_i is observed by a if the state of B_i is needed to decide whether κ_a holds, but is not participant in a. We denote $\mathit{observed}_\kappa(a)$ the set of such components. A component that is participant in an interaction needs to be notified whenever the interaction occurs, to execute its inner computation. To the contrary, an observed component does not need to be notified that it has been observed. Finally, we denote by $\mathit{invl}_\kappa(a)$ the set of all components involved in the execution of a, i.e. $\mathit{invl}_\kappa(a) = \mathit{participants}(a) \cup \mathit{observed}_\kappa(a)$.

Definition 5.5. Let $B = \kappa\gamma(B_1, \ldots, B_n)$ be a BIC model. The centralized engine for this model is defined as the Send/Receive BIP component $E = (L^E,\ P^E,\ T^E,\ X^E,\ \{X_p\}_{p \in P^E},\ \{g_\tau\}_{\tau \in T^E},\ \{f_\tau\}_{\tau \in T^E})$

- The set L^E of places is the union of the *waiting places* $\{w_i \mid i \in \{1, \ldots, n\}\}$, the receive places $\{r_i \mid i \in \{1, \ldots, n\}\}$ and the *sending places* $\{s_p \mid p \in P\}$. Recall that P is the set of all ports exported by the components $B_1, \ldots, B_n$.

- The set X^E of variables contains a copy of each variable exported by an offer port $\{x^E \mid x \in \bigcup_{i \in \{1,\ldots,n\}} X^{SR}_{o_i}\}$.

- The set P^E of ports is $\{o_i^E \mid i \in \{1, \ldots, n\}\} \cup \{p^E \mid p \in P\} \cup \gamma$, which are respectively receive ports, send ports and unary ports. The variables attached to each port o_i^E for $i \in \{1, \ldots, n\}$ are $X_{o_i^E} = \{x^E \mid x \in X^{SR}_{o_i}\}$. Similarly, the variables attached to each port p^E for $p \in P$ are $X_{p^E} = \{x^E \mid x \in X_p\}$. Unary ports do not have any attached variables.

- The set T^E of transitions consists of the following:
  - *receive offer*: for each $i \in \{1, \ldots, n\}$, T^E contains the transition (w_i, o_i^E, r_i). This transition has no guard nor update function.

– *send notification*: for each port $p \in P$, let i be the index of the component exporting p, i.e. such that $p \in P_i$, T^E contains the transition (s_p, p^E, w_i). This transition has no guard nor update function.

– *execute interaction* a: for each interaction $a \in \gamma$, T^E contains the transition $\tau_a = (\{r_i \mid B_i \in \mathit{invl}_\kappa(a)\}, a, \{s_p \mid p \in a\} \cup \{r_i \mid B_i \in \mathit{observed}_\kappa(a)\})$. This transition is guarded by

$$g^E_{\tau_a} = \bigwedge_{p \in a} x^E_p \wedge G^E_a \wedge \kappa^E_a$$

where the predicate ϕ^E is obtained by replacing each occurrence of a variable x in the predicate ϕ by its local copy x^E, and each occurrence of a at_ℓ predicate by the test $@^E_i = \ell$, where i is the index such that $\ell \in L_i$.

The associated update function $f^E_{\tau_a}$ is obtained by replacing each occurrence of a variable x in the data transfer function F_a by its local copy x^E, that is, $f^E_{\tau_a} = F^E_a$.

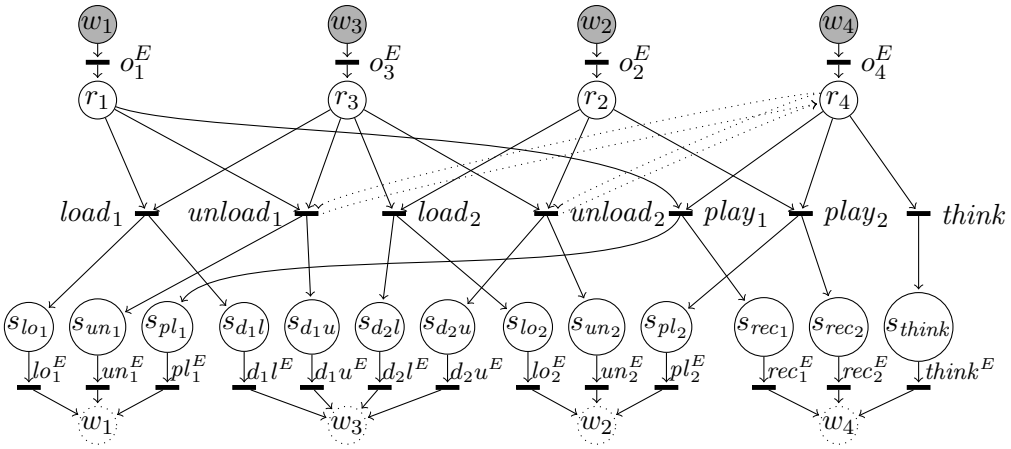

Figure 5.4: Petri net of the centralized engine for the model from Figure 5.1.

Example 5.6. Figure 5.4 presents the Petri net of the centralized engine obtained from the input model from Figure 5.1. Note that here we assume $B_1 = \mathrm{Disc}_1$, $B_2 = \mathrm{Disc}_3$, $B_3 = \mathrm{Jukebox}$ and $B_4 = \mathrm{Listener}$. The execution of the interaction $load_1$ requires that both offers from B_1 (Disc_1) and B_3 (Jukebox) have been received, since it takes tokens in places r_1 and r_3. Furthermore, this interaction is guarded by $x^E_{lo_1} \wedge x^E_{d_1l} \wedge \left[c^E_1 \leq \frac{C^E}{2} + 2\right]$, which ensures (1) that the port of the interaction are enabled and (2) that the guard of the original interaction evaluates to true. Note that this guard uses the local copy of the variables, which have been refreshed on the last offer as they are associated to the offer port. The execution of this transition puts one token in s_{lo_1} and one token in s_{d1_l}, indicating that a notification has to be sent on these ports. As the original interaction does not have a data transfer function, there is no update function on that transition.

The interaction $unload_1$ yields a similar transition, with additional (dotted) edges from and to the place r_4. Indeed, $unload_1$ is originally a low priority interaction and therefore the engine must ensure that $play_1$ cannot execute before executing $unload_1$. This is done through the Condition predicate $\neg at_{L_1} \vee \neg at_A$, which needs to check the state of B_4 (Listener), i.e. B_4 is in $observed_\kappa(unload_1)$. The guard of the transition $unload_1$ is $x^E_{un_1} \wedge x^E_{d_1u} \wedge \big(\neg(@^E_1 = L_1) \vee \neg(@^E_4 = A)\big)$. The dotted edges ensure that there is a token in the place r_4, meaning that the offer has been received and not consumed yet and therefore the value of $@^E_4$ is still valid.

5.2.3 Connecting the Engine and the Distributed Components

The whole distributed model is obtained by composing the distributed version of the atomic components and the engine using Send/Receive interactions.

Definition 5.7. Given a model $\kappa\gamma(B_1, \ldots, B_n)$, we define its distributed version with a centralized engine as the Send/Receive BIP model $\gamma^{SR}(B_1^{SR}, \ldots, B_n^{SR}, E)$, where γ^{SR} is defined as follows:

- for each component B_i, γ^{SR} contains the Send/Receive interaction $\{o_i, o_i^E\}$ where o_i is a send port and o_i^E is a receive port.
- for each port $p \in P$, γ^{SR} contains the Send/Receive interaction $\{p^E, p\}$ where p^E is a send port and p is a receive port.
- for each interaction $a \in \gamma$, γ^{SR} contains the unary interaction $\{a\}$, where a is the unary port exported by the engine.

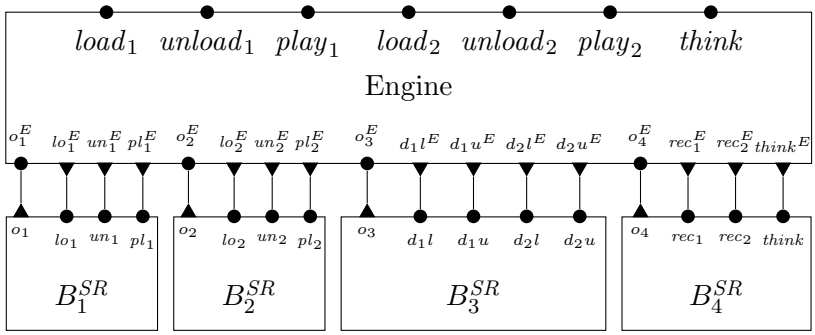

Figure 5.5: Global view of the solution with the centralized engine for the model from Figure 5.1.

Recall that a Send/Receive interaction copies variables attached to the send port to variables attached to the received port. Here, the offer interaction from component B_i copies the value of each variable $x \in X_{o_i}$ to the variable $x^E \in X^E_{o_i}$. Similarly, the notification copies the value of the variable $x^E \in X^E_p$ to the corresponding variable $x \in X_p$. Figure 5.5 presents the Send/Receive BIP model obtained from the model in

Figure 5.1. This centralized solution is very similar to the one provided in [9], with the difference that in our case, the engine is a component in the model. In particular, the glue provided by the Send/Receive interactions can be replaced by Send/Receive primitives. The Send/Receive model executed with the glue provided by low-level Send/Receive primitives yields the same behavior as the Send/Receive model executed according to the centralized BIP semantics.

5.3 Correctness

We show that the transformation developed in this chapter is correct, that is the distributed model is observationally equivalent to the original model. Before proving the observational equivalence, we show that the built model is a valid Send/Receive model.

5.3.1 Validity of the Target Model

The Definition 5.2 specifies the class of Send/Receive models. The first two criteria of this definition are syntactic, namely only Send/Receive or unary interaction are allowed and each send port participate in exactly one Send/Receive interaction. These criteria are met by the previous definition. The third criterion of the Definition states that whenever a send port is enabled, the associated receive port will unconditionally become enabled within a finite number of transitions in the receiver component. This criterion requires more thorough attention. We prove in Lemma 5.8 a stronger assumption, that is, the receive port is enabled as soon as the send port becomes enabled.

Lemma 5.8. *Let $\gamma^{SR}(B_1^{SR}, \ldots, B_n^{SR}, E)$ be the Send/Receive model obtained from $\kappa\gamma(B_1, \ldots, B_n)$. Then, at each reachable state of the Send/Receive model, whenever a send-port s is enabled, the associated receive-port r is already enabled.*

Intuitively, this property holds since each component starts listening to any notification as soon as it sends an offer. Dually, the scheduler starts listening again to offers from a given component as soon as it sends a notification to that component.

Proof: Let B_i^{SR} be a Send/Receive atomic component. We show that all Send/Receive interactions involving B_i^{SR} meet the statement of the lemma. Recall that each atomic component has an associated token in the engine. The token is either in a waiting place, in a receive place or in a sending place. We distinguish the following configuration, for the state of B_i^{SR} and the location of the corresponding token:

i) B_i^{SR} is in a busy control location and the corresponding token is in the w_i place. (This is the initial state.) The send-port o_i is enabled as well as the receive-port o_i^E. Thus, the property holds for the initial configuration, and in general for configurations of this form. Moreover, by executing this request interaction, we fall into the second configuration.

ii) B_i^{SR} is in a stable control location and the associated token is in the r_i place. From this configuration, no send-port involving a communication between B_i^{SR} and E is

enabled. Only the token in the engine can move, provided a unary interaction is executed.

iii) B_i^{SR} is in a stable state and the associated token is in a sending place s_p. At that state, the send port p^E is enabled. By construction, the place s_p can be reached only if the variable x_p was previously set to true by B_i^{SR} before sending the offer. Therefore, from the current (stable) state in B_i^{SR}, there is a transition labeled by the receive port p. Thus, the Send/Receive interaction $\{p^E, p\}$ is possible and by executing it, we reach back the first configuration. ■

5.3.2 Observational Equivalence

In order to prove that the model obtained after transformation implements the original model, we use the notion of observational equivalence. An *observational equivalence* is defined between two transition systems $A = (Q_A, P \cup \{\beta\}, \rightarrow_A)$ and $B = (Q_B, P \cup \{\beta\}, \rightarrow_B)$. It is based on the usual definition of weak bisimilarity [66], where β-transitions are considered unobservable.

Definition 5.9 (Weak Simulation)**.** A *weak simulation* over A and B is a relation $R \subseteq Q_A \times Q_B$ such that we have $\forall (q,r) \in R,\ a \in P :\ q \xrightarrow{a}_A q' \implies \exists r' :\ (q',r') \in R \wedge r \xrightarrow{\beta^* a \beta^*}_B r'$ and $\forall (q,r) \in R :\ q \xrightarrow{\beta}_A q' \implies \exists r' :\ (q',r') \in R \wedge r \xrightarrow{\beta^*}_B r'$

A weak bisimulation over A and B is a relation R such that R and R^{-1} are weak simulations. We say that A and B are *observationally equivalent* and we write $A \sim B$ if for each state of A there is a weakly bisimilar state of B and conversely.

To compare the original and the Send/Receive model, we have to precise which interactions are observable. For the high-level BIC model the actions are the interactions and all interactions are observable. The corresponding actions are the unary interactions in the engine of the Send/Receive model. In particular, Send/Receive interactions are unobservable.

Let us fix some notations. Let $q^\perp, r^\perp \in Q^{SR}$ be two states of B^{SR} and $a \in \gamma^{SR}$ be an interaction such that $q^\perp \xrightarrow{\alpha}_{\gamma^{SR}} r^\perp$. We rewrite $q^\perp \xrightarrow{\beta}_{\gamma^{SR}} t^\perp$ if α is a Send/Receive interaction, otherwise $\alpha = a \in \gamma$ is a unary interaction and is observable in B^{SR}.

Lemma 5.10. *The transition system* $(Q^{SR}, \{\beta\}, \xrightarrow{\beta}_{\gamma^{SR}})$ *is terminating and confluent.*

Proof. The behavior of Send/Receive atomic components imposes that each offer is followed by a notification and each notification is followed by an offer. A notification can be immediately followed by an offer. However, between an offer and the corresponding notification, one of the unary interaction must occur to move the token associated to the component from the received to the waiting place. By doing only β transitions, at most one notification can be received and one offer can be sent by each component. Therefore the transition system terminates in at most $2n$ steps.

Recall that components are deterministic. By construction,β transitions affect independent places and variables in the engine. Therefore, if two β transitions can be

executed from a given state, both orderings are possible and reach the same state. Thus the transition system is confluent. □

Formally, for any state $q^\perp$, we denote by $[q^\perp]$ the unique state that is reached by executing all the possible Send/Receive interactions. The state $[q^\perp]$ verifies the property $q^\perp \rightarrow {\beta^*}_{SR}[q^\perp]$ and $[q^\perp] \stackrel{\beta}{\not\rightarrow}_{SR}$. Furthermore, Lemma 5.11 asserts that, at state $[q^\perp]$, atomic components are in a stable state and variables in the engine have the same value as the corresponding variables in components.

Lemma 5.11. *At state* $[q^\perp] = ((\ell_1, v_1), \ldots, (\ell_n, v_n), (m, v^E))$*, we have*

- $\forall i \in \{1, \ldots, n\}, \ell_i \notin L^\perp$*, and*
- $\forall x^E \in X^E$ *s.t.* $x \in X_i, v_i(x) = v^E(x^E)$.

Proof. By construction of the distributed atomic components, for each busy state $\perp_\ell$ there is one outgoing offer transition (to state ℓ) labeled by a send port, with a guard that is always true. According, to Lemma 5.8 the associated offer transition is always possible when the sender port is enabled. Therefore, at state $[q^\perp]$ every atomic component is in a stable state.

According to the proof of Lemma 5.8, a token in the engine cannot be in its waiting place whenever the corresponding component is in a stable state (unreachable configuration). Sending places cannot contain tokens, since the notification interactions would be possible, which contradicts $[q^\perp] \stackrel{\beta}{\not\rightarrow}_{SR}$. Thus, tokens are in received places at $[q^\perp]$. Furthermore, for each variable $x^E \in X^E$, the last modifying transition was the offer from the corresponding atomic component B_i, which ensures $v^E(x^E) = v_i(x)$. □

A direct corollary of Lemma 5.11 is that at state $[q^\perp] = ((\ell_1, v_1), \ldots, (\ell_n, v_n), (m, v^E))$, the value of each the variable $@_i^E$ in E is the same as in the atomic component. In the atomic components, the value of $@_i$ is the current stable state, therefore $v^E(@_i^E) = \ell_i$.

To prove the correctness of our transformation, we exhibit a relation between the states of the original model and the states of the distributed model and prove that it is an observational equivalence. We define the relation by assigning to each state $q^\perp \in Q^{SR}$ an equivalent state $equ(q^\perp) \in Q$. This function considers only the states of distributed atomic components at $[q^\perp]$. According to Lemma 5.11, at $[q^\perp] = ((\ell_1^\perp, v_1^\perp), \ldots, (\ell_n^\perp, v_n^\perp), (m, v^E))$ atomic components are in stable control states. Therefore control states $\ell_i^\perp$ are valid control states for the original atomic components. Similarly, by restricting each valuation $v_i^\perp$ to the variables that are present in the original atomic component B_i, we obtain a valuation $v_i = v_i^\perp|_{X_i \setminus X_i^\perp}$ that is a valid valuation for B_i. With the previous notations, we define $equ(q^\perp) = ((\ell_1^\perp, v_1), \ldots (\ell_n^\perp, v_n))$.

Theorem 5.12. *Let* $B^{SR} = \gamma^{SR}(B_1^{SR}, \ldots, B_n^{SR}, E)$ *be the Send/Receive model obtained from* $B = \kappa\gamma(B_1, \ldots, B_n)$*. B and B^{SR} are observationally equivalent when hiding all Send/Receive interactions in B^{SR}.*

Proof. We define the relation $R = \{(q, q^\perp) \in Q \times Q^{SR} \mid equ(q^\perp) = q\}$. Let $q, r \in Q$ be some states of B, $q^\perp, r^\perp \in Q^{SR}$ be some states of B^{SR}, and $a \in \gamma$ an interaction. Since

there is no unobservable action in the original model, R is an observational equivalence if it meets the following properties:

(i) If $(q, q^\perp) \in R$ and $q^\perp \xrightarrow{\beta}_{\gamma^{SR}} r^\perp$, then $(q, r^\perp) \in R$.

(ii) If $(q, q^\perp) \in R$ and $q^\perp \xrightarrow{a}_{\gamma^{SR}} r^\perp$, then $\exists r \in Q$ such that $q \xrightarrow{a}_{\kappa} r$ and $(r, r^\perp) \in R$

(iii) If $(q, q^\perp) \in R$ and $q \xrightarrow{a}_{\kappa} r$ then $\exists r^\perp \in Q^{SR}$, such that $q^\perp \xrightarrow{\beta^* a}_{\gamma^{SR}} r^\perp$ and $(r, r^\perp) \in R$

The property (i) is a direct consequence of the confluence of $\xrightarrow{\beta}$: if $q^\perp \xrightarrow{\beta}_{\gamma^{SR}} r^\perp$ then Lemma 5.10 implies $[q^\perp] = [r^\perp]$ and thus $equ(q^\perp) = equ(r^\perp)$.

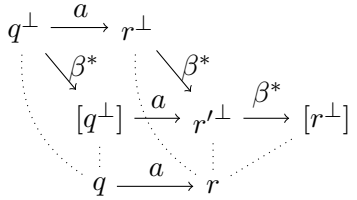

Figure 5.6: Point (ii) of the proof of Theorem 5.12.

To prove property (ii), we observe that the unary interaction a is enabled in the engine at state $q^\perp$. Therefore, offers from the participants in a as well as the components observed by a have been received. The values of variables corresponding to these components cannot be modified when reaching the state $[q^\perp]$, since β transitions, i.e. receiving other offers, modify distinct variables in the engine. As shown in Figure 5.6, if a is possible at state $q^\perp$, it remains possible at state $[q^\perp]$. Let us expand a as $\{p_i\}_{i \in I}$. If the transition a is possible in the engine, each variable $x^E_{p_i}$ is true, and by construction of the distributed atomic component B_i, we have $q_i \xrightarrow{p_i}_i$. By Lemma 5.11, if G^E_a evaluates to true in E then G_a evaluates at state true $equ(q^\perp) = (q_1, \ldots, q_n)$. Similarly, if κ^E_a evaluates to true in E, $\kappa_a(equ(q^\perp))$ also evaluates to true by Lemma 5.11. Thus we have $equ(q^\perp) \xrightarrow{a}_{\kappa} r$.

By applying update function F^E_a and executing notification transitions, one reaches a state where values in distributed atomic components participant in a is the same the values in original atomic component after executing a. These values are not changed through the subsequent offers, therefore $(r, [r^\perp]) \in R$ and since $r^\perp \xrightarrow{\beta} [r^\perp]$, we have $(r, r^\perp) \in R$.

To prove (iii), we assume that $q \xrightarrow{a}_{\kappa} r$ is valid in the original model. By definition of $[q^\perp]$, we have $q^\perp \xrightarrow{\beta^*} [q^\perp]$. As in the previous case, the Lemma 5.11 ensures that if the interaction a is possible in the original atomic component at state $equ(q^\perp)$, then the transition a is also possible in the engine at state $[q^\perp]$. Therefore we have $q^\perp \xrightarrow{\beta^*} [q^\perp] \xrightarrow{a} r^\perp$. As previously, executing the transition in the engine and sending notification to participating components yields the same results as executing the interaction in the original model. Thus $(r, r^\perp) \in R$. □

5.4 Taking Decision Earlier: Knowledge-Based Optimization

Before executing an interaction a, the engine has to wait for an offer from each component in $\mathit{participants}(a)$ and in $\mathit{observed}_\kappa(a)$. Since executing an interaction changes the state its participants, waiting for an offer from each of them cannot be avoided.

However, it is not always mandatory to wait for all observed components. Assume for instance an interaction a whose condition predicate is $\kappa_a = X \vee Y$, where X depends only on component B_1 and Y depends only on component B_2. In the case where B_1 has sent its offer and the offer satisfies X, then the predicate κ_a holds. In that case, a could safely be executed but the engine waits that all components in $\mathit{observed}_\kappa(a)$ have their offer. This pattern occurs very often with Condition predicates obtained from Priority, whenever κ_a checks that an interaction is not enabled. For instance, if a has less priority than b, which involves port p of B_1 and port q of B_2, then the predicate κ_a is $\neg EN_p^1 \vee \neg EN_q^2$.

In this section, we propose to use distributed knowledge based on the local state, as in Section 3.1, to replace a given Condition κ by a new one κ'. The new Condition κ' is based on knowledge obtained from an over approximation of the reachable states and, for each interaction a, a restricted set of components to observe.

There is a tradeoff between minimizing the number of observed components in κ' and maximizing the number of transitions valid for κ that are also detected to be valid by κ'. In the case of a Condition predicate $\kappa_a = X \vee Y$, where X is entirely determined by observing B_1, and Y by observing B_2, one could decide to observe only B_1. The resulting condition predicate $\kappa'_a = X$ would not detect that a can execute whenever only Y holds. To characterize how much of originally valid transitions are detected, we define *detection levels*. Such a detection level should guarantee, for instance, that no deadlocks are introduced.

In Subsection 5.4.1, we show how distributed knowledge can be used to compute a correct Condition layer, providing the set of components to observe is given. We also define two detection levels that characterize the system according to the observed components. In Subsection 5.4.2, we provide heuristics that try to minimize the number of observed components, while ensuring a given detection level.

5.4.1 Building a Condition with Reduced Observation

This Subsection defines a new Condition κ' built using the distributed knowledge of κ as presented in Section 3.1. This new Condition is parameterized by assigning to each interaction $a \in \gamma$ a set of components $\mathit{obs}(a)$ that restrict observation. We require that $\mathit{obs}(a) \cap \mathit{participants}(a) = \emptyset$. Furthermore, an over approximation $\tilde{\mathcal{R}}$ of the reachable states is needed to compute the knowledge. Such an approximation can be provided by the invariants presented in Subsection 7.2.2.

Definition 5.13. Given a BIC model $\kappa\gamma(B_1, \ldots, B_n)$ and sets of observed components $\{\mathit{obs}(a)\}_{a\in\gamma}$, the Condition κ' with reduced observation associates to each interaction $a \in \gamma$ the predicate

$$\kappa'_a = K_{\mathcal{L}_a}^{\tilde{\mathcal{R}}} \kappa_a$$

where $\mathcal{L}_a = \mathit{participants}(a) \cup \mathit{obs}(a)$.

According to Proposition 3.4, we have $\kappa'_a \implies \kappa_a$. Furthermore, since κ'_a is actually a knowledge predicate, it depends only on the state of components in $\mathcal{L}_a$. In other words, the actual set of observed component for each interaction a is exactly $\mathit{observed}_{\kappa'}(a) = \mathcal{L}_a \setminus \mathit{participants}(a)$. Since $\mathcal{L}_a$ is the disjoint union the participants in a and the components in $\mathit{obs}(a)$, we have $\mathit{observed}_{\kappa'}(a) = \mathit{obs}(a)$. Therefore, we assume in the sequel that $\mathit{observed}_{\kappa'}(a)$ is a parameter of κ'.

By restricting too much the sets $\mathit{observed}_{\kappa'}(a)$, one takes the risk of always obtaining $\kappa'_a = \mathit{false}$, as the observation might not be sufficient to ensure knowledge of κ_a. We define two criteria characterizing different detection levels, namely basic and complete.

Definition 5.14 (Detection levels)**.** Let $\kappa\gamma(B_1, \cdots, B_n)$ be a BIC model and κ' be the Condition obtained by restricting observation in κ. The obtained Condition κ' is:

- *basic* iff $\quad \forall q \in \tilde{\mathcal{R}} \quad \bigvee_{a\in\gamma} EN_a(q) \wedge \kappa_a(q) = \bigvee_{a\in\gamma} EN_a(q) \wedge \kappa'_a(q)$.
- *complete* iff for each interaction $a \in \gamma$: $\forall q \in \tilde{\mathcal{R}} \quad \kappa'_a(q) = \kappa_a(q)$.

Theorem 5.15 below, relates the detection levels and corresponding guarantees on the model equipped with the computed Condition κ'. Baseness ensures that κ' does not introduce deadlocks. Completeness ensures that the global behavior of the model equipped with κ and the model equipped with κ' are the same.

Theorem 5.15. *Let $\kappa\gamma(B_1, \cdots, B_n)$ be a BIC model and κ' be the Condition obtained by restricting observation in κ. Then, $\rightarrow_{\kappa'} \subseteq \rightarrow_\kappa$ and:*

1. *If κ' is basic, then $q \in Q$ is a deadlock for $\rightarrow_{\kappa'}$ only if q is a deadlock for $\rightarrow_\kappa$.*
2. *If κ' is complete, then $\rightarrow_{\kappa'} = \rightarrow_\kappa$.*

Proof. Since for each $a \in \gamma$, $\kappa'_a \implies \kappa_a$, we have $\rightarrow_{\kappa'} \subseteq \rightarrow_\kappa$.

1. By contraposition, let $q \in Q$ be a deadlock-free state for $\rightarrow_\kappa$, *i.e.* such that $\exists a \in \gamma \;\; EN_a(q) \wedge \kappa_a(q)$. Baseness ensures that $\bigvee_{a\in\gamma} EN_a(q) \wedge \kappa'_a$ holds and thus $\exists b \in \gamma$ such that $EN_b(q) \wedge \kappa'_b(q)$. Thus $q \xrightarrow{b}_{\kappa'}$ and q is a deadlock-free state for $\rightarrow_{\kappa'}$.

2. Assume that $q \xrightarrow{a}_\kappa q'$. Then $\kappa_a(q)$ holds and $q \xrightarrow{a}_\gamma q'$. Completeness ensures that $\kappa'_a(q)$ also holds. Thus $q \xrightarrow{a}_{\kappa'} q'$. □

These results characterize to what extent the original behavior can be captured through partial observation. However, they do not state how to choose components in order to ensure a given detection level. In the next subsection, we propose heuristics that minimize the number of observed atomic components, yet ensuring the required detection level.

5.4.2 Heuristics to Minimize Observed Components

In this Subsection, we propose for each detection level defined in 5.14 a heuristic that takes as input a BIC model and outputs minimized sets of observed components $\{observed_{\kappa'}(a)\}_{a\in\gamma}$. Each of these heuristics guarantees that the reduced Condition κ' built using the returned sets of observed components meets the corresponding detection level. The results depend upon the approximation $\tilde{\mathcal{R}}$ of the reachable states used for computing the knowledge. We assume throughout this subsection a fixed over approximation $\tilde{\mathcal{R}}$ of the reachable states.

Algorithm 1 Pseudo-code of Simulated Annealing

Input: An initial solution *init*, a `cost` function, an `alter` function.
Output: A solution with a minimized cost.

1: sol:=$init$
2: T:=T_{max}
3: **while** $T > T_{min}$ **do**
4: $sol' :=$ `alter`(sol)
5: $\Delta :=$ `cost`(sol') - `cost`(sol)
6: **if** $\Delta < 0$ or random() $< e^{\frac{-\Delta}{T}}$ **then**
7: sol:=sol'
8: **end if**
9: T:= $0.99 \times T$
10: **end while**
11: **return** sol

The proposed solution to the minimizing observation problem is based on simulated annealing meta heuristic [59]. A pseudo-code for the simulated annealing is shown in Algorithm 1. This heuristic allows searching for optimal solutions to arbitrary cost optimization problems. The search through the solution space is controlled by a *temperature* parameter T. At every iteration, temperature decreases slowly (line 9) and the current solution moves into a new, nearby solution still ensuring either baseness or completeness (line 4). If the new solution is better (i.e. observes fewer components), then it becomes the current solution. Otherwise, it may be accepted with a probability that decreases when (1) the temperature decreases or (2) the extra cost of the new solution increases (line 6). The idea is to temporarily allow a bad solution whose neighbors may be better than the current one. By the end of the process, the temperature is low, which prevents bad solutions from being accepted. Now, we provide initial solutions *init* as well as `alter` and `cost` functions that are used to ensure either completeness or baseness.

Ensuring Completeness

According to Definition 5.14, checking for completeness is performed interaction by interaction, Therefore, minimizing observation can be carried out independently for each interaction. Given an interaction a we are seeking for a minimal set of atomic compo-

nents $\mathcal{L}_a$ such that $K^{\tilde{\mathcal{R}}}_{\mathcal{L}_a}\kappa_a = \kappa_a$. Note that finding such a set $\mathcal{L}_a$ yields the corresponding set of components to observe by taking $observed_{\kappa'}(a) = \mathcal{L}_a \setminus participants(a)$.

Algorithm 2 Function `alter` for ensuring completeness

Input: A BIC component $\kappa\gamma(B_1, \ldots, B_n)$, an interaction a and a solution $\mathcal{L}_a$.
Output: A solution $\mathcal{L}'_a$ that is a neighbor of $\mathcal{L}_a$.

```
1:  L'_a:=L_a
2:  choose B_i in L'_a \ participants(a)
3:  L'_a:=L'_a \{B_i}                                   //perturbation
4:  while K^R̃_{L'_a} κ_a ≠ κ_a do
5:     choose B_i in {B_1, ..., B_n} \ L'_a
6:     L'_a:=L'_a ∪ {B_i}                               //completion
7:  end while
8:  if L'_a = participants(a) then
9:     return L'_a
10: end if
11: choose B'_i in L'_a \ participants(a)
12: while K^R̃_{L'_a\{B'_i}} κ_a = κ_a do
13:    L'_a:=L'_a \ {B'_i}                               //reduction
14:    if L'_a = participants(a) then
15:       return L'_a
16:    end if
17:    choose B'_i in L'_a \ participants(a)
18: end while
19: return L'_a
```

The initial solution is obtained by taking the set of atomic components that are needed to decide κ_a, that is $init_a = participants(a) \cup observed_\kappa(a)$. At each iteration of the simulated annealing, a new solution is computed using the `alter` function shown in Algorithm 2. First, one atomic component is removed from the solution (perturbation, line 3), possibly breaking completeness. Then, new atomic components are added randomly until the solution ensures complete detection again (completion, line 6). Finally, atomic components are removed randomly, provided they do not contribute to completeness (reduction, line 13).

After completion and during reduction steps, the completeness condition is checked (line 12). On termination, this ensures that the solution returned by the heuristic is complete. During this steps, if only participants in the interaction remain in $\mathcal{L}'_a$, then the solution is optimal because no observed components can be further removed. In that case, the set of participants is returned.

The cost of the solution is obtained by counting the number of atomic components in $observed_{\kappa'}(a) = \mathcal{L}_a \setminus participants(a)$. The cost function is thus $\texttt{cost}(\mathcal{L}_a) = |\mathcal{L}_a \setminus participants(a)|$.

During the completion step, the algorithm can choose to observe components that

were not originally observed. Indeed, for some models, observing components that are not in $observed_\kappa(a)$ gives more knowledge than those originally in this set.

Ensuring Baseness

Baseness is achieved if for every state where an interaction is allowed by κ, at least one interaction is also allowed in κ'. Baseness has to be ensured at a global level. On one hand, allowing an interaction to observe additional atomic components may extend the set of states where it knows that it can execute. On the other hand, reducing observation of the interaction, while restricting the set of possible executions, might not necessarily break the baseness. Therefore, a solution $\{\mathcal{L}_a\}_{a\in\gamma}$ to the minimizing observation ensuring baseness cannot be built independently for each interaction.

Algorithm 3 Function `alter` for ensuring basic detection of false conflicts

Input: A BIC component $\kappa\gamma(B_1,\ldots,B_n)$, a solution $\{\mathcal{L}_a\}_{a\in\gamma}$,
Output: A solution $\{\mathcal{L}'_a\}_{a\in\gamma}$ that is a neighbor of $\{\mathcal{L}_a\}_{a\in\gamma}$.

1: $\{\mathcal{L}'_a\}_{a\in\gamma}$:=$\{\mathcal{L}_a\}_{a\in\gamma}$
2: **choose** b **in** γ **and** B_i **in** $\mathcal{L}_b \setminus participants(b)$
3: $\mathcal{L}'_b$:=$\mathcal{L}'_b \setminus\{B_i\}$ *//perturbation*
4: **while** $\bigvee_{a\in\gamma} EN_a \wedge K^{\tilde{\mathcal{R}}}_{\mathcal{L}'_a}\kappa_a \neq \bigvee_{a\in\gamma} EN_a \wedge \kappa_a$ **do**
5: **choose** b **in** γ **and** B_i **in** $\{B_1,\ldots,B_n\}\setminus\mathcal{L}_b$
6: $\mathcal{L}'_b$:=$\mathcal{L}'_b \cup \{B_i\}$ *//completion*
7: **end while**
8: **if** $\bigwedge_{a\in\gamma}\mathcal{L}'_a = participants(a)$ **then**
9: **return** $\{\mathcal{L}'_a\}_{a\in\gamma}$
10: **end if**
11: **choose** b **in** γ **and** B_i **in** $\mathcal{L}_b \setminus participants(b)$
12: **while** $\bigvee_{a\neq b}\left(EN_a \wedge K^{\tilde{\mathcal{R}}}_{\mathcal{L}'_a}\kappa_a\right) \vee \left(EN_b \wedge K^{\tilde{\mathcal{R}}}_{\mathcal{L}'_b\setminus\{B_i\}}\kappa_b\right) = \bigvee_{a\in\gamma} EN_a \wedge \kappa_a$ **do**
13: $\mathcal{L}'_b$:=$\mathcal{L}'_b \setminus \{B_i\}$ *//reduction*
14: **if** $\bigwedge_{a\in\gamma}\mathcal{L}'_a = participants(a)$ **then**
15: **return** $\{\mathcal{L}'_a\}_{a\in\gamma}$
16: **end if**
17: **choose** b **in** γ **and** B_i **in** $\mathcal{L}_b \setminus participants(b)$
18: **end while**
19: **return** $\{\mathcal{L}'_a\}_{a\in\gamma}$

The initial solution assumes that each interaction a observes all atomic components that are needed to decide κ_a, that is $init_a$. Thus the initial solution is $init = \{init_a\}_{a\in\gamma}$. As for completeness, the `alter` function for baseness presented in Algorithm 3 computes a new solution based on the same three steps (perturbation,completion,reduction) being performed on a family of sets of observed atomic components, instead of a single set.

After completion and during reduction steps, the baseness condition is checked (line 12). This guarantees that the returned solution is basic. During reduction step, it can occur

that no observed components can be removed, that is, for each interaction a, $\mathcal{L}_a$ contains only the participants in a. In that case, the solution is directly returned.

Here the cost of the solution is the sum of the number of atomic components observed by each interaction. Thus, we define the `cost` function as $\texttt{cost}(\{\mathcal{L}_a\}_{a\in\gamma}) = \sum_{a\in\gamma} |\mathcal{L}_a \setminus \mathit{particpants}(a)|$.

The simulated annealing algorithm is evaluated in Section 8.3. We try to reduce the set of observed for two different examples. We compare the costs of the best solution found when using linear invariants and boolean invariants, for both basic and complete implementations.

6 Decentralizing the Engine

In the previous chapter, we defined a distributed model relying on a single centralized engine for executing all interactions. Such distributed models allow parallelism between computations in the components. However, concurrency between interactions is not possible since they are executed within the same distributed component.

This Chapter provides methods for decentralizing the engine into a set of concurrent decentralized engines. Each decentralized engine is responsible for executing a given subset of the interactions. The decentralization is therefore parameterized by a partition of the interactions, each class of the partition corresponding to a separate engine.

Partitioning the interactions into several engines introduces conflicts between engines, as explained in Section 6.1. These conflicts are caused either by the Interaction or by the Condition layers. In the first case, conflicts happen when a component can participate in two different interactions. In the second case, conflicts happen when interaction a observes a component involved in interaction b.

A solution to avoid introducing conflicts between engines consists in grouping conflicting interactions in the same engines. This solution leads to a conflict-free partition. The distributed model obtained from a conflict-free partition is sketched in Section 6.2. Engines in this distributed model are very similar to the centralized engine.

As considering only conflict-free partitions restricts significantly the design choices, we provide a solution that handles any partition in Section 6.3. In order to solve conflicts between engines, this construction incorporate a conflict resolution protocol. We propose three different conflict resolution protocols from Bagrodia [6], that rely on counters. We prove correctness of the distributed model embedding the centralized version of the conflict resolution protocol through observational equivalence. The correctness of distributed models embedding other protocols is proven through trace equivalence.

Finally, we present another conflict resolution protocol, named α-core [73], in the Section 6.4. The protocol is described by two generic automata, one implementing a component, called participant, and one implementing a coordinator that is responsible for one interaction. We quickly present how this protocol is implemented in a distributed BIP model.

The α-core protocol avoids using counters but requires a handshake mechanism to cancel each unused offer. We present in Section 6.5 an optimization for this protocol, that reduces the number of messages. This optimization can be applied as well to the 3-layer model, although it is not described in this section.

Finally, in Section 6.6, we discuss the different approaches presented in the chapter. In particular, we try to see how the interesting points of each solution can be applied to the other ones.

6.1 Conflicts

In general, there is a conflict between two entities whenever there are competing to use a given resource. When decentralizing the engine, the resources being used are the offers sent by the components.

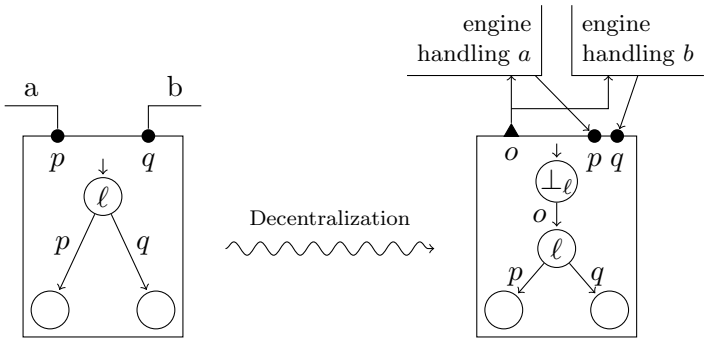

Figure 6.1: An interaction conflict between a and b.

Consider the simple fragment of model presented on the left of Figure 6.1. Whenever the component reaches the control state ℓ, it can either execute the transition labeled by p or the one labeled by q but not both.

When decentralizing the model, assuming that interactions a and b are handled by separate engines, we obtain the fragment on the right of Figure 6.1. In the decentralized case, the component first sends an offer during the transition from $\perp_\ell$ to ℓ. As the next interaction involving the component can either be a through port p or b through port q, the offer is sent to both the engine handling a and the engine handling b. At this point, a conflict arises as only one of the engines should respond to the offer, since the component can execute only one of the transitions outgoing from ℓ.

In the previous example, the conflict is said to be an *interaction* conflict as it comes from the Interaction layer. This is a symmetric conflict. The Condition layer introduces some asymmetric conflicts.

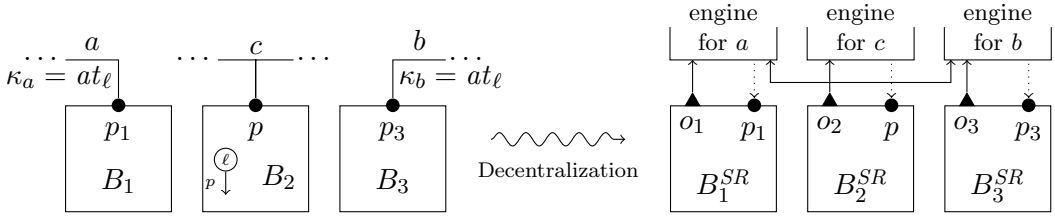

Figure 6.2: Condition conflicts and non-conflicts.

On the left of Figure 6.2, we represented a fragment of a model with Condition. In that model, the component B_2 is observed by a and b. Furthermore, B_2 is a participant in c.

When decentralizing this fragment, one obtains the fragment depicted on the right of Figure 6.2. Since the component B_2 is observed by a and b, it has to send its offer to the engines handling a and b. The engine handling c can execute it directly (provided there is no other conflicts). However, the engine handling a has to ensure that the Condition predicate κ_a holds. In particular, if c is executed, the component B_2 will move, possibly falsifying the predicate κ_a. This is an asymmetric conflict between a and c: c can execute without further check whereas a has to check that the component B_2 did not move.

On the other hand, two interactions observing the same component are not conflicting. Indeed, the execution of b in our example has no effect on the fact than a can execute, although both a and b observe the component B_2.

We define formally the Interaction and Condition conflict in the centralized model.

Definition 6.1 (Conflict states)**.** Let $B = \kappa\gamma(B_1, \ldots, B_n)$ be a BIC model and $a, b \in \gamma$ be two interactions.

- a and b are Interaction conflicting if
  - $\mathit{participants}(a) \cap \mathit{participants}(b) \neq \emptyset$, and
  - there is at least a component B_i in the intersection containing a control location with an outgoing transition labeled by a port of a and an outgoing transition labeled by a port of b.

  We denote this situation by $a \# b$ or equivalently $b \# a$.
- a is Condition conflicting with respect to b if
  - $\mathit{observed}_\kappa(a) \cap \mathit{participants}(b) \neq \emptyset$,
  - there is a component B_i in the intersection that can execute a transition labeled by a port of b.

  We denote this situation by $a \blacktriangleleft b$. Note that this relation is asymmetric.

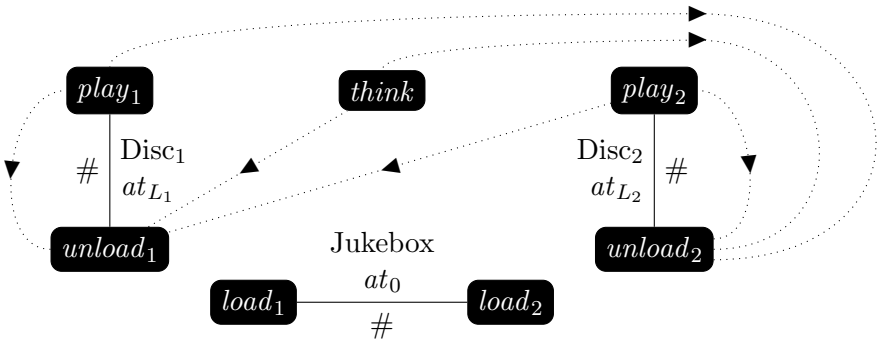

Figure 6.3: Conflict graph of the example from Figure 5.1.

The conflict graph of the example from Figure 5.1

In general, we say that the interactions a and b are *conflicting* if either $a \# b$, $a \blacktriangleleft b$ or $b \blacktriangleleft a$. Interaction that are not conflicting do not require a conflict resolution mechanism.

Definition 6.2 (Conflict-free interactions)**.** Let $B = \kappa\gamma(B_1, \ldots, B_n)$ be a BIC model. Two interactions a and b in γ are *conflict-free* if neither $a \# b$, $a \blacktriangleleft b$ nor $b \blacktriangleleft a$.

Two disjoint subsets γ_1, γ_2 of γ are *conflict-free* if for all couples (a, b) such that $a \in \gamma_1$ and $b \in \gamma_2$, a and b are conflict-free.

Although we defined conflicts in the centralized model, they only have a meaning because of the decentralized model. Conflicts between interactions handled by the same engine are solved locally by the engine, only conflicts between interactions from distinct engines requires an additional resolution mechanism.

6.2 Conflict-Free Partitioning

In the centralized engine from Chapter 5, conflicts between interactions appear as conflicts between Petri net transitions. Interactions that are conflicting correspond to transitions that share a common input place. The conflict is solved as the Petri net semantics ensures that if any one of these transitions executes, it consumes the token in the common input place, thus disabling the other transition. In particular, the semantics allows conflicts occurring between two interactions to be resolved locally, within the same engine.

A first solution for decentralizing the engine is to keep conflicts inside engines, and separate only conflict-free interactions. This solution forbids to have two conflicting interactions handled by different engines. In other words, it requires that the sets of interactions handled by the different engines are pairwise conflict-free. A version of this solution, not handling priorities nor Condition has been presented in [24].

Let $B = \kappa\gamma(B_1, \ldots, B_n)$ be a BIC model. Recall that a set of subsets $\gamma_1, \ldots, \gamma_k$ of γ, is a partition of γ if:

- $\gamma = \gamma_1 \cup \ldots \cup \gamma_k$ and
- $\forall i, j \in \{1, \ldots, k\}, i \neq j \implies \gamma_i \cap \gamma_j = \emptyset$

Furthermore, $\gamma_1, \ldots, \gamma_j$ is a conflict-free partition if the γ_i are pairwise conflict-free, that is if for any two distinct integers i, j in $\{1, \ldots, k\}$, γ_i and γ_j are conflict-free.

Assume that a separate engine is built for each class of a conflict-free partition. Let B_i^{SR} be a distributed component, sending an offer indicating which ports are enabled from local state $q_i = (\ell_i, v_i)$. Following the Definition 6.1, all interactions that contains a port enabled from state q_i, or that can observe the component B_i are conflicting. Since the partition is conflict-free, all these interactions are handled by the same engine. Therefore, whenever the distributed component is at state $(\bot_{\ell_i}, v_i)$, it can choose to which engine send the offer, based on v_i.

We do not detail here how distributed atomic component from Definition 5.3 are modified to handle this case. However, we present a conflict-free decentralized version of the Jukebox example from Figure 5.1.

First, we need to identify the conflicts between interactions in the Jukebox example and build a conflict-free partition. Interactions $load_1$ and $load_2$ are Interaction conflicting because at state 0, the Jukebox atomic component can execute both of them.

Interactions $unload_i$ and $play_i$ are Interaction conflicting because at state L_i, the Disc_i atomic component can execute both of them. Furthermore, the Condition predicate of $load_i$ need to observe the state of the Listener component. Consequently, $unload_1$ is Condition conflicting with respect to $play_2, think$ and $unload_2$ is Condition conflicting with respect to $play_1, think$. To obtain a conflict-free partition, the $load_i$ interaction cannot be separated, neither can the interactions $play_i$ and $unload_i$ be. Thus we choose the following partition: $\gamma_1 = \{load_i\}_{i=1,2}$, $\gamma_2 = \{unload_i, play_i\}_{i=1,2} \cup \{think\}$.

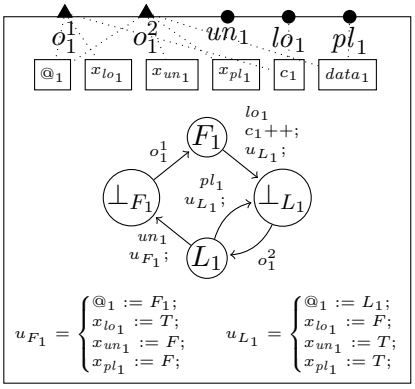

Figure 6.4: Distributed version of the component Disc_1 from Figure 5.1 for communicating with conflict-free engines.

We present the distributed component Disc_1^{SR} in Figure 6.4. When reaching the stable control location F_1, Disc_1^{SR} sends an offer that can only be used by the engine E_1 handling γ_1. There is a dedicated offer port o_1^1 for this engine. Similarly, when reaching the control location L_1, Disc_1^{SR} sends through the port o_1^2 an offer that can only be used by the engine E_2 handling γ_2. The modification consists mainly in creating a separate offer port for each engine that need to receive offers from the component, and relabel the offer transitions accordingly.

We present in Figure 6.5 the Petri net of the engine E_2 handling the interactions in γ_2. This engine is very similar to the centralized engine, with the exception that it executes only interactions from γ_2. The name of the offer receive ports have been changed to match the engine name (E_2). Send ports do not require distinctive labels, as conflict-freedom imposes that only one engine can notify a given component on a given port.

The global conflict-free distributed model is presented in Figure 6.6. For the sake of readability, the superscripts on engines ports labels have been dropped. This version contains three additional Send/Receive interactions, as three of the components may send an offer to both engine. Since the component B_4 in not observed nor participant in any interaction of E_1, it sends only offers to E_2.

This decentralization method is constrained by the Interaction and Condition conflicts of the model. In particular, if there a chain of conflicts between any two interactions of the model, one ends with the centralized engine solution. In order to decentralize the

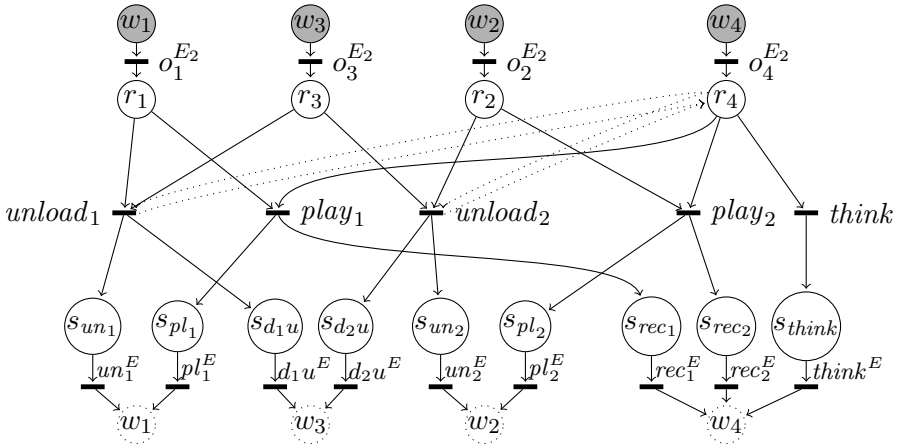

Figure 6.5: Engine E_2 handling the class γ_2 of the conflict-free partition for the example from Figure 5.1.

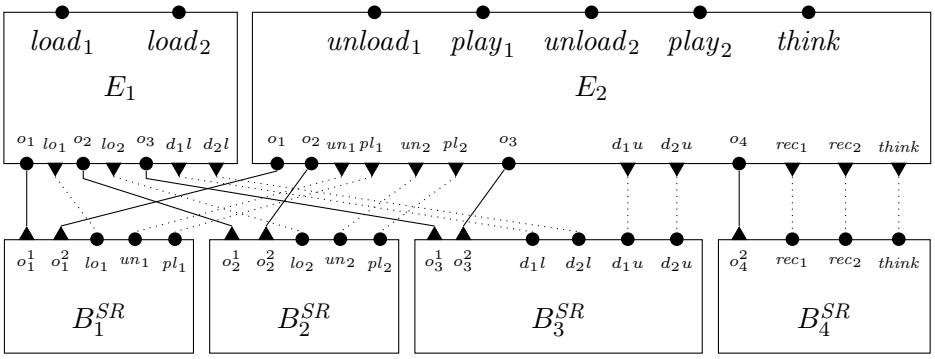

Figure 6.6: Global view of the conflict-free distributed model for the example from Figure 5.1.

engine according to an arbitrary partition, one needs a conflict resolution protocol.

6.3 3-Layer Send/Receive BIP

The 3-layer BIP model contains three kinds of distributed components:

- the distributed atomic components, obtained by transforming the atomic components of the original model,
- the distributed engines and
- the conflict resolution protocol components.

These three kinds of components correspond to the three layers of the model. The distributed atomic components and distributed engines are slightly modified with respect to the centralized and conflict-free versions. Communication between these two layers consists of offers and notifications.

In this new model, interactions that are conflicting only with local interactions are executed through Petri net transitions, as in the centralized version. For executing interactions that are conflicting with interactions from other engines, a *reservation* request is sent to the conflict resolution protocol. The latter either grants or deny the execution, depending on whether conflicting interactions have already executed or not.

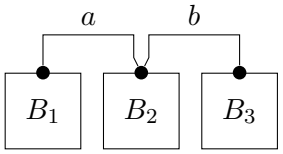

Figure 6.7: An example with conflicting interactions.

We shall sketch the conflict resolution mechanism before describing the details. Our solution relies on Bagrodia's protocols, described in Subsection 2.3.1. As said at the beginning of this chapter, the conflicting resources between interactions are the offers from atomic components. Conflict resolution requires that each offer sent by a component is used only once. By numbering the successive offers sent by the component, that is by using a counter, this problem falls back to ensuring that each offer number is used only once.

To illustrate the principle of conflict resolution based on offer numbers, we consider the example depicted in Figure 6.7. We assume one engine E_a, handling a, and one engine E_b, handling b. The resulting three layer model is sketched in Figure 6.8.

In this model, each atomic component B_i numbers the offers (starting from 1) and stores this values in variable n_i. This value is sent with the offer, and stored in the engine as well.

Whenever an engine detects an enabled interaction, as a in our case, it tries to execute it. Since a is conflicting with b, handled by another engine, E_a cannot directly execute a.

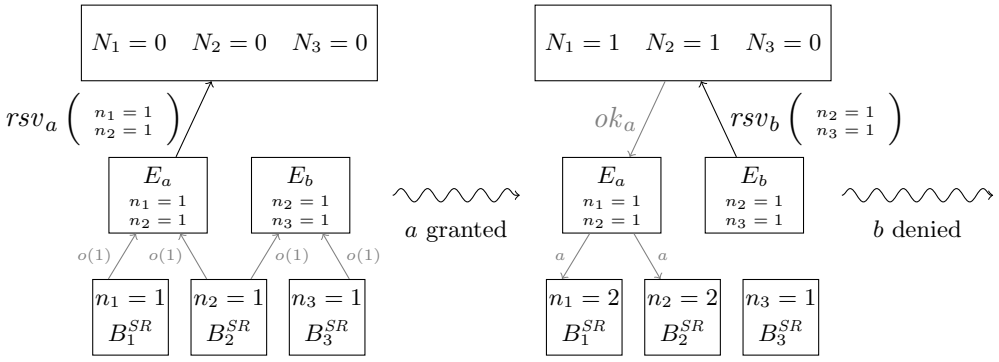

Figure 6.8: Principle of conflict resolution based on offer numbers.

Instead, it calls the conflict resolution protocol through a *reservation* rsv_a that contains the offer numbers for which a has been detected enabled. The model on left of Figure 6.8 depicts the global state just before the reservation is sent.

For each component B_i, the conflict resolution protocol maintains the last offer number used in a variable N_i. On the left of Figure 6.8, no interaction have been granted yet and the values of all N_i variables is 0. Since all offer numbers n_i in the reservation from a are greater than the corresponding last used offer number N_i, execution is granted. To prevent conflicting interactions from reusing the current offers from B_1^{SR} and B_2^{SR}, the conflict resolution protocol updates the last used offer numbers N_1 to the value of n_1 and N_2 to the value of n_2.

On the right of Figure 6.8, a has executed and its participants have incremented their offer number. From the point of view of E_b, the interaction b is enabled. Therefore E_b also sends a reservation to the conflict resolution protocol. The offer number n_2 in the reservation is 1, which is not greater than the last offer number N_2 used for B_2^{SR}. The conflict resolution protocol thus denies execution of b by sending a $fail_b$ message (not shown on the figure).

The above example shows an example of Interaction conflict resolution. It is the original protocol from Bagrodia. The case of Condition conflicts needs a small modification. Let a be an interaction. Components in $participants(a)$ move after executing a, therefore the offer that was sent to execute a is not valid anymore. The offer sent by a component in $observed_\kappa(a)$ remains valid after a has executed, since the component did not move. However, executing a requires to check that the offer sent by the observed component is still valid. To this end, the reservation for a also includes the offer numbers from the observed components. The interaction is granted only if for all components, the offer number from the reservation is greater than the last offer number used. Only last used offer variables of participants are updated, offers from observed components may be reused.

We now detail the construction of the three layer model. The input parameters are a BIC model $\kappa\gamma(B_1, \ldots, B_n)$ and a partition $\gamma_1, \ldots, \gamma_k$ of the interactions. To illustrate the construction of the Send/Receive model, we consider the example from Figure 5.1,

with the partition $\{load_1, unload_1\}, \{load_2, unload_2\}, \{play_1, play_2, think\}$.

6.3.1 Distributed Atomic Components

As in Definition 5.3, we obtain the distributed version of the atomic components by breaking the atomicity of the transitions. This definition differs slightly from Definition 5.3. Namely, we add the offer count variable and a separate port o^j for each class γ_j of the partition. This port is used to send offers to the corresponding engine E_j.

Definition 6.3. Let $B = (L,\ P,\ T,\ X,\ \{X_p\}_{p\in P},\ \{g_\tau\}_{\tau\in T},\ \{f_\tau\}_{\tau\in T})$ be an atomic component, and $X^\kappa \subseteq X$ be a set of variables needed by the Condition layer. The corresponding Send/Receive atomic component is $B^{SR} = (L^{SR},\ P^{SR},\ T^{SR},\ X^{SR},\ \{X_p^{SR}\}_{p\in P},\ \{g_\tau\}_{\tau\in T^{SR}},\ \{f_\tau\}_{\tau\in T^{SR}})$, such that:

- $L^{SR} = L \cup L^\perp$, where $L^\perp = \{\perp_\ell^j \,|j \in \{1,\dots,k\}\ \ \ell \in L\}$.
- $X^{SR} = X \cup \{x_p\}_{p\in P} \cup \{@\} \cup \{n\}$, where each x_p is a new Boolean variable, @ is a control location variable and n is the offer count variable.
- $P^{SR} = P \ \cup\ \{o^1,\dots,o^k\}$, where each *offer* port o^j exports the variables $X^{SR}_{o^j} = \{n, @\} \cup \bigcup_{p\in P}(\{x_p\}\cup X_p)\cup X^\kappa$, that is the offer count variable, the state variable, the new Boolean variables as well as the exported variables associated to each port, and the variables needed for the Condition layer. For all other ports $p \in P$, we keep $X_p^{SR} = X_p$.
- For each location $\ell \in L$, we include the offer transitions $\{(\perp_\ell^1, o^1, \perp_\ell^2),\ (\perp_\ell^2, o^2, \perp_\ell^3),\ \dots (\perp_\ell^k, o^k, \ell)\}$ in T^{SR}. Each of these transitions has *true* as guard and the identity function as the update function.
- For each transition $\tau = (\ell, p, \ell') \in T$, we include a response transition $\tau_p = (\ell, p, \perp_{\ell'}$) in T^{SR}. The guard g_{τ_p} is true. The function f_{τ_p} first applies the original update function f_τ, increments the offer count variable n, then sets the state variables to the next control location (i.e. $@ := \ell'$) and finally updates the Boolean variables:

$$\text{for all } r \in P \quad x_r := \begin{cases} g_{\tau'} & \text{if } \exists \tau' = (\ell', r, \ell'') \in T \\ \mathit{false} & \text{otherwise} \end{cases}$$

The distributed version of the Disc_1 component from the example in Figure 5.1 is depicted in Figure 6.9. The input partition has 3 classes, therefore there are three offer ports. Before reaching a stable state, each offer port has to send an offer to the corresponding engine. Whenever the distributed component reaches a stable state, all offer have been sent.

This version sends all the port variables, all data exported by at least one port and all data needed to evaluate Condition predicate to each engine. An optimization would be to send an offer to an engine only if the offer contains a port involved in the interaction of the engine, or if the component is observed by an interaction of the engine. Similarly, one could send data only if the port exporting them is enabled.

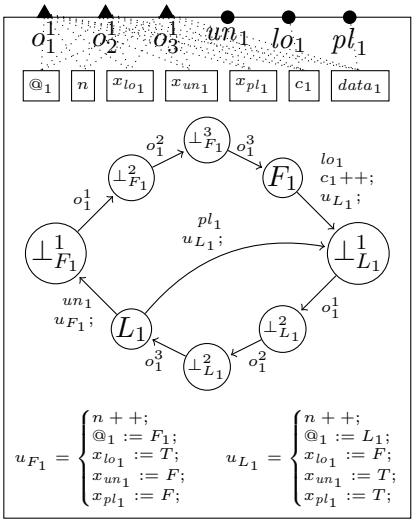

Figure 6.9: Distributed version of the Disc_1 component.

6.3.2 Engines

For each class γ_j of the partition, one builds an engine E_j handling interactions γ_j. An interaction of γ_j is *externally conflicting* if it conflicts with at least one interaction that is not in γ_j, otherwise it is *internally conflicting*. An internally conflicting interaction does not require to call the conflict resolution protocol as conflicts can be solved locally. Such interactions are executed through a unary interaction, in a similar way as in the centralized and conflict-free cases.

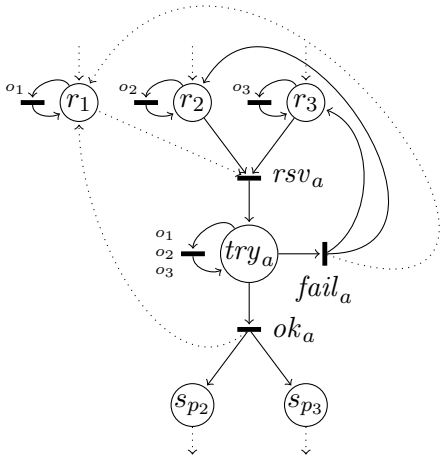

Figure 6.10: Reservation mechanism for interaction a involving ports p_2 and p_3 from components B_2 and B_3 and observing component B_1.

For externally conflicting interactions, we add a reservation mechanism. Figure 6.10

presents the skeleton of such a mechanism for an interaction a whose participants are components B_2 and p_3, and that observes component B_1. On the Figure, only transitions between receive places and sending places have been represented ; other transitions (receive offer and send notifications) are exactly the same as previously. Whenever offers from B_1, B_2 and B_3 have been received, the corresponding receive places r_1, r_2 and r_3 contain a token. The transition rsv_a is then possible if both the interaction guard G_a and as the Condition predicate κ_a evaluate to true. This transition does not modify the variables, it only removes tokens from the receive places to prevent internally conflicting interactions to execute. Execution of the transition rsv_a triggers the emission of a reservation request to the conflict resolution protocol. The latter either grants the execution, by answering on the port ok_a, or denies it, by answering on the port $fail_a$ In the first case, the data transfer function F_a is executed and the tokens corresponding to participants are put in appropriate sending places. In the second case, the tokens are put back in the receive places.

With this new construction, each offer is sent to all engines, and only one of the engines actually executes an interaction based on that offer. After execution of the interaction, each of its participants sends a new offer. From the point of view of all engines but the one that executed the interaction, this corresponds to receiving two successive offers from the participants. On Figure 6.10, offer transitions from the waiting places are not represented, but from each receive place, an additional loop transition allows two offers to be received successively. Furthermore, offers may be received whenever the engine is waiting for an answer from the conflict resolution protocol. Indeed, for each offer o_i incoming from a component B_i involved in the interaction a, there is a loop transitions from try_a labeled by o_i.

Definition 6.4. Let $B = \kappa\gamma(B_1, \ldots, B_n)$ be a BIC model, and $\gamma_j \subset \gamma$ a subset of the interactions. The centralized engine for this model is defined as the Send/Receive BIP component $E = (L^{E_j}, P^{E_j}, T^{E_j}, X^{E_j}, \{X_p\}_{p \in P^{E_j}}, \{g_\tau\}_{\tau \in T^{E_j}}, \{f_\tau\}_{\tau \in T^{E_j}})$

- The set L^{E_j} of places is the union of the *waiting places* $\{w_i \mid i \in \{1, \ldots, n\}\}$, the *receive places* $\{r_i \mid i \in \{1, \ldots, n\}\}$, the *sending places* $\{s_p \mid p \in P\}$ and the *trying places* $\{try_a \mid a \in \gamma_j,\ a$ externally conflicting$\}$.
- The set X^{E_j} of variables contains a copy of each variable exported by an offer port $\{x^{E_j} \mid x \in \bigcup_{i \in \{1,\ldots,n\}} X^{SR}_{o_i}\}$.
- The set P^{E_j} of ports contains:
  - the offer receive ports $\{o_i^{E_j} \mid i \in \{1, \ldots, n\}\}$, with variables $X_{o_i^{E_j}} = \{x^{E_j} \mid x \in X^{SR}_{o_i}\}$ attached,
  - the notification send ports $\{p^{E_j} \mid p \in P\}$, with variables $X_{p^{E_j}} = \{x^{E_j} \mid x \in X_p\}$ attached,
  - the unary port a, with no attached variables for each interaction $a \in \gamma_j$ that is internally conflicting,

– the send port $rsv_a^{E_j}$ and the receive ports $\{ok_a^{E_j}, fail_a^{E_j}\}$ for each interaction a that is externally conflicting. The ports $ok_a^{E_j}$ and $fail_a^{E_j}$ do not have any variable attached. The port $rsv_a^{E_j}$ has the variables $\{n_i^{E_j} \mid B_i \in invl_\kappa(a)\}$ attached.

- The set T^{E_j} of transitions consists of the following:
  – *receive offer*: for each $i \in \{1, \ldots, n\}$, T^{E_j} contains the transition $(w_i, o_i^{E_j}, r_i)$ and the transition $(r_i, o_i^{E_j}, r_i)$. For each externally conflicting interaction a, T^{E_j} contains the transitions $\{(try_a, o_i^{E_j}, try_a) \mid B_i \in invl_\kappa(a)\}$. These transitions have no guard nor update function.
  – *send notification*: for each port $p \in P$, let i be the index of the component exporting p, i.e. such that $p \in P_i$, T^{E_j} contains the transition (s_p, p^{E_j}, w_i). This transition has no guard nor update function.
  – For each interaction $a \in \gamma$, we define the guard

$$G^{E_j}(a) = \bigwedge_{p \in a} x_p^{E_j} \wedge G_a^{E_j} \wedge \kappa_a^{E_j}$$

where the predicate ϕ^{E_j} is obtained by replacing each occurrence of a variable x in the predicate ϕ by its local copy x^{E_j}, and each occurrence of a at_ℓ predicate by the test $@_i^{E_j} = \ell$, where i is the index such that $\ell \in L_i$. The update function $F^{E_j}(a)$ is obtained by replacing each occurrence of a variable x in the data transfer function F_a by its local copy x^{E_j}.

If a is internally conflicting, T^{E_j} contains the transition $\tau_a = (\{r_i | B_i \in invl_\kappa(a)\}, a, \{s_p \mid p \in P_a\} \cup \{r_i | B_i \in observed_\kappa(a)\})$. The guard and update function are $g_{\tau_a} = G^{E_J}(a)$ and $f_{\tau_a} = F^{E_J}(a)$.

If a is externally conflicting, T^{E_j} contains the following transitions:

* $\tau_{r_a} = (\{r_i | B_i \in invl_\kappa(a)\}, rsv_a^{E_j}, \{try_a\})$, guarded by $G^{E_j}(a)$,
* $\tau_{ok_a^{E_j}} = (\{try_a\}, ok_a^{E_j}, \{s_p \mid p \in P_a\} \cup \{r_i | B_i \in observed_\kappa(a)\})$, with update function $F^{E_J}(a)$.
* $\tau_{f_a} = (\{try_a\}, fail_a^{E_j}, \{r_i | B_i \in invl_\kappa(a)\})$, with no guard nor update function.

As an example, consider the engine handling the partition class $\{load_1, unload_1\}$, depicted in Figure 6.11. On the figure, there are only places of the connected component that contains transitions corresponding to reservation mechanism. Furthermore, the superscripts on the port names have been dropped. Each reservation transition is possible only if the interaction is enabled and the Condition predicates evaluates to true. The port rsv_{load_1} exports only the offer counts $n_1^{E_j}, n_3^{E_j}$ of the atomic components participant in $load_1$. The responsibility of the conflict resolution protocol is only to check whether the offer counts are still valid. Only the engine checks whether the interaction can be executed, based on the partial view of the system obtained through these offers.

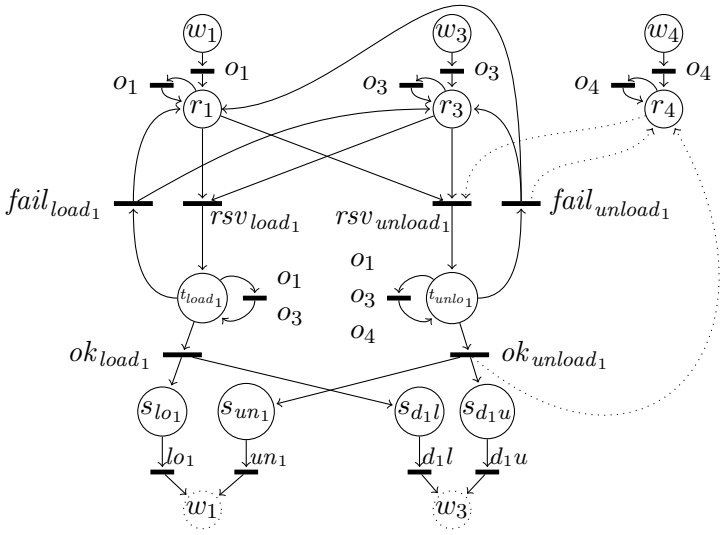

Figure 6.11: The distributed engine handling the class $\{load_1, unload_1\}$ of the partition.

6.3.3 Conflict Resolution Protocol

In [6], Bagrodia provides three different implementations for the conflict resolution protocol. The global principle is to keep the last offer number used for each component. Whenever a reservation arrives from of the distributed engines, each offer number from the request is compared against the value of the last offer number used in the conflict resolution protocol. If each number from the request is greater than the corresponding one in the conflict resolution protocol, the interaction is granted to execute. Otherwise execution is forbidden.

Centralized Protocol

The first version implements this behavior in a single process. The BIP version is a single component denoted CP.

Definition 6.5. Given a BIC model $\kappa\gamma(B_1, \ldots, B_n)$ and a partition $\gamma_1, \ldots, \gamma_k$ of the interactions, the centralized reservation protocol is the component $CP = (L^{CP}, P^{CP}, T^{CP}, X^{CP}, \{X_p\}_{p \in P^{CP}}, \{g_\tau\}_{\tau \in T^{CP}}, \{f_\tau\}_{\tau \in T^{CP}})$, where:

- X^{CP} contains the last used offer variable N_i for each component B_i.
- For each externally conflicting interaction a:
  - L^{CP} contains the waiting place w_a and the receive place r_a.
  - P^{CP} contains the ports rsv_a, ok_a and $fail_a$.
  - X^{CP} contains the variables $\{n_i^a \mid B_i \in invl_\kappa(a)\}$. The variables associated to the port rsv_a are $X_{rsv_a} = \{n_i^a \mid B_i \in invl_\kappa(a)\}$. The ports ok_a and $fail_a$ do not have associated variables.

– T^{CP} contains the transitions $\tau_{rsv_a} = (w_a, rsv_a, r_a)$, $\tau_{ok_a} = (r_a, ok_a, w_a)$ and $\tau_{fail_a} = (r_a, fail_a, w_a)$. The transition τ_{rsv_a} has no guard and no update function. The transition τ_{ok_a} is guarded by $G_{\tau_{ok_a}} = \bigwedge_{B_i \in invl_\kappa(a)} n_i^a > N_i$ and its update function updates the N_i variables of the participants: `foreach` $B_i \in participants(a)$ `do` $N_i := n_i^a$. The transition τ_{fail_a} has no guard and no update function.

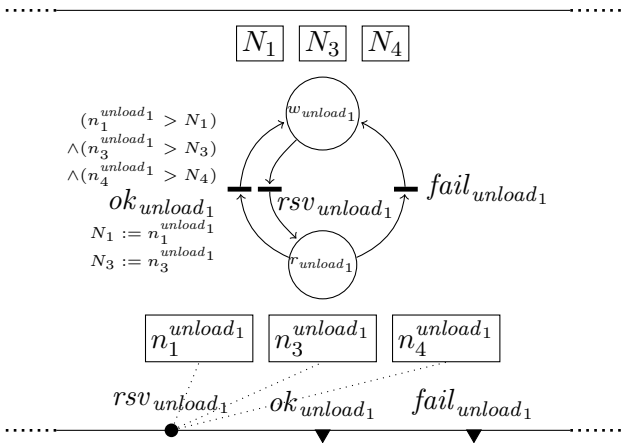

Figure 6.12: Fragment of the centralized conflict resolution protocol for handling $unload_1$.

The Figure 6.12 presents the places, transitions, variables, guards and update functions involved in handling the interaction $unload_1$. Initially, there is a token in the place w_{unload_1}. Whenever a reservation for executing $unload_1$ arrives, this token moves to the place r_{unload_1}. From this place, if the guard of the transition labeled by ok_{unload_1} is true according to the current values of N_i variable and freshly received $n_i^{unload_1}$ variables, the transition can take place. The transition labeled by $fail_{unload_1}$ is always possible. If two reservation requests for executing conflicting interactions are simultaneously received, one of the two *ok* labeled transition will be selected and executed, according to the Petri net semantics. The unselected *ok* transition will then become disabled, leaving only the *fail* transition to reach back the waiting state.

In general, the correctness of the protocol is ensured by the atomic access to the N_i variables. Here, atomicity is achieved through the semantics of atomic component's execution. The two remaining protocols are actually protocols to ensure atomicity for accessing the N_i values.

Token Ring Protocol

The next version of the conflict resolution protocol is inspired by the token-based algorithm due to Bagrodia [6]. The token ring protocol includes one component for each externally conflicting interaction. A token circulates between all the components. Atom-

icity of access to the N_i variables is ensured as only the component holding the token can modify these variables.

Definition 6.6. Given a BIC model $\kappa\gamma(B_1, \ldots, B_n)$ and a partition $\gamma_1, \ldots, \gamma_k$ of the interactions, the component handling reservations for interaction a is $TR_a = (L^{TR_a}, P^{TR_a}, T^{TR_a}, X^{TR_a}, \{X_p\}_{p \in P^{TR_a}}, \{g_\tau\}_{\tau \in T^{TR_a}}, \{f_\tau\}_{\tau \in T^{TR_a}})$, where:

- L^{TR_a} contains the waiting place w_a, the receive place r_a, the token place t_a and the send token place st_a.
- P^{TR_a} contains the ports rsv_a, ok_a and $fail_a$ as well as the ports RT_a, ST_a.
- X^{TR_a} contains the variables $\{n_i^a \mid B_i \in invl_\kappa(a)\}$, and a variable N_i for each component B_i. The variables associated to the port rsv_a are $X_{rsv_a} = \{n_i^a \mid B_i \in invl_\kappa(a)\}$. The variables associated to ports ST_a and RT_a are $\{N_1, \ldots, N_n\}$. The ports ok_a and $fail_a$ do not have associated variables.
- T^{TR_a} contains the transitions:
  - $\tau_{rsv_a} = (w_a, rsv_a, r_a)$, with no guard and no update function.
  - $\tau_{ok_a} = (t_a, ok_a, st_a)$ guarded by $G_{\tau_{ok_a}} = \bigwedge_{B_i \in invl_\kappa(a)} n_i^a > N_i$, with `foreach` $B_i \in participants(a)$ `do` $N_i := n_i^a$ as update function.
  - $\tau^1_{fail_a} = (r_a, fail_a, w_a)$ and $\tau^2_{fail_a} = (t_a, fail_a, st_a)$, guarded by $\neg G_{\tau_{ok_a}}$ with no update function.
  - $\tau_{ST_a} = (st_a, ST_a, w_a)$, with no guard and no update function.
  - $\tau^1_{RT_a} = (w_a, RT_a, st_a)$ and $\tau^2_{RT_a} = (r_a, RT_a, t_a)$, with no guard and no update function.

Figure 6.13 presents the component that handles interaction $unload_1$ in the conflict resolution protocol. Initially, the component is at state w_{unload_1}. From that state it can either receive the token RT or receive a reservation request rsv. Receiving the token updates the N_i variables, the only next possible action is to propagate the token RT. Upon reception of a reservation request (rsv), if the current values of N_i variables already discard the request, and a *fail* message is sent back. Otherwise, the component waits to receive the token and takes the decision to grant or deny the execution based on the latest N_i values.

We assume that externally conflicting interactions are numbered, i.e. they are written $\{a_1, \ldots, a_m\}$. The token ring protocol is obtained by building one component TR_{a_j} for each externally conflicting interaction. These components are connected by the set γ^{TR} of Send/Receive interactions between the following couples of ports (ST_{a_1}, RT_{a_2}), (ST_{a_2}, RT_{a_3}), $\ldots$, $(ST_{a_{m-1}}, RT_{a_m})$, (ST_{a_m}, RT_{a_1}).

Dining Philosophers Protocol

A more decentralized conflict resolution protocol is obtained by embedding a distributed solution to the dining philosophers problem, as the one provided by Chandy and Misra

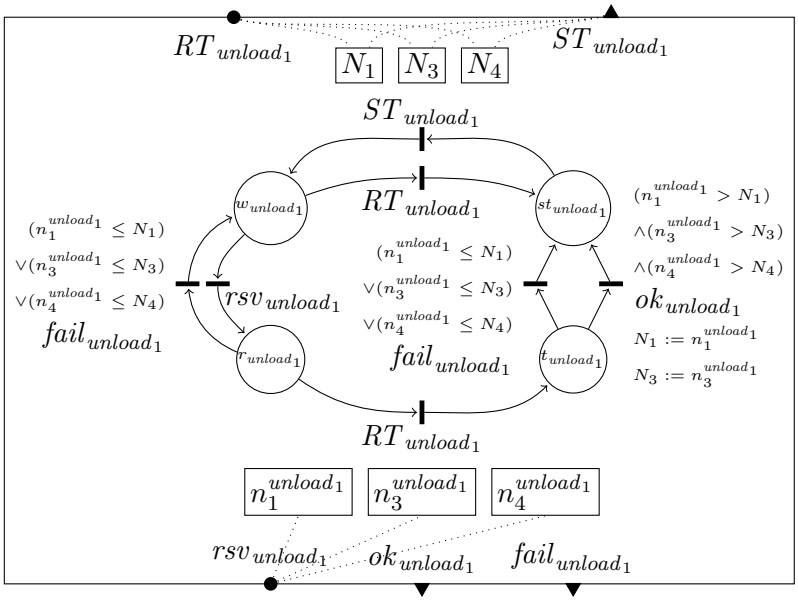

Figure 6.13: Component handling reservation for $unload_1$ in the token ring conflict resolution protocol.

in [32]. In this conflict resolution protocol, there is one atomic component DP_a for each externally conflicting interaction a. If the interactions a and b are in external conflict, the two components DP_a and DP_b share a fork carrying the N_i variables for components involved in both interactions. In order to ensure atomic access to the N_i variables, each component must obtain all the forks before granting execution.

Initially, the component DP_a is at state w_a. From that state, it can receive a rsv_a message indicating that the interaction wants to execute. Upon reception of this message, tokens are put in w_{f_b} places, to start negotiating the fork with each component DP_b corresponding to an interaction b in external conflict with a. The negotiation terminates when all forks have been obtained. At this point, the component DP_a is the only one able to modify the N_i variables of components involved in a and can therefore take the decision to grant or deny execution. Transition ok_a and $fail_a$ are enabled from that state and manipulate the N_i variables as in the previous cases.

Given two interactions a and b, we denote by $conf_\kappa(a,b) = invl_\kappa(a) \cap invl_\kappa(b)$ the set of components that are involved in both interactions. We denote by $extconf_\kappa(a)$ the set of interactions b that are in external conflict with a.

Negotiation of forks between components of the dining philosophers is done through the transitions depicted in Figure 6.14. More precisely, the figure shows the transitions from the component DP_a that are used to communicate with component DP_b, assuming that a and b are in external conflict. For each interaction c that is in external conflict with a, DP_a contains a copy of these transitions and additional places w_{f_c} and r_{f_c}. All ports whose name starts with S are send ports and all ports whose name starts with R are receive ports.

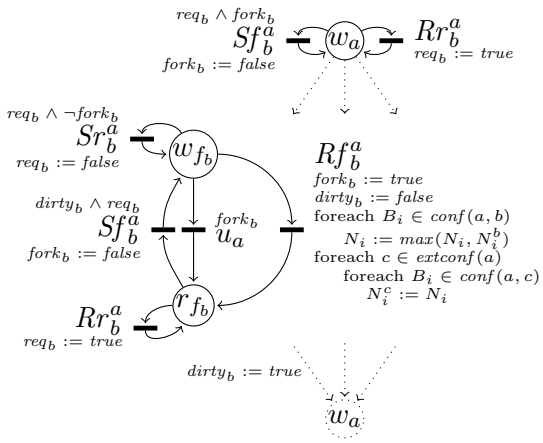

Figure 6.14: Mechanism to exchange forks between components of the dining philosophers protocol.

For the component DP_a, the status of its negotiation with b is encoded through the boolean variables req_b, $fork_b$ and $dirty_b$. The variables req_b (respectively $fork_b$) are true whenever DP_a hold the request (respectively the fork) shared with b. The variable $dirty_b$ indicates whether the fork is dirty, in which case DP_a cannot keep it upon reception of a request.

Whenever the negotiation starts, if DP_a already holds the fork shared with b, the unary transition u_a brings the token directly in r_{f_b}. Otherwise, the request req_b is sent to b (trough the port Sr_b^a) and the component awaits for the fork to reach r_{f_b}. A newly received fork is always considered as clean. Furthermore, the values of N_i variables are updated on reception of the fork.

The component DP_a may receive a request when it holds the fork (at state r_{f_b}). This request is honored either immediately if the fork is dirty, otherwise the component keeps the fork until it takes a decision that brings back the component in w_a state. As soon as a decision is taken, either granting or denying execution of a, the fork is considered as dirty. The dirty fork may be reused as long as no request for it is received.

Definition 6.7. Given a BIC model $\kappa\gamma(B_1, \ldots, B_n)$ and a partition $\gamma_1, \ldots, \gamma_k$ of the interactions, the component handling reservations for interaction a is $DP_a = (L^{DP_a}, P^{DP_a}, T^{DP_a}, X^{DP_a}, \{X_p\}_{p \in P^{DP_a}}, \{g_\tau\}_{\tau \in T^{DP_a}}, \{f_\tau\}_{\tau \in T^{DP_a}})$, where:

- L^{DP_a} contains the waiting place w_a, and the received place r_a.
- P^{DP_a} contains the ports rsv_a, ok_a, $fail_a$ and u_a.
- X^{DP_a} contains the variables $\{n_i^a \mid B_i \in invl_\kappa(a)\}$, and the variables $\{N_i \mid B_i \in invl_\kappa(a)\}$. The variables associated to the port rsv_a are $X_{rsv_a} = \{n_i^a \mid B_i \in invl_\kappa(a)\}$. The ports ok_a and $fail_a$ do not have associated variables.
- For each interaction $b \in extconf_\kappa(a)$:

- L^{DP_a} contains the additional places w_{f_b} and r_{f_b}.
- X^{DP_a} contains the additional variables $\{N_i^b \mid B_i \in conf(a,b)\}$, $fork_b$, req_b and $dirty_b$.
- P^{DP_a} contains the additional ports Sf_b^a, Rf_b^a, Sr_b^a and Rr_b^a. The variables $\{N_i^b \mid B_i \in conf(a,b)\}$ are associated to both ports Sf_b^a and Rf_b^a. Ports Sr_b^a and Rr_b^a do not have associated variables.
- T^{DP_a} contains the transitions:
  - (w_a, Rr_b^a, w_a) no guard and has $req_b := true$ as update function.
  - (w_a, Sf_b^a, w_a), guarded by $req_b \wedge fork_b$ guard and with $fork_b := false$ as update function.
  - $(w_{f_b}, SR_b^a, w_{f_b})$, guarded by $req_b \wedge \neg fork_b$, with the update function $req_b = false$.
  - (w_{f_b}, u_a, r_{f_b}), guarded by $fork_b$, with no update function.
  - $(w_{f_b}, Rf_b^b, r_{f_b})$, with no guard the following update function:
    $dirty_b := false$
    $fork_b := true$
    `foreach` $B_i \in conf(a,b)$
    $\quad N_i := max(N_i^b, N_i)$
    `foreach` $c \in extconf_\kappa(a)$
    $\quad$ `foreach` $B_i \in conf(a,c)$
    $\quad\quad N_i^c := N_i$
  - $(r_{f_b}, Rr_b^a, r_{f_b})$, with no guard and $req_b := true$ as update function.
  - $(r_{f_b}, Sf_b^a, w_{f_b})$, guarded by $dirty_b \wedge req_b$ and with $fork_b := false$ as update function.

- T^{DP_a} contains the transitions:
  - $\tau_{rsv_a} = (w_a, rsv_a, r_a)$, with no guard and no update function.
  - $\tau_{ok_a} = (\{r_{f_b} | b \in extconf(a)\}, ok_a, w_a)$ guarded by $G_{\tau_{ok_a}} = \bigwedge_{B_i \in invl_\kappa(a)} n_i^a > N_i$, with the following update function:
    `foreach` $B_i \in participants(a)$
    $\quad N_i := n_i^a$
    `foreach` $b \in extconf_\kappa(a)$
    $\quad$ $dirty_b := true$
    $\quad$ `foreach` $B_i \in conf(a,b)$
    $\quad\quad N_i^b := N_i$
  - $\tau_{check_a} = (r_a, u_a, \{w_{f_b} \mid b \in extconf_\kappa(a)\})$, guarded by $G_{\tau ok_a}$ and without update function.
  - $\tau_{fail_a} = (r_a, fail_a, w_a)$ and $\tau^2_{fail_a} = (\{r_{f_b} | b \in extconf(a)\}, fail_a, w_a)$, guarded by $\neg G_{\tau_{ok_a}}$. The transition τ_{fail_a} has no update function. The transition $\tau^2_{fail_a}$ has

the following update function:
`foreach` $b \in extconf_\kappa(a)$
$dirty_b := true$

The interactions γ^{DP} between components of this conflict resolution protocol transmit the requests and the forks. For any couple (a, b) of interactions in external conflict, γ^{DP} contains the two Send/Receive interactions:

- a Send/Receive interaction from port Sf_a^b to port Rf_b^a, and,
- a Send/Receive interaction from port Sr_a^b to port Rr_b^a.

Furthermore, for each externally conflicting interaction, γ^{DP} contains the unary interaction involving the port u_a.

6.3.4 Connections between Layers

To complete the description of the three layer model, we have to define the interactions between the distributed components. Between the components layer and the engines layer, offers and notifications are exchanged as in the previous cases. Communication between the engines layer and the conflict resolution protocol layer involves *rsv*, *ok* and *fail* messages transmission.

Definition 6.8. Given a BIC model $\kappa\gamma(B_1, \ldots, B_n)$ and a partition $\gamma_1, \ldots, \gamma_k$ of the interactions, the cross-layer Send/Receive interactions γ^{SR} of the distributed model include:

- for each component B_i and each class γ_j of the partition, a Send/Receive interaction from port o_i^j to port $o_i^{E_j}$,
- for each port $p \in \bigcup_{i=1}^n P_i$ and each class γ_j of the partition, a Send/Receive interaction from port p^{E_j} to port p,
- for each externally conflicting interaction a:
  - a Send/Receive interaction from port $rsv_a^{E_j}$ to port rsv_a,
  - a Send/Receive interaction from port ok_a to port $ok_a^{E_j}$, and
  - a Send/Receive interaction from port $fail_a$ to port $fail_a^{E_j}$.

Let $B = \kappa\gamma(B_1, \ldots, B_n)$ be a BIC component and $\gamma_1, \ldots, \gamma_k$ a partition of the interactions. We denote by $\{a_1, \ldots, a_m\}$ the set of externally conflicting interactions. We define for each conflict resolution protocol the corresponding 3-layer BIP model:

- $B_{CP}^{SR} = \gamma^{SR}(B_1^{SR}, \ldots, B_n^{SR}, E_1, \ldots, E_k, CP)$, embedding the centralized conflict resolution protocol,

- $B^{SR}_{TR} = (\gamma^{SR} \cup \gamma^{TR})(B^{SR}_1, \ldots, B^{SR}_n, E_1, \ldots, E_k, TR_{a_1}, \ldots, TR_{a_m})$, embedding the token ring conflict resolution protocol and
- $B^{SR}_{DP} = (\gamma^{SR} \cup \gamma^{DP})(B^{SR}_1, \ldots, B^{SR}_n, E_1, \ldots, E_k, DP_{a_1}, \ldots, DP_{a_m})$, embedding the dining philosophers conflict resolution protocol.

6.3.5 Correctness

We first show that the 3-layer model B^{SR}_{CP} is indeed a Send/Receive model as defined in Section 6.3. We then prove that the initial high-level BIC model is observationally equivalent to B^{SR}_{CP}. Finally, we prove trace equivalence of models embedding other implementations of the conflict resolution protocol with the original BIC model.

Compliance with Send-Receive Model

We need to show that receive port of B^{SR}_{CP} will unconditionally become enabled whenever one of the corresponding send ports is enabled. Intuitively, this holds since communications between two successive layers follow a request/acknowledgement pattern. Whenever a layer sends a request, it enables the receive port to receive acknowledgement and no new request is sent until the first one is acknowledged.

Proposition 6.9. *Given a BIC model B and a partition of its interaction, the model B^{SR}_{CP} meets the properties of Definition 5.2.*

Proof. The send ports and receive ports determined in Definition 6.8 respect the syntax presented in the two first properties of Definition 5.2. We now prove the third property, that is whenever a send port is enabled, its associated receive port will unconditionally become enabled.

Between engines and conflict resolution protocol layers, for *rsv*, *ok* and *fail* interactions related to $a \in \gamma_j$ it is sufficient to consider the places try_a in the engine E_j, w_a and r_a in the conflict resolution protocol layer. Initially, the configuration is as follows: try_a is empty, and w_a is active. From that configuration only the send port $rsv_a^{E_j}$ of the engine might be enabled, and the receive port rsv_a is enabled. If the rsv_a message is sent, in the reached configuration try_a is active. Only send ports ok_a and $fail_a$ in the conflict resolution protocol might be enabled, and the associated receive ports in engines are enabled. Then, if either an *ok* or a *fail* interaction takes place, we switch back to the initial configuration.

Concerning interactions between a component B^{SR}_i and the engines, we consider a first configuration where no $s_p^{E_j}$ place is active, for p exported by B_i. Note that the initial state falls in that case, In this configuration, only send ports corresponding to offer interaction may become enabled and by construction of the engine the associated received ports are enabled as well. The property thus holds in this configuration.

The other configuration considered is reached by executing either a transition labelled by a or ok_a in one of the engines. In the first case, no other interaction involving the current round of offers from B_i can take place; otherwise, it would be externally

conflicting with a. In the second case, according to the conflict resolution protocol, ok_a is given for the current participation number for the component B_i^{SR} and no other interaction using this number will be granted. Thus in all cases, for each round of offers from B_i^{SR} there is only one active place s_p with p exported by B_i among all the engines and thus one notification send port for B_i^{SR} enabled. The latter may have to finish sending all the offers to reach the stable state from which the receive port p is enabled. Thus, the property holds in that configuration as well. □

This proof ensures that any component ready to perform a transition labeled by a send port will not be blocked by waiting for the corresponding receive ports. In other terms, it proves that any Send/Receive interaction is initiated by the sender.

Observational Equivalence between Original and Transformed Models

We show that B and B_{CP}^{SR} are observationally equivalent. We consider the correspondence between actions of B and B_{CP}^{SR} as follows. To each interaction $a \in \gamma$ of B, we associate either the binary interaction ok_a or the unary interaction a of B_{CP}^{SR}, depending whether a is externally conflicting. All other interactions of B_{CP}^{SR} (offer, notification, reserve, fail) are unobservable and denoted by β.

We proceed as follows to complete the proof of observational equivalence. Among unobservable actions β, we distinguish between β_1 actions, that are interactions between the atomic components layer and the engines (namely offer and notification), and β_2 actions that are interactions between the engines and the conflict resolution protocol (namely reserve and fail). We denote by $q^\perp$ a state of B_{CP}^{SR} and q a state of B. A state of B_{CP}^{SR} from where no β_1 action is possible is called a *stable state*, in the sense that any β action from this state does not change the state of the atomic components layer.

Lemma 6.10. *From any state $q^\perp$, there exists a unique stable state $[q^\perp]$ such that $q^\perp \xrightarrow{\beta_1^*} [q^\perp]$.*

Proof. The state $[q]^{SR}$ exists since each Send/Receive component B_i^{SR} can do at most $k+1$ β_1 transitions: receive a response and send an offer to each engine. Since two β_1 transitions involving two different components are independent (i.e. modify distinct variables and places), the same final state is reached independently of the order of execution of β_1 actions. Thus $[q]^{SR}$ is unique. □

The above lemma proves the existence of a well-defined stable state for any of the transient states reachable by the distributed model B_{CP}^{SR}. The state $[q^\perp]$ verifies the property $q^\perp \xrightarrow{\beta_1^*}_{SR} [q^\perp]$ and $[q^\perp] \not\xrightarrow{\beta_1}_{SR}$. Furthermore, Lemma 6.11 asserts that, at state $[q^\perp]$, atomic components are in a stable state and variables in the engine have the same value as the corresponding variables in components.

Lemma 6.11. *At state $[q^\perp] = ((\ell_1, v_1), \ldots, (\ell_n, v_n), (m^{E_1}, v^{E_1}), \ldots, (m^{E_k}, v^{E_k}), (m^{CP}, v^{CP}))$, we have*

- *$\forall i \in \{1, \ldots, n\}, \ell_i \notin L^\perp$, and*

- $\forall j \in \{1,\dots,k\} \forall x^{E_j} \in X^{E_j}$ *s.t.* $x \in X_i, v_i(x) = v^{E_j}(x^{E_j})$.

Proof. By construction of the distributed atomic components, for each busy state $\perp^j_\ell$ there is one outgoing offer transition labeled by a send port, with a guard that is always true. Therefore, at state $[q^\perp]$ every atomic component is in a stable control location.

In each engine E_j, at a stable state, the last message received or transition executed cannot be a unary interaction a or *rsv* message, since they would trigger a notification followed by an offer, which contradicts stability of the state. Therefore the last transition executed is either an offer or a *fail* message reception or a *rsv* message emission. For each variable $x^E \in X^E$, the last modifying transition was the offer from the corresponding atomic component B_i, which ensures $v^{E_j}(x^{E_j}) = v_i(x)$. □

As for the centralized engine, the value of each the variable $@_i^{E_j}$ in E_j is the same as in the atomic component. In the atomic components, the value of $@_i$ is the current stable state, therefore $v^{E_j}(@_i^{E_j}) = \ell_i$. Furthermore, the values of the n_i variables are the same in the atomic components and in the engines $\forall i,j \in \{1,\dots,n\} \times \{1,\dots,k\}, n_i = n_i^{E_j}$.

Lemma 6.12. *When B^{SR}_{CP} is in a stable state, for each pair $i \in \{1,\dots,n\}$, we have $n_i > N_i$.*

Proof. Initially, for each component B_i, $N_i = 0$ and $n_i = 1$. By letting all components sending offers to all engines, we reach the first stable state where the property holds, since β_1 actions do not modify the N_i variables.

The variables N_i in the conflict resolution protocol are updated upon execution of an ok_a transition, using values provided by the engines, that are values from components according to Lemma 6.11. Thus, in the unstable state reached immediately after an ok_a transition, we have $n_i = N_i$ for each component B_i^{SR} participant in a. Then, the notification transition increments participation numbers in components so that in the next stable state $n_i > N_i$. For components $B_{i'}$ not participating in a, by induction on the number of *ok* interactions, we have $n_{i'} > N_{i'}$. □

Lemma 6.12 shows that the participation numbers propagate in a correct manner. In particular, at any stable state the conflict resolution protocol has only previously used values and engines have the freshest values, that is the same as in the atomic components.

To prove the correctness of our transformation, we exhibit a relation between the states of the original model and the states of the distributed model and prove that it is an observational equivalence. We define the relation by assigning to each state $q^\perp \in Q^{SR}$ an equivalent state $equ(q^\perp) \in Q$. This function considers only the states of distributed atomic components at $[q^\perp]$. According to Lemma 6.11, at $[q^\perp] = ((\ell_1^\perp, v_1^\perp), \dots, (\ell_n^\perp, v_n^\perp), \dots)$ atomic components are in are stable control states. Therefore control states $\ell_i^\perp$ are valid control states for the original atomic components. Similarly, by restricting each valuation $v_i^\perp$ to the variables that are present in the original atomic component B_i, we obtain a valuation $v_i = v_i^\perp|_{X_i \setminus X_i^\perp}$ that is a valid valuation for B_i. With the previous notations, we define $equ(q^\perp) = ((\ell_1^\perp, v_1), \dots (\ell_n^\perp, v_n))$.

Theorem 6.13. $B \sim B^{SR}_{CP}$.

Proof. We define the relation $R = \{(q, q^\perp) \in Q \times Q^{SR} \mid equ(q^\perp) = q\}$. Let $q, q \in Q$ be some states of B, $q^\perp, r^\perp \in Q^{SR}$ be some states of B^{SR}, and $a \in \gamma$ an interaction. Since there is no unobservable action in the original model, R is an observational equivalence if it meets the following properties:

(i) If $(q, q^\perp) \in R$ and $q^\perp \xrightarrow{\beta}_{\gamma^{SR}} r^\perp$, then $(q, r^\perp) \in R$.

(ii) If $(q, q^\perp) \in R$ and $q^\perp \xrightarrow{a}_{\gamma^{SR}} r^\perp$, then $\exists r \in Q$ such that $q \xrightarrow{a}_\kappa r$ and $(r, r^\perp) \in R$

(iii) If $(q, q^\perp) \in R$ and $q \xrightarrow{a}_\kappa r$ then $\exists r^\perp \in Q^{SR}$, such that $q^\perp \xrightarrow{\beta^* a}_{\gamma^{SR}} r^\perp$ and $(r, r^\perp) \in R$

The property (i) is a direct consequence of Lemma 6.10: if $q^\perp \xrightarrow{\beta}_{\gamma^{SR}} r^\perp$ then either:

- β is a β_2 action and does not modify the state of atomic components. Thus $equ(q^\perp) = equ(r^\perp)$.
- β is a β_1 action and by definition $[q^\perp] = [r^\perp]$ which implies $equ(q^\perp) = equ(r^\perp)$.

To prove property (ii), we assume that either the transition a is possible in the engine handling a or the transition ok_a is possible in the component CP. These transitions are enabled according to values of variables in the engine handling a. If a is not externally conflicting, these values cannot be modified when reaching the state $[q^\perp]$, since β_1 transitions modify distinct variables in the engine. If a is externally conflicting, ok_a is enabled, meaning that for each component B_i involved in a, $n_i > N_i$. In particular, for each involved component B_i, the offer corresponding to the number n_i has not been consumed yet. In both cases, the values of variables associated to component involved in the engine handling a remain the same when reaching the next stable state $[q^\perp]$.

Let us denote a by $\{p_i\}_{i \in I}$. If the transition a (or rsv_a is possible in the engine, each variable $x^E_{p_i}$ is true, and by construction of the distributed atomic component B_i, we have $q_i \xrightarrow{p_i}_i$. By Lemma 6.11, if $G_a^{E_j}$ evaluates to true in E_j then G_a evaluates at state $equ(q^\perp) = (q_1, \ldots, q_n)$. Similarly, if $\kappa_a^{E_j}$ evaluates to true in E_j, $\kappa_a(equ(q^\perp))$ also evaluates to true by Lemma 6.11. Thus we have $equ(q^\perp) \xrightarrow{a}_\kappa r$.

By applying update function $F_a^{E_j}$ and executing notification transitions, one reaches a state where the values in distributed atomic components participant in a are the as same the values in original atomic component after executing a. These values are not changed through the subsequent offers, therefore $(r, [r^\perp]) \in R$ and since $r^\perp \xrightarrow{\beta} [r^\perp]$, we have $(r, r^\perp) \in R$.

To prove (iii), we assume that $q \xrightarrow{a}_\kappa r$ is valid in the original model. By definition of $[q^\perp]$, we have $q^\perp \xrightarrow{\beta_1^*} [q^\perp]$. As in the previous case, the Lemma 6.11 ensures that at state $[q^\perp]$, the values of variables in distributed atomic component and engines are the same. By doing all possible *fail* interactions, which can be executed even if the interaction could be granted, all tokens are brought back in r_i places in engines. Then if a is not externally conflicting, it can be executed directly. Otherwise, the reserve interaction rsv_a is possible and can be executed to reach another stable state. According to Lemma 6.12,

we have $n_i > N_i$ for every component B_i, thus the transition ok_a is enabled. In both cases, we have $q^{\perp} \xrightarrow{\beta_1^*} [q^{\perp}] \xrightarrow{\beta_2^*} \xrightarrow{a} r^{\perp}$. As previously, executing the transition in the engine and sending notifications to participating components yields the same results as executing the interaction in the original model. Thus $(r, r^{\perp}) \in R$. □

Interoperability of Reservation Protocol

The centralized implementation *CP* of the conflict resolution protocol can be seen as a specification. Two other implementations, namely token ring and dining philosophers, can be used as conflict resolution protocol. However, these implementations are not observationally equivalent with the centralized implementation. More precisely, the centralized version defines the most liberal implementation: if two reservation requests a_1 and a_2 are received, the protocol may acknowledge them in any order. This general behavior is not implemented neither by the token ring nor by the dining philosophers implementations, which are focused on ensuring progress. In the case of token ring, the response may depend on the order the token travels through the components. In the case of dining philosophers, the order may depend on places and the current status of forks.

Nevertheless, we can prove observational equivalence if we consider *weaker* versions of the above implementations. More precisely, for the token ring protocol, consider the weaker version $TR^{(w)}$ which allows each conflict resolution component to release the token or provide a fail answer regardless of the values of counters. Likewise, for the dining philosophers protocol, consider the weaker version $DP^{(w)}$, where forks can always be sent to neighbors, regardless of their status, and a fail answer can always be issued. Clearly, a weakened conflict resolution protocol is not desirable for a concrete implementation since it does not enforce progress. But, it is an artifact for proving the correctness of our approach. The following proposition establishes the relation between the different implementations of the Reservation Protocol.

Proposition 6.14. $CP \sim TR^{(w)} \sim DP^{(w)}$

Proof. The observable actions are requests rsv_a, ok_a and $fail_a$ messages. The unobservable actions are token passings for $TR^{(w)}$ and forks exchange for $DP^{(w)}$.

Given a state s^{TR} of the *TR* protocol or a state s^{DP} of the *DP* protocol, we construct a state s^{CP} for the centralized protocol as follows. For each interaction $a \in \gamma$:

- If a request for a is pending in s^{TR} or s^{DP}, the equivalent state s^{CP} is defined such that the place r_a contains a token. Otherwise, the place w_a contains a token.
- For each component B_i involved in a, we set the participation number n_i^a associated to the pending requests in s^{CP} to the value of n_i held in the component managing interaction a, that is TR_a or DP_a.

Moreover, we set the last used participation number N_i in s^{CP} to:

- the value of variable N_i stored on the token in s^{TR}, or

- the maximum value among variables $\{N_i^b\}_{b \text{ externally conflicting}}$ stored on the forks in s^{DP}.

In $TR^{(w)}$ and $DP^{(w)}$, it is clear that any unobservable action does not change the associated state s^{CP}.

Weakening makes the above relation an observational equivalence. Indeed, whenever an action (either ok_a, f_a or r_a) is possible at s^{CP}, then by moving the token, (resp. the forks), we can always reach a state s^{TR} (respectively s^{TR}) where this action is possible as well. Reciprocally, if an action is possible in either s^{TR} or s^{DP} then, in the equivalent state s^{CP}, the same action is allowed and reaches an equivalent state. □

Recall that we denote B_X^{SR} the 3-layer model obtained from the initial system B and that embeds the Reservation Protocol X, which ranges over CP, TR and DP. Also, let us denote by $Tr(B)$ the set of all possible traces of observable actions allowed by an execution of B. We now show the correctness of the implementation, by using the weak implementations of TR and DP. Since the real implementations of these protocols restrict the behavior of their weak version, we only state correctness as trace inclusion with respect to the original model. The traces of the distributed implementation are included in the traces of the original model.

Proposition 6.15. *(i)* $B \sim B_{CP}^{SR} \sim B_{TR^{(w)}}^{SR} \sim B_{DP^{(w)}}^{SR}$
(ii) $Tr(B) \supseteq Tr(B_{TR}^{SR})$ *and* $Tr(B) \supseteq Tr(B_{DP}^{SR})$.

Proof. (i) The leftmost equivalence is a consequence of Theorem 6.13. The other equivalences come from Proposition 6.14 and the fact that observational equivalence is a congruence with respect to parallel composition.

(ii) Trace inclusions come from the fact that any trace of TR (respectively DP) is also a trace of $TR^{(w)}$ (respectively $DP^{(w)}$). □

Practical Aspects The 3-layer model relies on unbounded counters for execution. At the end of Subsection 2.3.1, we recall Bagrodia's suggestion to reset these counters and also suggest to dimension counters according to system's lifetime. However, some solution without counters exists, such as Kumar's token [61], presented in subsection 2.3.2 and α-core. The distributed implementation of LOTOS presented in [81] embeds an algorithm similar to α-core but still relies on counters. In the next section, we present the α-core protocol.

6.4 α-Core

The α-core protocol [73] provides a fully distributed solution where each interaction is handled by a separate coordinator. Contrarily to the previous solution, α-core is able to solve conflicts without using counters. The α-core protocol implements only multiparty interactions. Models with Condition can be implemented through the transformation provided in Subsection 4.2.5.

The α-core protocol specify two different behaviors for participants and coordinators. There is one participant for each atomic component and one coordinator for each interaction. Each participant communicates only with coordinators and each coordinator communicates only with participants.

6.4.1 Protocol Description

The main idea of the protocol is that each coordinator tries to lock all the participants in the interaction it handles. All coordinators lock components sequentially, according to a global order. If a and b are (Interaction) conflicting, the intersection $\mathit{participants}(a) \cap \mathit{participants}(b)$ is not empty. The conflict between a and b is solved as the minimal component in the intersection will be locked either by a or b, but not both.

In α-core, the following messages are sent from a participant to a coordinator:

PARTICIPATE A participant is interested in a single particular interaction (hence it can commit to it), and notifies the related coordinator.

OFFER A participant is interested in one out of several potentially available interactions (a non-deterministic choice).

OK Sent as a response to a **LOCK** message from a coordinator (described below) to notify that the participant is willing to commit on the interaction.

REFUSE Notify the coordinator that the previous **OFFER** is not valid anymore. This message can respond to a **LOCK** message from the coordinator.

Messages from coordinators are as follows:

LOCK A message sent from a coordinator to a participant that has sent an **OFFER**, requesting the participant to commit to the interaction.

UNLOCK A message sent from a coordinator to a locked participant, indicating that the current interaction is canceled.

START Notifying a participant that it can start the interaction.

ACKREF Acknowledging a participant about the receipt of a **REFUSE** message.

Figure 6.15 describes the extended state machine of a participant. Transitions are triggered either by an incoming message or a guard becoming true. If the same incoming message has different effects depending on the state of the variables, several transitions with different guards are provided. This description is not a Send/Receive BIP component, as the `send` function is called in the update functions. Each participant process keeps some local variables and constants:

IS: a set of coordinators for the interactions the participant is interested in.

locks: a set of coordinators that have sent a pending **LOCK** message.

unlocks: a set of coordinators from which a pending **UNLOCK** message was received.

locker: the coordinator that is currently considered.

n: the number of **ACKREF** messages required to be received from coordinators until a new coordination can start.

α: the coordinators that asked for interactions and subsequently refused.

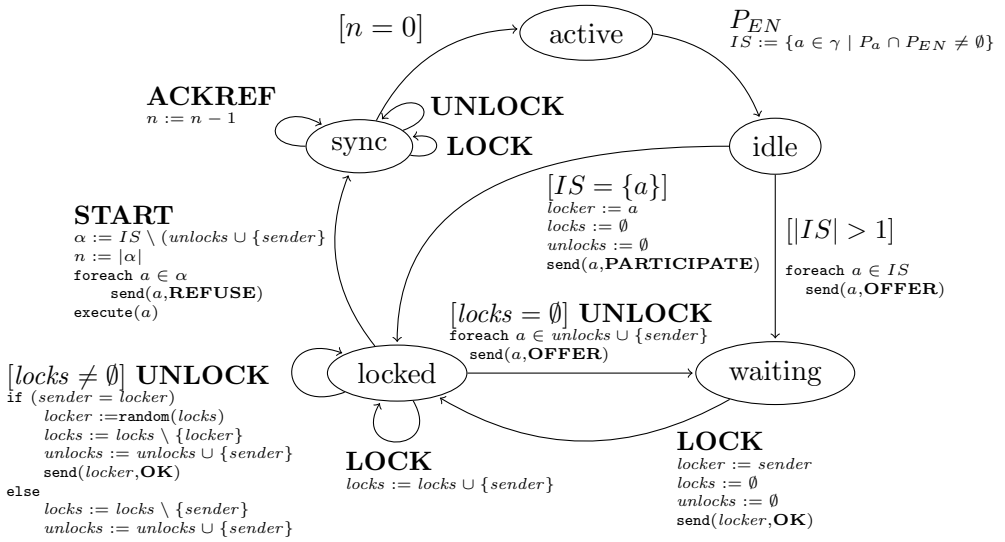

Figure 6.15: Behavior of a participant in α-core.

Initially, a participant is *active*, that is doing some internal computation. Upon termination of its computation, the participant becomes idle with a given set of enabled ports P_{EN} and computes the set of interactions IS that involve these ports. In the sequel, we denote by a both the interaction a and the corresponding coordinator. If only one interaction is possible, the participant sends a **PARTICIPATE** message to the corresponding coordinator and directly goes to state locked. Otherwise, the participant is the source of a conflict and requires to be locked in order to solve that conflict. It sends an **OFFER** message to each coordinator in IS and waits for incoming **LOCK** messages. The first **LOCK** is always granted, all other ones are recorded.

When locked, the participant can receive a **START** message indicating that the locking interaction took place. This triggers execution of the interaction and emission of **REFUSE** messages to all coordinators that could potentially lock the participant. Before resuming internal execution, the participant waits for all coordinators to acknowledge the **REFUSE**.

When locked, the participant may also receive an **UNLOCK** message, because the locking interaction failed to execute. If some other coordinators have sent a **LOCK** message, one of them is chosen randomly to be the next locking coordinator. Otherwise offers are sent again to coordinators that previously sent the sequence **LOCK UNLOCK**.

The behavior presented here has been modified on two points from the one in [73]:

- In the original version, the places active and idle are merged and there is no transition labeled by P_{EN}. The latter was added to link this protocol with the BIP concepts.
- The transition triggered by the **UNLOCK** message when the set *locks* is not empty has been modified. In the original version, it is assumed that the sender of the **UNLOCK** message is always the locking coordinator. It is also possible that another coordinator sent that message in which case the version from [73] deadlocks.

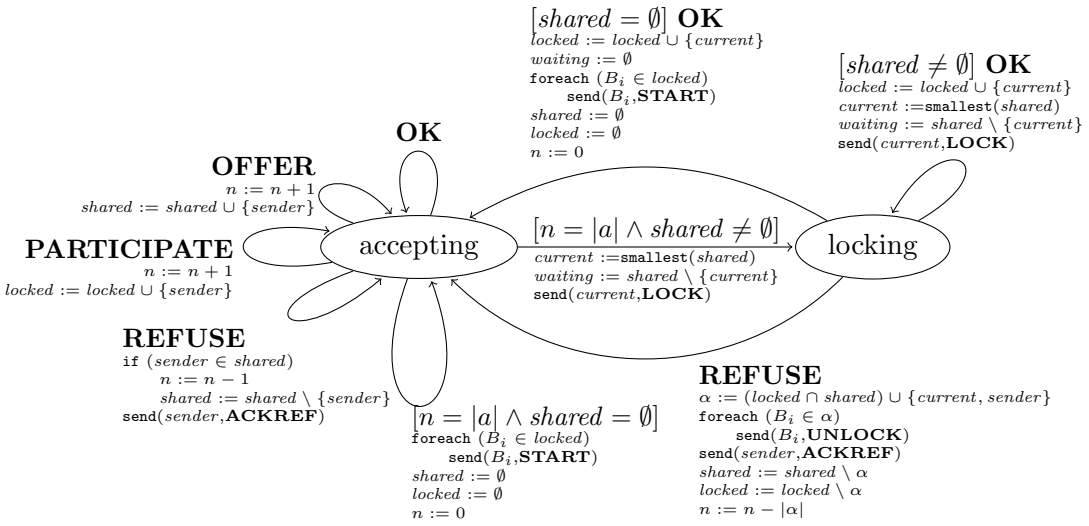

Figure 6.16: Behavior of a coordinator for interaction a in α-core.

The coordinator handling interaction a is depicted in Figure 6.16. Initially, the coordinator is accepting incoming **OFFER** and **PARTICIPATE** messages, and counts them in the n variable. We denote by $|a|$ the number of participants in a. Whenever $|a|$ **OFFER** or **PARTICIPATE** messages have been received, the interaction is enabled. If there is no shared participant, that is all participants have sent a **PARTICIPATE** message, the interaction a is not conflicting at that state. In that case, there is no need to lock participants and a **START** message can be directly issued. Otherwise, the coordinator switches to the locking state and sends the first **LOCK** message to the smallest participant to lock, according to the global order on the participants.

In the locking state, when the coordinator receives an **OK** message, it marks the sending participant as locked and sends a **LOCK** message to the next participant to lock, according to the global order. If all participants have been locked, the coordinator switches back to the accepting state, emits **START** messages and reset the variables for the next execution. If a **REFUSE** message arrives before the locking is complete, the coordinator sends an **UNLOCK** message to each locked participants and switches back

to the initial state. Participants that sent a **PARTICIPATE** message are still locked, however other ones have to resend an **OFFER** before the locking phase restarts.

Finally, if a **REFUSE** message arrives while the coordinator is waiting for offers, the number of participants ready to execute the interaction (n) is decremented, if the sender is in the list of participants that sent an offer (called *shared*). The refusing participant is also removed from that list.

This last transition (receiving **REFUSE** from the accepting state) was modified from the version in [73], as the latter one would unconditionally decrement n. Such a behavior leads to a deadlock as it can remove twice the same participant from the number of ready participants, thus preventing the locking to start. This bug was detected and corrected automatically using code mutation techniques in [58].

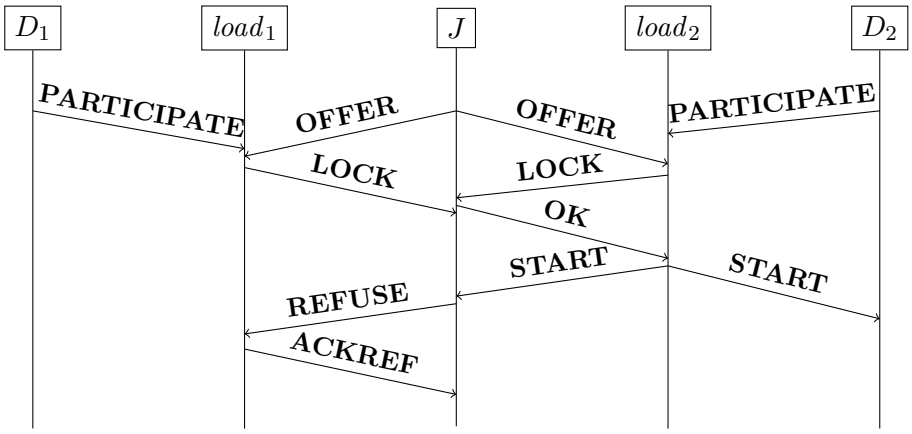

Figure 6.17: First messages exchanged in the α-core protocol during execution of the model from Figure 5.1.

We illustrate the execution of this protocol on our running example from Figure 5.1. Recall that α-core does not take Condition into account. Execution of this example starts by resolving the conflict between $load_1$ and $load_2$. Figure 6.17 depicts the corresponding messages in the α-core protocol. First all participants send **OFFER** or **PARTICIPATE** messages. Since each disc D_i can only perform the $load_i$ interaction, it sends a **PARTICIPATE** message. The jukebox can perform either $load_1$ or $load_2$ and sends an offer to both. Once the coordinator for $load_i$ has received the two offers, it starts locking participants. In the example, only the jukebox is a shared participant. In the execution from Figure 6.17, the **LOCK** message from $load_2$ arrives first at J and therefore $load_2$ wins the conflict. After receiving the **OK** message from J, the coordinator $load_2$ has locked all the participant and can issue **START** messages so that the participants start executing. Finally, J reports to $load_1$ that it lost the conflict through a **REFUSE** message. Upon reception of the **REFUSE** message, the coordinator $load_1$ forgets the offer from J and returns to the state accepting. However, it recalls that D_1 sent a **PARTICIPATE** message.

6.4.2 SR-BIP Implementation of α-Core

We do not formally define the SR-BIP implementation of α-core but rather give an overview of its construction. Each atomic component from the initial BIP model is implemented through a set of components. One of these components is the α-core participant behavior from Figure 6.15. Other components include components for receiving and sending message and the distributed version of the atomic components. Similarly, each interaction is implemented through a set of components. The communications between the set of components implementing an interaction and the set of components implementing an atomic component remain asynchronous message-passing.

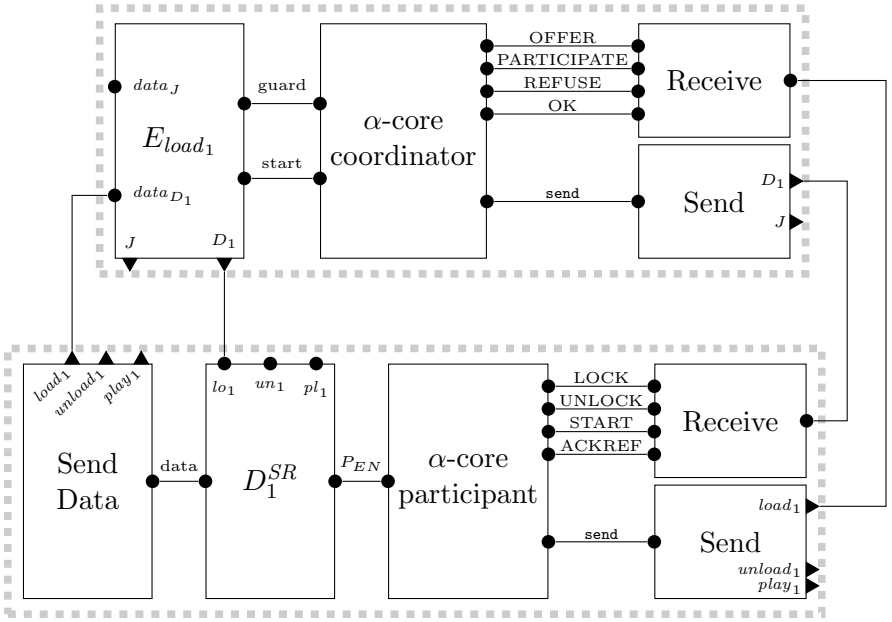

Figure 6.18: Fragment of the distributed BIP model using α-core protocol implementing the component D_1 and the interaction $load_1$.

Figure 6.18 presents a fragment of the α-core based distributed implementation of the example from Figure 5.1. This fragment contains a set of components implementing the atomic component D_1 and a set of components implementing the interaction $load_1$.

The grey dashed line at the bottom of the Figure delimits the part corresponding to the atomic component D_1. This part contains a distributed version D_1^{SR} of D_1, which is similar to the distributed version for the centralized engine. The main difference is that the offer is cut in two parts, first data to send is copied to a dedicated component for sending data to the right engines, then the set of enabled ports is copied to the component implementing the behavior from Figure 6.15. Two additional components are responsible for receiving and sending messages. In particular, the Receive component stores the received messages until they are consumed by the α-core participant component, through one of the synchronizations. The Send component has one port `send` exporting a message

and a set of recipients. Upon activation of this port, the message is sent to all the recipients.

Similarly, the part implementing the interaction $load_1$ contains a version of a distributed engine that receives the data, evaluates the guard and computes the data transfer function. The guard interaction is used by the coordinator to decide whether to start locking, i.e. locking happens only if the guard is true. Similarly, when issuing the message **START**, the coordinator triggers computation of the data transfer function trough the start interaction. The engine directly notifies the distributed component so that computation of the latter start immediately. In the meantime, the α-core participant component receive the **START** and reaches the sync state to wait for **ACKREF** messages.

Finally, by merging together local components using the technique presented in 7.2.3, one obtains a Send/Receive BIP model. In that model, each component and each interaction is implemented through a dedicated component, that is the composition of previously described components.

6.5 Optimization using Knowledge with Perfect Recall

The protocol α-core implements a very simple but efficient optimization through the distinction between **OFFER** and **PARTICIPATE** messages. Emission of a **PARTICIPATE** message happens whenever the component knows that there is only one interaction it can execute next. Note that within the BIP context, even if only one port is possible, there might be several interactions involving that port.

With the α-core protocol, executing an interaction a involving n components that have no other choice than executing a requires $2n$ messages: a **PARTICIPATE** and a **START** message for each component. The same interaction using **OFFER** messages requires at least additional **LOCK** and **OK** messages, that is at least $4n$ messages to execute a. Furthermore, once the **OFFER** is consumed by a coordinator, it has to be canceled by a **REFUSE** message sent to conflicting coordinators. Thus it is highly beneficial to detect cases where **PARTICIPATE** messages can be sent instead of **OFFER** messages.

A component basically knows its local state and for each port the set of corresponding interactions. A **PARTICIPATE** message is sent in the case where only one transition is possible from the current state and this transition is labeled by a port that is involved in only one interaction. For instance, the component D_1 from the example in Figure 5.1 can only take part in $load_1$ interaction from the state F_1.

However, there are some global states where only one interaction is enabled, but each component sees locally that it can participate in several of them. Consider the Figure 2.3 presenting the interactions allowed by the control state of the Jukebox example. At state $\{L_1, F_2, 1, B\}$, only the interaction $unload_1$ is possible. The component D_1 is at control state L_1 from where both ports un_1 and pl_1 are enabled, and hence sends an **OFFER**.

We propose to use knowledge with perfect recall in components so that such cases can be detected [16]. Our approach relies only on control states and does not take data into account. In particular, if an interaction is not enabled because of its guard, our method

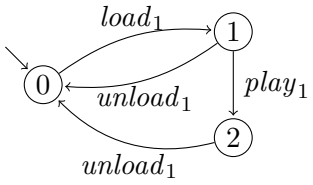

Figure 6.19: Support automata for D_1.

considers it as enabled. The support automaton for a component B_i of a BIP model $B = \gamma(B_1, \ldots, B_n)$ is obtained as follows:

- Building the global control behavior $B^{ctrl} = (Q, \gamma, \rightarrow_\gamma{}')$ that is obtained by considering the behavior of B assuming that all guards in B are true, The states of this automaton contain only control state information.
- Defining A_i as the set of interactions $\{a \in \gamma \mid P_a \cap P_i \neq \emptyset\}$ in which B_i participates,
- Finally, building the support automaton $\mathcal{K}_i$ as prescribed in Section 3.2 by considering B^{ctrl} as global behavior and A_i as visible actions.

In Figure 6.19, we present the minimized version of the support automata for the component D_1 from Figure 5.1. States are numbered instead of containing a set of global states, as in Figure 3.3. This automaton can be played in parallel with the component, that is, whenever an interaction involving the component takes place, the corresponding transition of the support automaton is executed as well. On this example, the state 0 correspond to the control location F_1 and that both states 1 and 2 correspond to the control location L_1 of the component. In particular, the support automaton recalls whether $play_1$ has been executed (state 2) or not (state 1). If the support automaton is at state 2 the component locally knows that only $unload_1$ is possible.

By construction, the global behavior $(Q, \gamma, \rightarrow)$ from Definition 2.5 of the system of processes $\mathcal{K}_1, \ldots, \mathcal{K}_n$ is actually B^{ctrl}. Since in B^{ctrl} all guards are true, the transitions allowed in B are a restriction of the transitions allowed in B^{ctrl}. Therefore, we have that $\rightarrow_\gamma \subseteq \rightarrow$. In other words, by considering only interactions allowed by the composition of the support automata $\mathcal{K}_i$, we have an over approximation of the interactions actually allowed in B.

We use this property to build a correct and optimized version of α-core participant. The modified participant includes a variable q that contains the current state of the support automaton. The support automaton is encoded as a transition function δ such that $\delta(q, a)$ returns the unique state q' such that $q \xrightarrow{a}_{\mathcal{K}_i} q'$. We denote by $EN_{\mathcal{K}_i}(q)$ the set of interactions that are possible from the state q of the support automaton. Formally, $EN_{\mathcal{K}_i}(q) = \{a \in \gamma \mid q \xrightarrow{a}_{\mathcal{K}_i}\}$. The modified α-core participant differs in two points from the original α-core participant:

- In the transition from active to idle, the set IS is computed as $IS := \{a \in \gamma \mid P_a \cap p_{EN} \neq \emptyset\} \cap EN_{\mathcal{K}_i}(q)$.

- In the transition from locked to sync, the following computation is added: $q := \delta(q, a)$.

The first modification actually restricts the set of interactions to which an offer has to be sent. In the case where the size of the restricted set is 1, a **PARTICIPATE** message is issued. The second modification updates the state of the support automaton.

In [16], we proposed the same kind of construction for α-core coordinator. In that case, the knowledge automaton takes into account the offers and the interactions executions (transitions that issue **START** messages in the coordinator). This support automaton can be used in two manners:

- discard offers that will be received again later (which can be seen on the automaton),
- avoid locking participants that cannot send an offer before the next interaction involving them.

This last optimization relies on the fact that each α-core participant waits for all coordinators to acknowledge a **REFUSE** message before sending new offers. This is not the case in the 3-layer BIP implementation, where each component can resend an offer as soon as it received a notification.

The correctness of the two optimizations are proved in [16]. To prove the first one, we remark that composing all support automata obtained from the components gives back the global behavior of the original model. The correctness of the second (not explained here) construction is show through a trace equivalence. Note that the two optimizations can be combined, provided the second one is computed using the support automata from the first optimization instead of the original behavior of the components. Indeed, as the second optimization takes offer into accounts, it has to take into account possible restrictions of these offers.

6.6 Discussion

In this chapter, we presented several solutions to resolve conflicts arising between interactions. The conflict-free solution can be seen as a degenerate case of the 3-layer solution where no conflict resolution is needed. The 3-layer solution is very general and flexible but requires to use unbounded counters. The protocol α-core provides a solution that does not use counters.

Each of these solutions exhibits different features. The 3-layer BIP solution encompasses Condition and is parameterized by an arbitrary partitioning of the interactions. The α-core solution dynamically disables unneeded conflict resolution on a per component-basis. This last feature can be optimized by adding knowledge with perfect recall in components. We have not extended each solution to support each feature, however it seems possible.

Extending α-core to support Condition can be done by having coordinators locking both participants and observed components. Whenever the interaction starts, participants are sent a **START** message and observed components an **UNLOCK** message.

Multiple coordinators may lock the same observed component, as observing the same component does not create a conflict.

Having a single α-core coordinator for handling several interactions requires heavier modifications. In particular, in the original α-core version, **OFFER** and **PARTICIPATE** messages are simply counted. A modified coordinator would handle several interactions and need a separate count for each interactions. Furthermore, each participant would have to indicate the set of interactions in the offers they send, instead of a simple message.

Implementing an equivalent of α-core **PARTICIPATE** message has several implications on the 3-layer BIP. In that case, a such message would be sent when only one engine is the recipient of the offer.

- In the engines, if an interaction is externally conflicting but all its participants sent a **PARTICIPATE** message, the engine can execute the interaction without calling the conflict resolution protocol.
- In the conflict resolution protocol, checking the validity of the offer numbers is needed only for components that sent an **OFFER** message. Only the dining philosophers protocol would benefit of this optimization as the set of forks to acquire would be reduced.

Concerning the knowledge-based optimization, the support automaton can be embedded in the distributed version of the corresponding atomic component. We leave these extensions as future work.

7 Implementation

This chapter discusses the implementation of the methods for generating distributed implementations using the BIP toolbox. We start by presenting in Section 7.1 the BIP language which provides a common textual representation for all the different models used. In Section 7.2, we present the existing BIP tools and focus on the tools implementing the methods presented in the previous chapters. Finally, Section 7.3 presents how the different tools could be organized in a coherent tool for generating distributed implementations.

7.1 The BIP Language

The BIP language represents components of the BIP framework [10]. BIP language provides syntactic constructs for describing systems. In practice, variables, data type declarations, expressions and statements are written in C, although another language could be used. The BIP language can be seen as a set of structural syntactic constructs for defining component behavior, specifying the coordination through connectors and describing the priorities. The basic constructs of the BIP language are the following:

- atomic component: to specify behavior, with an interface consisting of ports. Behavior is described as a set of transitions.
- connector: to specify the coordination between the ports of components, and the associated guarded actions.
- priority: to restrict the possible interactions, based on conditions depending on the state of the integrated components.
- composite component: to specify systems hierarchically, from other atoms or compounds, with connectors and priorities.
- model: to specify the entire system, encapsulating the definition of the components, and specify the top level instance of the system.

We now detail some parts of the Jukebox example from Figure 4.6. In the BIP language, one starts by defining types that are later instantiated, allowing several instances of the same element (components, connectors or ports) to created by using the same type. The most basic types are the port types. Each port has a type, which defines a generic name for each variable that is exported by such a port, as well as a type for each variable. Here we assume that `CD_DATA` is a declared C type.

```
port type IntPort (int i)
port type CdDataPort (CD_DATA data)
port type EventPort
```

As an example, we provide the BIP code for the Disc atomic component of Figure 4.3.

```
atomic type Disc
  data int c
  data CD_DATA data

  export port IntPort lo(c)
  export port CdDataPort pl(data)
  export port EventPort un

  place F,L
  initial to F do  { c=0 ; }
  on lo from F to L provided true
    do {c = c+1 ;}
  on pl from L to L
  on un from L to F
end
```

The description starts with variables, ports and control declaration. Upon port declaration, the variables that are bound to the port are explicitly given, and their type match those in the port type definition. The construct **initial to** specifies the initial transition, and possibly some initialization function to execute. Each transition of the behavior is declared, with a port (after **on**), a (set of) initial and final state(s) (after **from** and **to**), a guard (after **provided**) and an update function (after **do**). The functions and guards are written using a subset of the C syntax.

We now present the connector types used to describe *load* and *play* interaction.

```
connector type LoadConn(IntPort disc, IntPort jk)
  define disc jk
  on disc jk
    provided disc.i ≤ jk.i/2 + 2
end

connector type PlayConn(CdDataPort input, CdDataPort output)
  define input output
  on input output
    provided true
    down {output.data = input.data }
end
```

A connector type is parameterized by a list of port types that describes its support. The construct **define** defines the set of interactions allowed by the connector, using an algebraic notation. In the example, we have only two synchrons. A trigger would be specified by appending a quote to the port name. For each interaction allowed by the previous expression, a guard and an update function can be provided. Here the update function is provided with the **down** construct. The dotted notation, i.e. `port.var` is used to access the variable `var` associated to the port `port`, as defined in the port type declaration.

The connector described above does not allow hierarchical composition as presented in Subsection 4.2.2. Recall that hierarchical composition requires to export a port "on top" of the connector, which can be done through an **export** construct in the connector definition. An upward propagation function, useful only within a hierarchical connector, may be define with the **up** construct.

A compound component is a new component type defined from existing components by creating their instances, instantiating connectors between them and specifying the priorities. A compound offers the same interface as an atom, hence externally there is no difference between a compound and an atomic component. We define below the compound type that corresponds to the Figure 4.6, assuming that the atomic types Jukebox and Listener have been defined according to the Figure.

```
compound type FullJukebox
  component Disc disc1
  component Disc disc2
  component Jukebox jukebox
  component Listener listener

  connector LoadConn load1 (disc1.lo, jukebox.d1l)
  connector LoadConn load2 (disc2.lo, jukebox.d2l)
  connector PlayConn play1 (disc1.pl, listener.rec1)
  connector PlayConn play2 (disc2.pl, listener.rec2)
  connector EventConn unload1 (disc1.un, jukebox.d1u)
  connector EventConn unload2 (disc2.un, jukebox.d2u)
  connector Singleton think (listener.think)

  priority π1 unload1 < play1
  priority π2 unload2 < play2
end
```

This compound type mainly create one instance for each component, bind them using new instances of connectors and define priority between connectors. Note that when creating connectors and priority, each of them must be named. The dotted notation `port.comp` is used to denote the port `port` of the component instance `comp`. Finally, to complete the description in the BIP language, one must add a top level instance of the main compound type to execute.

7.2 The BIP Toolbox

This section presents the toolbox available with the BIP framework. The BIP toolbox provides a complete implementation, with a rich set of tools for modeling, executing and verifying BIP models.

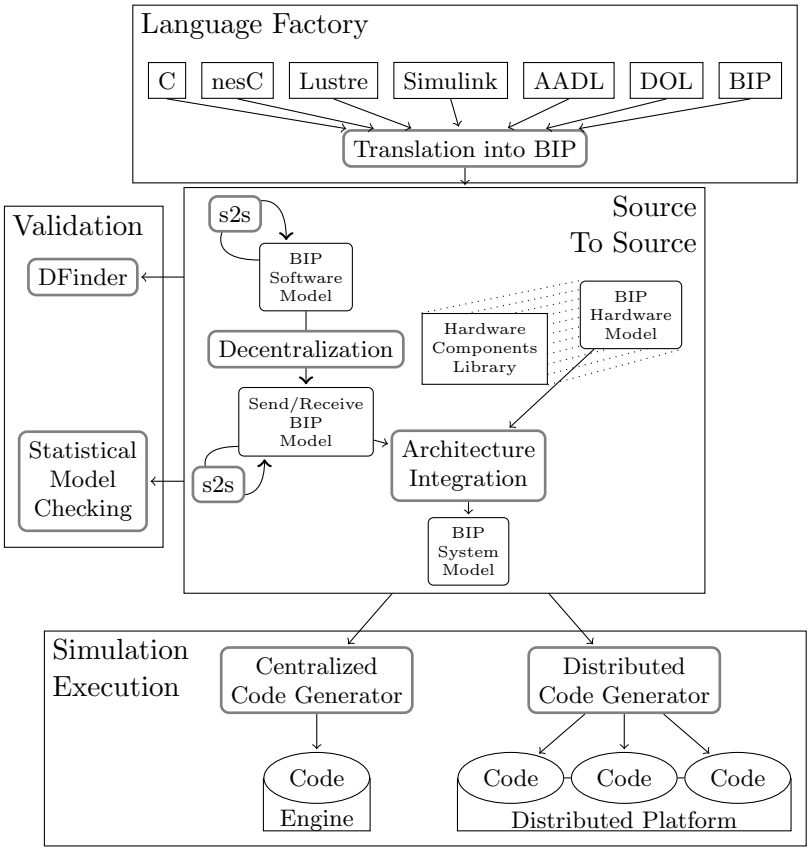

Figure 7.1: Overview of the BIP toolbox.

An overview of the BIP toolbox is shown in Figure 7.1. We distinguish between four categories of tools, namely Language Factory, Source To Source, Validation and Simulation/Execution. We now detail each of theses categories.

7.2.1 Language Factory

These tools translate into BIP existing software that rely on a different model of computation. The input can model the application software, the hardware architecture, or both of them. Existing transformations includes transformations from synchronous languages, i.e. transformations from Lustre [31] and Simulink [79]. These transformations target synchronous BIP [78], that is an extension of BIP dealing efficiently with synchronous models.

Other input languages model both the application software and the hardware architecture. These models can be transformed either into two separate models: one for the software and the other for the architecture or into a single model including both of them, called system model. Transformations to hardware model often rely on a library of hardware components such as memories, buses, processors, that are modeled in BIP. BIP models can be generated from the Architecture Analysis and Design Language (AADL) [35], from nesC/TinyOS [13] and from the Distributed Operation Layer (DOL) [27].

7.2.2 Verification

The BIP toolbox provides tools to validate models. DFinder [20, 21] verifies that a given BIP model satisfies a given property, based on invariants that can be computed in a compositional manner. Statistical model checking [18] allows properties expressed on the execution sequences of the model to be statistically checked.

DFinder

DFinder compositionally checks that a BIP model always satisfies a given property during its execution. Example of such properties include enablement of a given interaction (i.e. EN_a predicate), and by extension enablement of at least one interaction. By checking that the property "at least one interaction is enabled" always holds, one actually checks that the system is deadlock-free.

DFinder relies on invariants such as the ones presented in Chapter 3, that are incrementally computed. These invariants provide an over-approximation $\tilde{\mathcal{R}} = \{q \in Q \mid \mathfrak{I}(q)\}$ of the reachable states of a BIP model. Assume a property ϕ to check, for instance that at least one interaction is enabled: $\phi_{DF} = \bigvee_{a \in \gamma} EN_a$. Verifying that this property holds during execution is done by checking that $\mathfrak{I} \wedge \neg\phi$ is not satisfiable. It means that the intersection between the states violating the property and the states satisfying the invariant is empty. Thus the property holds for all reachable states, since they all satisfy the invariant.

Statistical Model Checking

Statistical model checking is performed using properties described with Probabilistic Bounded Linear Temporal Logic (PBLTL) formulas. These properties refer to the traces of the model. The model has to be expressed using stochastic BIP [18]. One can either ask the probability that a formula holds, or given a formula and a probability rate, ask whether the probability of the formula is higher than the given rate. Checking is performed by interactive cycles of model executions and statistical analysis. More precisely, results of the executions (i.e. traces) are fed to the analyser, which in turn can trigger some additional execution if needed. The process is parameterized by a degree of confidence as it relies on statistical methods.

7.2.3 Source to Source Optimizations

These transformations are presented in [30] and [54]. Flattening replaces a set of hierarchical connectors into an equivalent set of *flat* connectors, that are not hierarchically composed. Merging two components yields a single component with the same behavior and interface as the composition of the two original components. Flattening a BIP model and then merging all the components into a single one improves efficiency of the generated code [30]. In Figure 7.1, these tools appear as "s2s" boxes.

As said in the concrete BIP model presentation, we consider only *flat* models in this thesis. The flattening tool is therefore a crucial requirement in the toolbox, since it allows us to handle hierarchically composed models, at the cost of a pre-transformation to flat models. Intuitively, flattening of connectors is done by composing upwards and downwards data transfer function into a single data transfer function. In Figure 7.2,

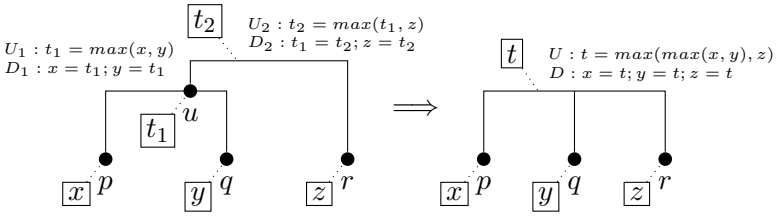

Figure 7.2: Flattening a connector.

we intuitively show how the connector from Figure 4.5 is flattened. The global upwards data transfer function U is obtained from U_2 by replacing each occurrence of t_1 with the value computed by U_1, i.e. $U = U_2 \circ U_1$. Similarly, we replace in D_1 each occurrence of t_1 with the value computed by D_2, i.e. $D = D_1 \circ D_2$. We can obtain a single data transfer function by taking $F = D \circ U = D_1 \circ D_2 \circ U_2 \circ U_1$. Note that here, only one interaction is allowed by this connector. Whenever several interactions are possible, flattening will create the corresponding connectors.

Merging can be done on flat models only. The main idea is that an interaction between components to merge becomes a single Petri net transition into the component resulting of the merge. Figure 7.3 presents a simple example where two components are merged

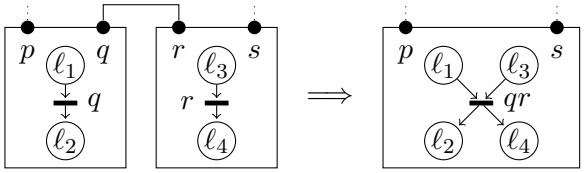

Figure 7.3: Merging components.

into a single one. The interaction qr between the two components is replaced by a single Petri net transition. By merging all components, one obtains a single Petri net that can be implemented as a standalone program since it does not require synchronizations with other components.

7.2.4 Source to Source Decentralization

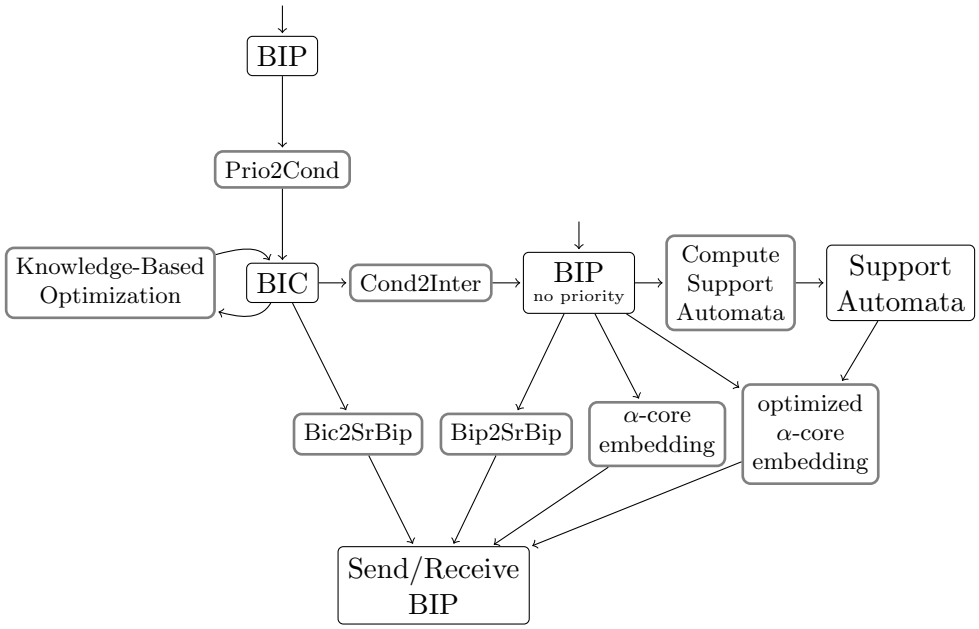

Figure 7.4: Tools involved in the decentralization process.

The methods presented in this thesis have been implemented through a set of tools that we detail here. A global overview of the different options to generate a distributed model from a BIP model is shown on Figure 7.4. Initially, only BIP models without priority where handled, through the following tools:

Bip2SrBip This tool generates a 3-layer Send/Receive model. Condition is not supported. It is parameterized by a partition of the interactions and the choice of the conflict resolution protocol.

α-core embedding This tool generates a Send/Receive BIP model that embeds the α-core protocol, as explained in Section 6.4. There are no parameters except the BIP model. This tool relies on the merging tool to build the final distributed model.

The idea of replacing Priority by Condition has been later introduced to add priority support. Furthermore, Condition can be optimized by using knowledge, as explained in Section 5.4. The following tools are prototypes and handle only Condition predicates containing control locations (i.e. Condition depending on data is not supported).

Prio2Cond This tool simply rewrites priorities as Condition predicates. This transformation is presented in Subsection 4.2.5.

Knowledege-Based Optimization This tool implements the techniques presented in Section 5.4. It relies on DFinder to compute the invariants used as an approximation of the global states. The tool can be used in two ways. First, without

any input, it tries to minimize the set of observed components using the simulated annealing approach. This first pass outputs a set of observed components for each interaction. Second, given a set of observed components for each interaction, it replaces the input Condition predicates with their knowledge approximation according to the observed components.

Cond2Inter This tool transforms a BIC model into an equivalent BIP model that has no priority. The transformation is described in Subsection 4.2.5.

Bic2SrBip This tool was obtained by extending the Bip2SrBip tool to support Condition. Given a BIC model, a partition of the interaction and a conflict resolution protocol, it generates a 3-layer Send/Receive BIP model as described in Section 6.3.

The last idea is to use knowledge for optimizing the conflict resolution protocol. We applied this idea to the α-core protocol through the following tools:

Support Automata Computation This tool computes the support automaton for each atomic component of a BIP model, as described in Section 3.2. During computation, the states of the automaton, which are sets of global states, are represented using Binary Decision Diagrams (BDDs).

optimized α-core embedding This tool was obtained by modifying the α-core embedding tool. From a BIP model without priority and the corresponding support automata, it generates a distributed model embedding the optimized version of α-core described in Section 6.5.

7.2.5 Execution/Simulation

These tools aim to produce code for either simulation or execution of a BIP model. One option is to use a centralized BIP execution Engine, which directly implements the BIP operational semantics. It plays the role of the coordinator in selecting and executing interactions between the components, taking into account the glue specified in the input component model. Executing a BIP model against a centralized engine can be done within a single thread or in a multi-threaded fashion, where the engine constitutes one thread and each component executes in its own thread as well. Finally, we present the distributed code generation, that transforms a Send/Receive BIP model into a set of standalone programs communicating through the primitives available on the target platform.

The BIP Engine for Single Thread Execution

From a BIP model, a compiler is used to generate code, which is currently C++, for atomic components and glue. The code is orchestrated by a sequential engine, implemented in C++ as well, that enforces the BIP operational semantic.

An execution cycle of the centralized Engine is depicted in Figure 7.5. This cycle executes an interaction, moving the system from a global state to another global state.

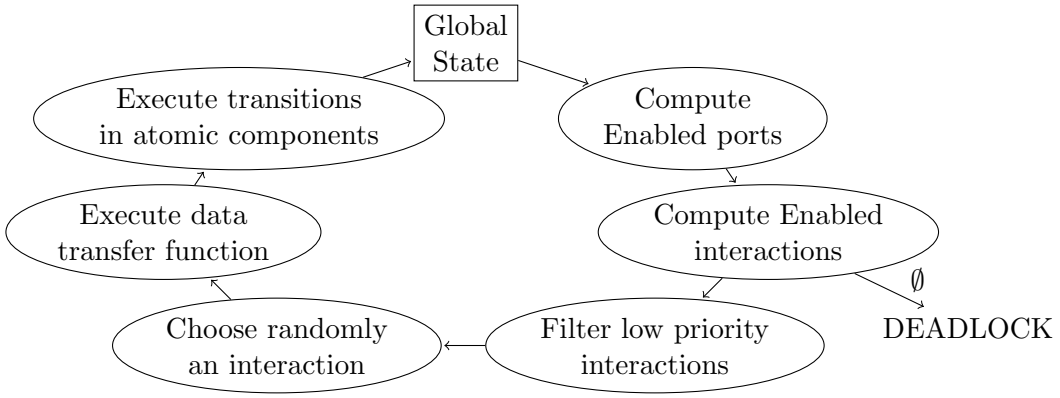

Figure 7.5: Execution of the centralized engine.

The Engine first computes the set of enabled ports for each atomic component. Then, the set of enabled interactions is computed, according to the enabled ports and the guards of the interactions. If no interaction is possible, the Engine stops and indicates that it reached a deadlock. The next step removes interactions that are not maximal for the priority order (among the enabled interactions). Then the data transfer function of the interaction is computed. The remaining computations are the transitions corresponding to ports involved in the interaction. The Engine sequentially executes these transitions in the corresponding atomic components to reach the next global state.

In practice, the code of the BIP model is compiled to run with the centralized implementation of the engine. More precisely, each BIP element (atomic components, connectors, compound components) is implemented as a C++ class. Each of these classes calls functions that constitute the implementation of the centralized engine. In turn, the engine may call functions of the generated code when needed, for instance when executing the data transfer function of an interaction.

BIP Engine for Multi-Threaded execution

The multi-threaded implementation with centralized engine assumes that each component is a separate thread. The engine is executed in another thread. Contrarily to the previous version, the global state is not known to the engine as an atomic component performing an internal computation has an undefined state. Therefore the engine executes according to a partial state semantics [9], that takes into account the fact that the state of some components may be unknown.

As presented in Figure 7.6, the execution of the multi-thread engine is very similar to the execution of the centralized engine. The Engine knows only a partial state, given by the offers received so far. From this set of offers, the multi-thread engine computes the set of enabled interactions. Checking enabledness of an interaction is not enough to ensure that it can execute, as further offers may enable a higher priority interaction. Therefore, the multi-thread engine relies on an *oracle* that must be true for the interaction to execute. A correct oracle allows only interactions such that no offer can enable a higher

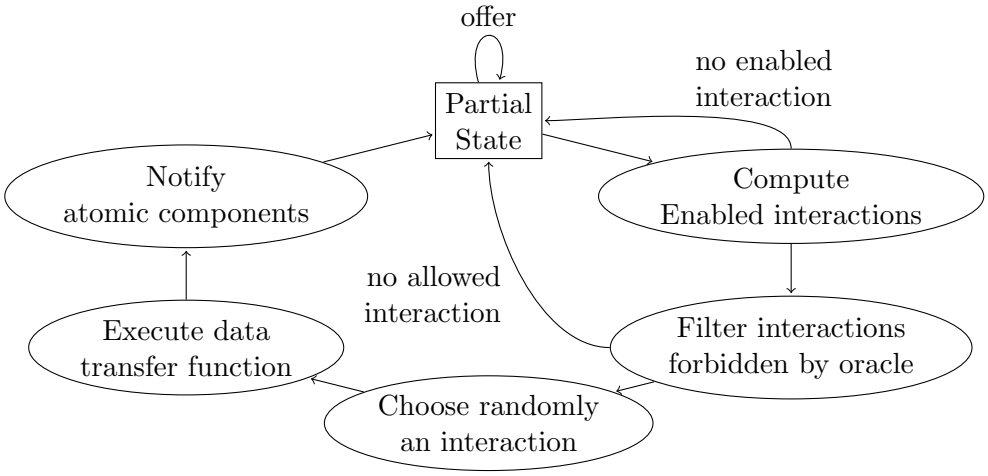

Figure 7.6: Execution of the multi-threaded engine.

priority interaction. The execution of the data transfer function is executed in the engine. Finally, each component executes its update function independently.

As in the centralized code, generated code contains a class for each BIP element. Each atomic component constitutes a separated thread. Communication between the atomic components and the engine (i.e. offers and notifications) is done through function calls. The variables are exchanged using shared memory.

Distributed Execution

The last step for obtaining an executable distributed implementation is to generate code from a Send/Receive model. In such a model, the only interactions between components are message-passing. To obtain an implementation, one generates a separate program for each component, and binds these programs by using the message-passing primitives available on the target platform.

Consider a Send/Receive BIP model. Each component is a Petri net, whose transitions are labeled by ports. The Definition 5.2 allows only three kinds of ports: send ports, receive ports, and unary ports. A send port is used to trigger the emission of a message. Therefore, the code generated from a transition labeled by a send port includes a call to a `send` function. The contents to send is specified by the set of variables attached to the port. According to Definition 5.2, there is a single receive port associated to each send port, which specifies the recipient of the message.

A unary port corresponds to an action in which only the local component is involved. The corresponding generated code does not involve any communication primitives.

The transitions labeled by send and unary ports can be executed as soon as they are enabled, according to the current state of the Petri net. To the contrary, transitions labeled by receive ports can be executed only if there is an incoming message corresponding to that port. Therefore, before executing a transition labeled by a receive port, the

component needs to wait for an incoming message. In order to reduce waiting time, the component starts waiting only if no unary transition or send transition is enabled.

For a given state of the Petri net, there may be several possible incoming messages. For instance, an engine may initially receive any offer. Even if only one receive port is enabled, there might be several potential senders. For instance, an atomic component may have a port involved in two externally conflicting interactions, in which case the engine winning the conflict will send the notification. In that case, the sender depends on who wins the conflict. Therefore, when waiting for a message, the component cannot know from which sender or on which port the message will arrive. This implies to have a `select` function that blocks until an incoming message is detected. The select function indicates to the component where the message comes from. The component then executes the transition labeled by the corresponding receive port. The content of the message is obtained through a call to a `receive` function.

Assuming a platform that provides the three functions presented above, one generates for each atomic component a program whose skeleton is presented in Algorithm 4. First the communication are initialized according to the platform requirements. Then a `while` loop is executed. The loop first checks whether a message can be sent, that is if a send port is enabled. If it is indeed possible, the message is sent and the loops restarts. Otherwise, the loop continues by checking if an internal action is possible. Again, if such an action is possible, it is executed and the loop restarts. Finally, if no send or unary actions is possible, the `select` function is called and blocks until a message arrives. The next incoming message is then received and the loop restarts. Sending messages as soon as they the corresponding send port is enabled ensures progress of the system.

We have written code generators for the following platforms:

POSIX sockets. The generated code can be compiled and run on any POSIX compliant platform. The generated code conforms to the pseudo code. It uses the POSIX socket functions called respectively `send`, `recv` and `select` as implementations of the `send`, `receive` and `select` functions from the pseudo-code. The code generator requires a mapping file assigning an IP address and a TCP port to each component of the Send/Receive BIP model.

POSIX threads. The generated code implements the functions `send`, `receive` and `select` by using shared memory. More precisely, for each atomic component, the implementation contains a shared-memory FIFO buffer, a semaphore, and a mutex. The function `send` is implemented by directly writing the contents of the message in the FIFO buffer of the recipient, and incrementing the semaphore. Each component watches its semaphore to implement the function `select`. Finally, `receive` is done by reading the local buffer. The mutex is used to ensure exclusive access to the FIFO buffer.

MPI. MPI [41], which stands for message-passing interface, is a library reference describing functions to handle message-passing. There are open-source and vendor-specific MPI implementations. Vendor-specific implementations usually optimize

Algorithm 4 Pseudo-code for executing a distributed Send/Receive BIP component.

```
1:  // Initialization
2:  InitializeConnections();
3:  PrepareInitialState();

4:  while true do
5:    // Send messages
6:    if there exists an enabled send port then
7:      send(...);
8:      PrepareNextState();
9:      goto line 4;
10:   end if

11:   // Internal computation
12:   if there exists an enabled unary port then
13:     DoInternalComputation();
14:     PrepareNextState();
15:     goto line 4;
16:   end if

17:   // Receive messages
18:   select(...);
19:   receive(...);
20:   PrepareNextState();
21: end while
```

the library for a specific hardware. We mainly used the OpenMPI implementation [43] which is open-source.

The `MPI_Send` function is used to implement `send`. The `MPI_IRecv` function is used to implement `receive`. Using this implementation of the receive function let the MPI runtime receive the message whenever it arrives, without triggering the transition. The function `select` is then implemented with `MPI_Waitsome`.

ASEBA. There is a prototype code generator targeting ASEBA scripts. ASEBA [64] is a set of tools for programing robots, through a script language. The tool requires to specify, for each component, the micro controller on which the component should be deployed. Communication between the processes is done through D-Bus. We used this tool to generate an ASEBA script from a BIP model. The script was run on a Marxbot robot [25].

7.3 Discussion

The BIP tool set is conceived to use BIP as a common semantic model along the design flow. However, most of the tools presented above are standalone in the sense that they include a parser to construct the BIP model from the textual representation and a BIP code generator to do the opposite. To overcome this, a more modular approach has been used to redevelop some of the tools. This approach relies on a set of front-ends, middle-ends and back-ends that can be combined to form a chain, corresponding to a path in the design flow.

Front-ends include a BIP parser, but should also contain the language factory tools. The middle-ends should be designed to ensure maximal reuse between them. For instance, the knowledge-based tool to minimize the number of observed components utilizes the invariants computed by DFinder. Therefore, one could imagine to have a middle-end in charge of computing invariants. The invariants could then be stored, e.g. as annotations, in the BIP model. The model enriched with invariants could then be used to check deadlock-freedom of the model in a dedicated middle-end. The same model could also be used to perform the knowledge-based optimization in another middle-end. Similarly, for each target platform, a code generator could be implemented as a back-end.

8 Experiments

We present here some experiments to illustrate how our design flow behave. Several transformations and optimizations have been introduced along the previous Chapter. Some of the transformations, for instance the construction of the 3-layer model, are tuned by several parameters, that control the degree of parallelism. The obtained distributed model is generic enough to allow its deployment on several platforms. In Section 8.1, we simulate different platform by adding delays either on computations or on communications. The idea is to see how different protocols and partitions behave when the characteristics of the platform are modified. These experiments have been done with model without priority.

In a second section, we use sorting algorithms that are more computationally intensive examples. In that context, we compare the implementations obtained with different code generators.

Sections 8.1 and 8.2 consider only BIP examples without priority and the corresponding 3-layer model. These first experiments compare different partitions and conflict resolution protocol performance. In Section 8.1, arbitrary waiting times are added to simulate computation times and communication times. In Section 8.2, we use more computationally intensive examples and do not introduce additional waiting times. The Section 8.3 presents how priority rules are handled through Condition and assess the benefits of the different optimizations. The optimization based on knowledge with perfect recall is studied in Section 8.4. Finally, the results are discussed in Section 8.5.

8.1 Simulations

We conducted simulations to study the impact of different choices of conflict resolution protocol and partition of interactions. We emphasize that we distinguish between simulations (results in this section) and an experiments (results in Section 8.2). In distributed systems, the execution of a task or network communication may take a considerable amount of time depending on the underlying platform. Thus, we provide simulations by adding communication delays and computation times to take into account the dynamics of different target platforms. Unlike simulations, all parameters and results in the experiments are determined by real platform characteristics.

We denote each simulation scenario by (i, X), where i is the number of engines and X is one among the three conflict resolution protocols described in Subsection 6.3.3 (i.e., CP, TR, or DP). For the cases where partition of interactions results in having no external conflicts and, hence, requires no conflict resolution, we use the symbol '$-$' to denote absence of conflict resolution protocol. The scenarios considered in this Section are referred to as 'simulations', because we consider a large number of processes running

on a limited number of stand-alone machines. Thus, we model communication delays by temporarily suspending communicating processes.

All simulations are conducted on five quad-Xeon 2.6 GHz machines with 6GB RAM running under Debian Linux and connected via a 100Mbps Ethernet network. Our aim is to show that different conflict resolution algorithms and partitions may result in significantly different performance. In Subsection 8.1.1, we present simulation results for a distributed diffusing computation algorithm. In Subsection 8.1.2, we describe results for a distributed transportation system.

8.1.1 Diffusing Computation

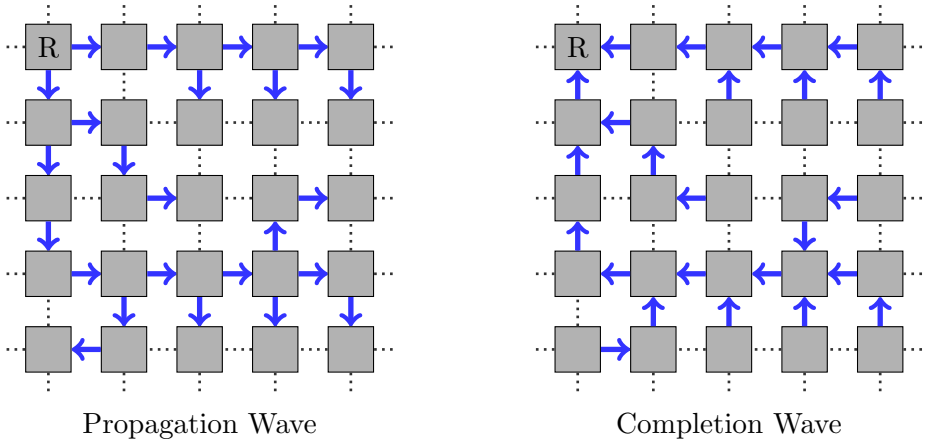

Figure 8.1: Two phases of the Diffusing computation example.

We model a simplified version of Dijkstra-Scholten termination detection algorithm for diffusing computations [38] in BIP. *Diffusing computation* consists in propagating a message across a distributed system; i.e., a wave starts from an initial node and diffuses to all processes in a distributed system. Diffusing computation has numerous applications including traditional distributed deadlock detection and reprogramming of modern sensor networks. One challenge in diffusing computation is to detect its termination. In our version, we consider a torus (wrapped around grid) topology for a set of distributed processes, where a spanning tree throughout the distributed system already exists. Figure 8.1 presents an example of this algorithm on a 5×5 torus. Each process has a unique parent and the root process, depicted by R on the figure, is its own parent. Termination detection is achieved in two phases: (1) the root of the spanning tree possesses a message and initiates a *propagation wave*, so that each process sends the message to its children; and (2) once the first wave of messages reaches the leaves of the tree, a *completion wave* starts, where a parent completes once all its children have completed. When the root has completed, termination is detected.

For a torus of size $n \times m$, the BIP model has $n \times m$ atomic components (see Figure 8.2 for a partial model). Each component participates in two types of interactions:

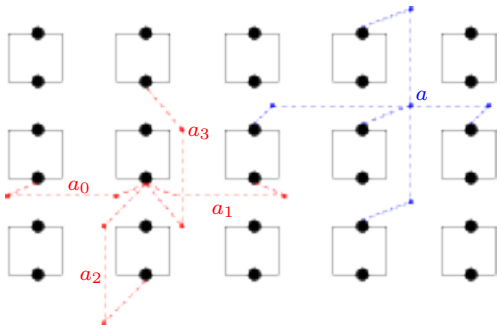

Figure 8.2: Partial BIP model for diffusing computations.

(1) four binary rendezvous interactions (e.g., $a_0 \cdots a_3$) to propagate the message to its children (as in a torus topology, each node has four neighbors and, hence, potentially four children), and (2) one 5-ary rendezvous interaction (e.g., a) for the completion wave, as each parent has to wait for all its children to complete. Finally, in order to make our simulations realistic, we require that execution of each interaction involves $10ms$ suspension of the corresponding engine.

Influence of Partition

Our first set of simulations is for a 6×4 torus. We used different partitions as illustrated in Figure 8.3. Figure 8.4 shows the time needed for 100 rounds for detecting termination of diffusing communication for each scenario. In the first two scenarios, the interactions are partitioned, so that all conflicts are internal and, hence, resolved in the engines. In case $(2, -)$, all interactions of the propagation wave are grouped into one engine and all interactions related to the completion wave are grouped into the second component. Such grouping does not allow parallel execution of interactions. This is the main reason for poor performance of $(1, -)$ and $(2, -)$ shown in Figure 8.4.

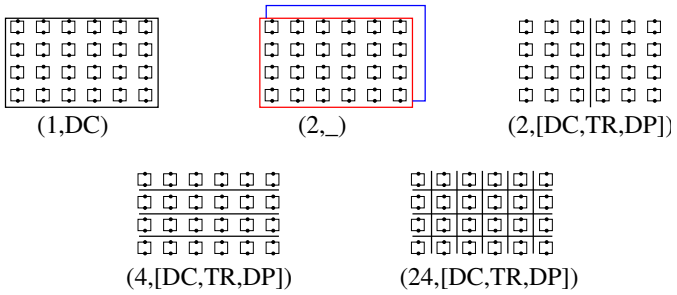

Figure 8.3: Different scenarios for diffusing computations.

Other scenarios group all interactions involved in components $1 \cdots 12$ into one component and the remaining interactions in a second engine. These provide simulations

$(2, CP)$, $(2, TR)$, and $(2, DP)$. Such partitions allow more parallelism during propagation and completion waves, as an interaction in the first class can be executed in parallel with an interaction in the second class. Recall that execution of each interaction involves $10ms$ of suspension of the corresponding engine. This is why $(2, CP/TR/DP)$ outperforms $(1, -)$ and $(2, -)$. As almost all propagation interactions conflict with each other and so do all completion interactions, the conflict graph is dense. Hence, to make a decision within DP, each philosopher needs to grab a high number of forks, which entails a lot of communication. Thus, the performance of $(2, TR)$ is slightly better than $(2, DP)$. It can also be seen that $(2, CP)$ performs as well as $(2, TR)$ and $(2, DP)$. This is due to the fact that we have only two classes, which results in a low number of reservation requests.

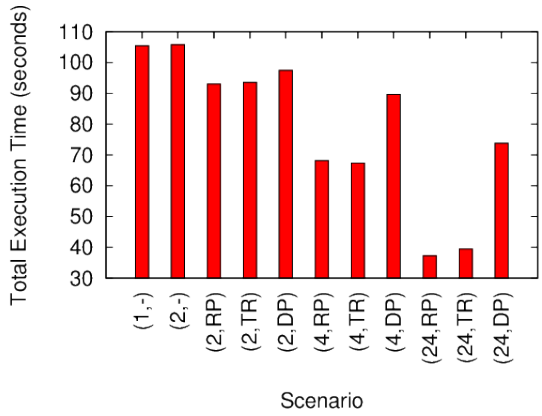

Figure 8.4: Performance of termination detection in diffusing computation in different scenarios for torus 6×4.

Figure 8.4 also shows the same type of simulations for 4 and 24 partition classes. As for two partitions, TR and CP for 4 and 24 partition classes have comparable performance. However, CP and TR outperform DP. This is due to the fact that for DP, each philosopher needs to acquire the forks corresponding to all conflicting interactions, which requires a considerable amount of communication. On the contrary, TR does not require as much communication, as the only task it has to do is releasing and acquiring the token. Moreover, the level of parallelism in DP for a 6×4 torus is not high enough to cope with the communication volume.

Following our observations regarding tradeoff between communication volume and parallelism, we design a scenario illustrating the advantages of DP. Recall that each component in DP resolves conflicts through communication involving only its neighboring components. This is not the case for TR, since the token has to travel through a large number of components. For a 20×20 torus, as can be seen in Figure 8.5, DP outperforms TR. This is solely because, in TR, the token travels along the ring of components and enables interactions sequentially. On the contrary, in DP, the conflict resolution protocol components act in their local neighborhood and although more communication

is needed, a higher level of concurrency is possible and, hence, more interactions can be enabled simultaneously. We expect for increasing size of the torus, *DP* outperforms *CP* as well.

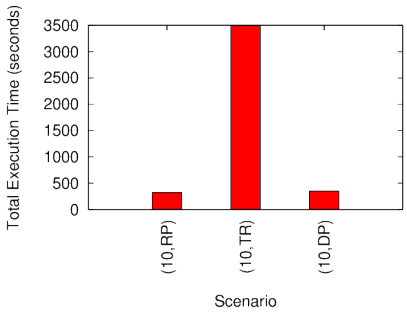

Figure 8.5: Performance of termination detection in diffusing computation in different scenarios for torus 20×20.

Influence of Communication Delays and Execution Times

We now study the influence of communication delays and execution times of functions attached to interactions. We simulate different environments by adding execution times and communication delays. The idea is to provide some guidelines on whether one should use a coarse or fine grain partition and which protocol to choose, depending on the application software and the architecture.

Note that adding execution times within atomic components will slow down the system regardless of the partition and committee coordination algorithm. The response time to a notification is the time needed for the reception of the notification, parallel execution of transitions in atomic components and emission of offer messages. Since we have parallelism between components, the response time to a notification is determined by the response time of the slowest component and does not depend on the partition of interactions or the conflict resolution protocol used. Therefore, we do not model execution times on atomic components transitions, nor the communication delay between atomic components and engines.

We consider three different environments, each of them defined by the following parameters:

- t_{inter} is the execution time for an interaction.
- $t_{E \leftrightarrow CP}$ is the communication delay between the engines and the conflict resolution protocol.
- $t_{CP \leftrightarrow CP}$ is the communication delay between components inside the conflict resolution protocol.

In the first environment, we assume $t_{inter} = 10ms$ for an interaction execution time, as for the previous simulation, and no communications delay ($t_{E \leftrightarrow CP} = t_{CP \leftrightarrow CP} = 0ms$). In the second environment, we still assume the same execution time and we add a delay of $t_{E \leftrightarrow CP} = 10ms$ for communication between the engines and the conflict resolution protocol. In the third environment, we assume slower processors with $t_{inter} = 100ms$ interaction execution time. Furthermore, we assume $t_{E \leftrightarrow CP} = 10ms$ for communications between the engines and the conflict resolution protocol and $t_{CP \leftrightarrow CP} = 1ms$ for communications inside the conflict resolution protocol.

For each of these environments, we executed a different scenario of diffusing computation built on a 5×5 grid. We used three different partitions of the 5×5 torus: a centralized one, a partition with 5 engines (similar to the one with 4 engine components depicted in Figure 8.3), and the fully decentralized one, with 25 engines. The total execution time of these scenarios in the three environments described above, are shown in Figure 8.6, Figure 8.7, and Figure 8.8 respectively.

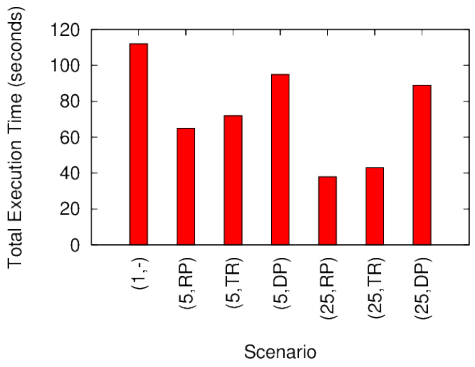

Figure 8.6: Simulation of a diffusing computation on a 5×5 torus. $t_{inter} = 10ms$, $t_{E \leftrightarrow CP} = 0ms$, $t_{CP \leftrightarrow CP} = 0ms$

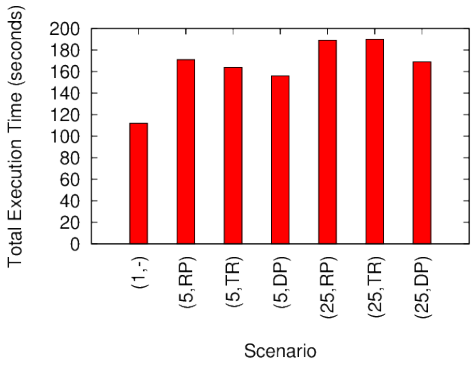

Figure 8.7: Simulation of a diffusing computation on a 5×5 torus, $t_{inter} = 10ms$, $t_{E \leftrightarrow CP} = 10ms$, $t_{CP \leftrightarrow CP} = 0ms$

Notice that in the first environment, the best performance is achieved for the most

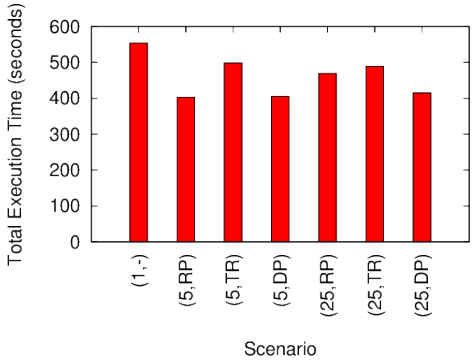

Figure 8.8: Simulation of a diffusing computation on a 5×5 torus, $t_{inter} = 100ms$, $t_{E \leftrightarrow CP} = 10ms$, $t_{CP \leftrightarrow CP} = 1ms$

decentralized partition, as in Figure 8.4, since there are no communication delays. In the second environment, where we made the assumption that one communication and one interaction execution require the same time, the best performance is obtained for the centralized solution. In this case, the communication between the engines and conflict resolution protocol is too expensive compared to the speedup obtained by having interactions running in parallel. Finally, in the third environment, where the communication costs 10 times less than executing an interaction, the solution with 5 interaction protocol components gives the best performance, except for the token ring protocol. Indeed, the bottleneck is the time needed for the token to cycle through all conflict resolution protocol components. Having more engines increases the number of pending reservations and thus diminishes the time between two granted reservations. Increasing the communication time $t_{RP \leftrightarrow RP}$ (experiments not shown here) penalizes token ring much more than dining philosophers.

8.1.2 Utopar Transportation System

Utopar is an industrial case study proposed in the context of the European Integrated Project SPEEDS[1]. Utopar is an automated transportation system managing requests for transportation. The system consists of a set of autonomous vehicles, called U-cars, a centralized automatic control (Central-Station), and calling units (see Figure 8.9).

We model a simplified version of Utopar in BIP. The overall system architecture is depicted in Figure 8.10. It is a composition of an arbitrary (but fixed) number of components of three different types: U-Cars, Calling-Units, and Central-Station. The Utopar system interacts with external users (passengers). Users are also represented as components, however, their behavior is not explicitly modeled.

The overall behavior of the system is obtained by composing the behavior of the components using the following set of interactions:

- *awake*: handling awake calls of cars by Central-Station;

[1] `http://www.speeds.eu.com/`

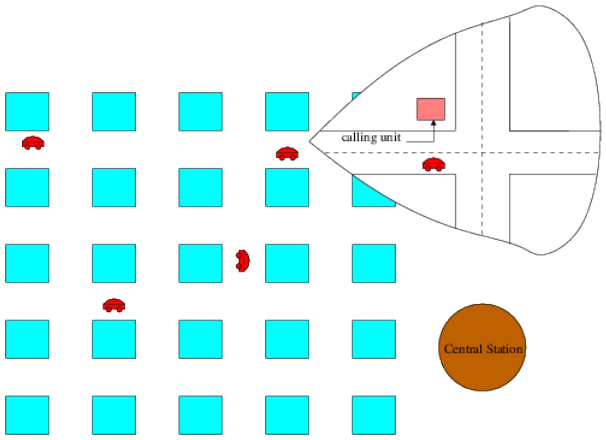

Figure 8.9: Utopar transportation system.

- *request*: handling car requests by users at Calling-Units;
- *destination*: handling destination requests by users seating within U-Cars;
- *enter*: handling the step on (resp. off) of users into U-Cars;
- *departure*: handling departure commands issued by Central-Station towards U-Cars;
- *posChanged*, *arrival*: information provided by moving U-Cars towards Central-Station;
- *open*, *close*: handling the opening/closing of U-Cars doors, while parked at Calling-Units.

Our first set of simulations consists of $25 = 5 \times 5$ calling units and 4 cars. For each calling unit, we group all the interactions it is involved, in one engine. Moreover, for each car, we define one engine handling all interactions involving the central unit. Thus, we obtain 29 engines. Using this partition, we generate the corresponding 3-layer Send/Receive model for the three conflict resolution protocols. We simulate the target platform as follows. We consider that there exists a machine on each calling unit. Moreover, each machine is connected to its four neighbours and the communication time between two neighbour machines is 1ms.

We generate the corresponding C++ executables as follows. We merge each calling unit with its associated engine and map the resulting code into the machine located on the calling unit. In a similar way, we merge each car with its associated engine and map the resulting code into an arbitrary machine, such that two different cars are located in different machines. Regarding CentralStation, we map its code into the central (midmost) machine. Regarding the conflict resolution protocol, in the case of *CP*, the best choice is to map its code into the central machine as well. In the case of *TR* and

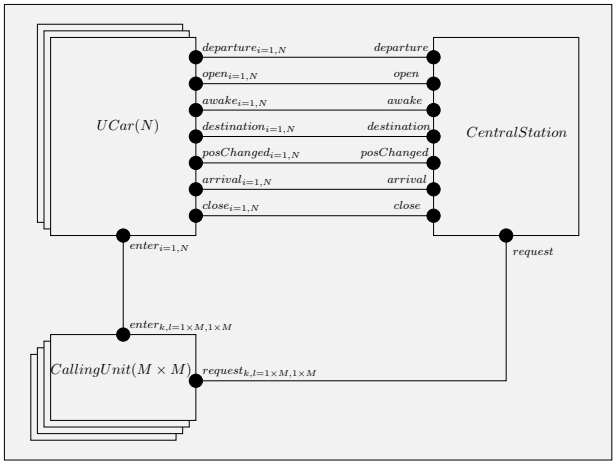

Figure 8.10: A BIP model for Utopar system.

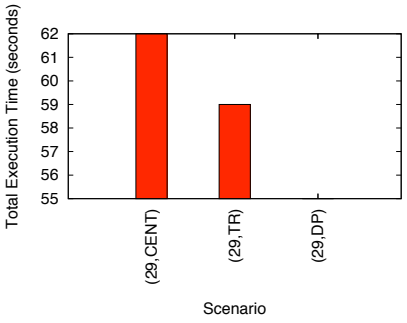

Figure 8.11: Performance of responding 10 requests per calling unit in Utopar System in different scenarios for 5×5 calling units and 4 cars.

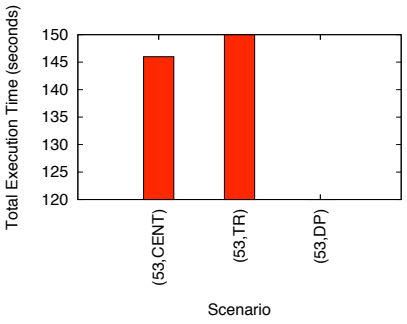

Figure 8.12: Performance of responding 10 requests per calling unit in Utopar System in different scenarios for 7×7 calling units and 4 cars.

DP protocols, we map the code of each component in the same machine as the engine communicating with it.

Figure 8.11 shows the time needed for responding to 10 requests by each calling unit. Clearly, $(29, DP)$ outperforms $(29, TR)$ and $(29, CP)$. This is due to the overhead of communications for the case of TR and CP. More precisely, regarding CP the overhead is due to the communication between the engines and the conflict resolution protocol, since the centralized conflict resolution protocol is placed in the central machine. Regarding TR, the overhead is due to communications between conflict resolution protocol which depend on the number of components in this layer. To the contrary, in DP, the conflict resolution protocol components act in their local neighborhood although more communication is needed.

Figure 8.12 also shows the same type of simulations by taking 4 cars and $49 = 7 \times 7$ calling units. Performance becomes worse for TR since the token has to travel a long way through the components of the conflict resolution protocol layer.

We conclude this section by stating the main lesson learned from our simulations. Different partitions and choice of committee coordination algorithm for distributed conflict resolution, suit different topologies and settings although they serve a common purpose. Designers of distributed applications should have access to a library of algorithms and choose the best according to parameters of the application.

8.2 Running Experiments

In this section, we present the results of our experiments on two popular parallel sorting algorithms: (1) network sorting algorithm (see Subsection 8.2.1), and (2) bitonic sorting

(see Subsection 8.2.2). Unlike simulations in Section 8.1, all results in this section are determined by physical computation and communication times.

All experiments are conducted on quad-Xeon 2.6 GHz machines with 6GB RAM running under Debian Linux and connected via a 100Mbps Ethernet network. We show that our method allows evaluation of parallel and multi-core applications modeled in BIP. Moreover, we show that different mergings of components may result in significantly different performance.

8.2.1 Network Sorting Algorithm

We consider 2^n atomic components, each containing an array of N items. The goal is to sort the items, so that all the items in the first component are smaller than those of the second component and so on. Figure 8.13 shows a BIP model of the Network Sorting Algorithm [2] for $n = 2$ using incremental and hierarchical composition of components. The atomic components $B_1 \dots B_4$ are identical. Each atomic component computes independently the minimum and the maximum values of its array. Once this computation completes, interaction a_1 compares the maximum value of B_1 with the minimum value of B_2 and swaps them if the maximum of B_1 is greater than the minimum of B_2. Otherwise, the corresponding arrays are correctly sorted and interaction a_2 gets enabled. This interaction exports the minimum of B_1 and the maximum of B_2 to interaction a_5. The same principle is applied to components B_3 and B_4 and interactions a_3 and a_4. Finally, interaction a_5 works in the same way as interaction a_1 and swaps the minimum and the maximum values, if they are not correctly sorted.

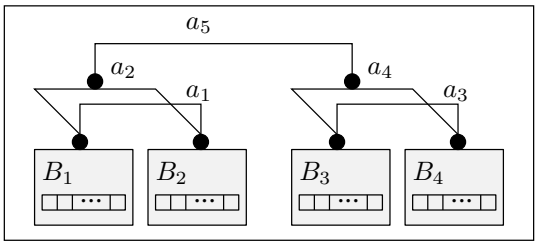

Figure 8.13: A BIP model for Network Sorting Algorithm.

All interactions in Figure 8.13 are in conflict. We choose to construct a single engine E_1 that encompasses all these interactions (see Figure 8.14). We try different merging schemes in order to study the degree of parallelism for each scenario. More precisely, we run experiments for five configurations $1c$, $2c$, $3c$, $4c$, and $5c$. For $1c$, we *merge* all components to obtain a single component, as described in Subsection 7.2.3. For $2c$, we merge components $[B_1^{SR}, B_2^{SR}, E_1]$ and $[B_3^{SR}, B_4^{SR}]$, obtaining two Send/Receive components. For $3c$, we merge components $[B_1^{SR}, E_1]$, $[B_2^{SR}, B_3^{SR}]$, and $[B_4^{SR}]$, obtaining three Send/Receive components. For $4c$, we merge components $[B_1^{SR}, E_1]$, $[B_2^{SR}]$, $[B_3^{SR}]$, and $[B_4^{SR}]$, obtaining four Send/Receive components. Finally, for $5c$ we do not merge components, hence, we have five Send/Receive components (see Figure 8.14).

Table 8.1 shows performance of the automatically generated C++ code using POSIX sockets and POSIX threads. The implementation using POSIX threads slightly outperforms POSIX sockets. This is due to the fact that the number of messages exchanged per component is huge, making the socket-based implementation slower, as it requires network communication. Performance clearly depends on the size of the input array as well. Also, notice that configuration $5c$ outperforms all the other configurations. Indeed, the computation load is much higher than the communication load. The added communication does not entail the performance gained by distributing the computation load.

	$1c$	$2c$	$3c$	$4c$	$5c$
k	C++/Threads				
1	1.78	1.61	0.93	0.73	**0.53**
5	9.72	8.75	5.08	4.21	**2.88**
10	21.52	19.42	11.21	9.42	**6.52**
50	191.14	171.12	98.07	82.96	**63.43**
k	C++/Socket				
1	1.8	1.94	1.27	1.07	**0.83**
5	9.81	9.44	5.85	4.91	**3.53**
10	21.7	20.55	12.56	10.03	**7.51**
50	191.47	177	104.1	86.17	**65.84**

Table 8.1: Performance (execution time in seconds) of NSA ($n = 2$), where $k \times 10^3$ is the size of array to sort.

8.2.2 Bitonic Sorting

Bitonic sorting [14] is one of the fastest sorting algorithms suitable for distributed implementations or in parallel processor arrays. A sequence is called *bitonic* if it is initially non-decreasing, then it is non-increasing. The first step of the algorithm consists in constructing a bitonic sequence. Then, by applying a logarithmic number of bitonic merges,

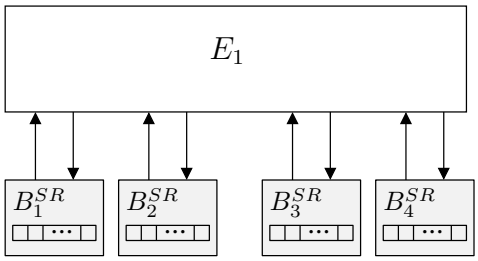

Figure 8.14: 3-layer Send/Receive BIP model for Network Sorting Algorithm.

the bitonic sequence is transformed into a totally ordered sequence. We provide an implementation of the bitonic sorting algorithm in BIP using four atomic components, each handling one quarter of the array. These components are connected as shown in Figure 8.15.

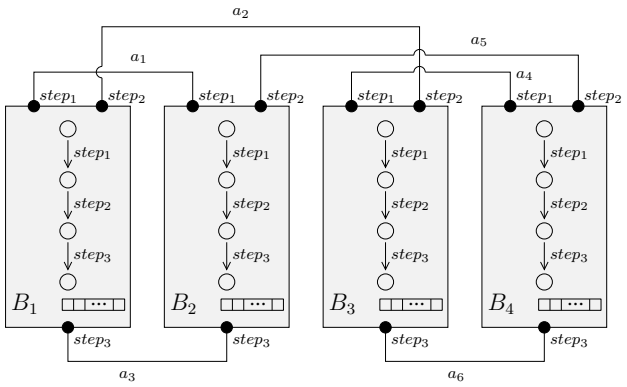

Figure 8.15: A BIP model for Bitonic Sorting Algorithm.

The six interactions are non-conflicting. Moreover, interactions a_1, a_2, and a_3 cannot run in parallel. The same holds for interactions a_4, a_5, and a_6. Thus, two engines suffice to obtain maximal parallelism between interactions. The first engine E_1 handles interactions a_1, a_2, and a_3 and the second engine E_2 handles interactions a_4, a_5, and a_6. Furthermore, since all interactions are non-conflicting, there is no need for conflict resolution protocol. According to this partition of interactions, we obtain the 3-layer Send/Receive BIP model shown in Figure 8.16. In this example, each component sends only three messages containing the array of values.

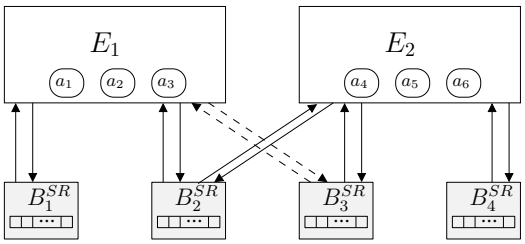

Figure 8.16: 3-layer Send/Receive BIP model for Bitonic Sorting Algorithm (6c).

We choose different merging schemes to study the degree of parallelism with respect to each configuration. More precisely, we run experiments for four configurations 1c, 2c, 4c, and 6c (see Figure 8.17). For 1c, we merge all components in one component. For 2c, we merge components $[B_1^{SR}, B_2^{SR}, E_1]$ and $[B_3^{SR}, B_4^{SR}, E_2]$, obtaining two components. For 4c, we merge components $[B_1^{SR}, E_1]$, $[B_2^{SR}]$, $[B_3^{SR}]$, and $[B_2^{SR}, E_2]$, obtaining four components. Finally, for 6c we do not merge components, hence, we have six Send/Receive components (see Figure 8.16).

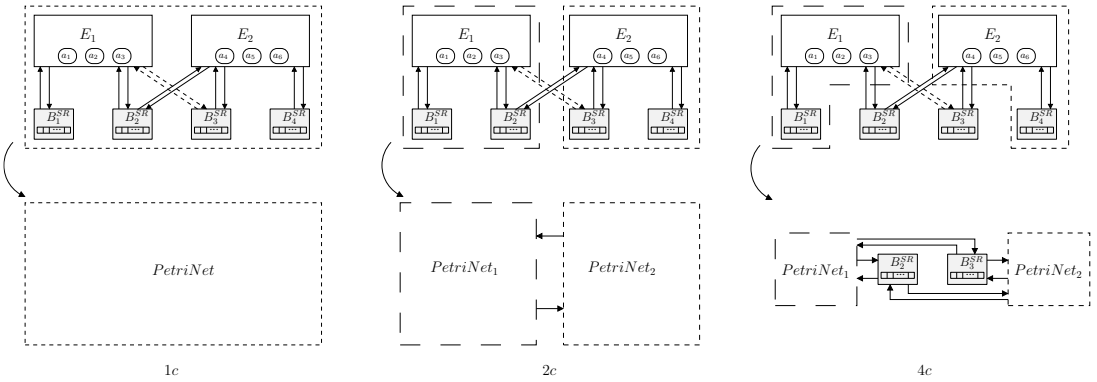

Figure 8.17: Merging schemes applied on Send/Receive BIP model of Bitonic Sorting Algorithm.

Table 8.2 shows performance of the automatically generated C++ code using POSIX sockets, POSIX threads (shared memory), and MPI. Clearly, the configuration 6*c* outperforms the other configurations for POSIX sockets and for POSIX threads. Furthermore, the overall performance of these implementations is quite similar. This is due to the fact that, in contrast to the previous example, the number of messages exchanged per component is small. More precisely, each component performs three steps in order to obtain a totally ordered sequence. Each step requires a binary synchronization and leads to one message exchange between an atomic component and an engine. On the other hand, at each step the amount of computation per atomic component is huge with respect to the communication time.

For the MPI implementation, configuration 4*c* outperforms the other configurations. This is due to the fact that MPI uses active waiting, which entails CPU time consumption when a component is waiting. The MPI code consisting of four processes is therefore the best fitting the four cores available on the target machine and yields best performances.

Moreover, Table 8.2 shows performance of the handwritten C++ code using MPI collective communication primitives (e.g., Gather and Scatter) instead of Send/Receive to transfer data. We notice that the best performance of automatically generated C++ code (obtained in configuration 6*c*) is comparable to the performance of handwritten code (and run on configuration 4*c*).

The main observation from these experiments is that determining adequate component merging and communication primitives depends on (1) the topology of the system with respect to communication delays, and (2) the computational load of the system, and (3) the target architecture on which the system is deployed.

8.3 Condition

We compare the execution time and the number of exchanged messages for several distributed implementations of a BIC component. This component is obtained from a

	1c	2c	4c	6c
k	C++/Threads			
1	0.02	**0.01**	**0.01**	**0.01**
10	1.8	0.96	0.75	**0.54**
50	44	23.57	18.37	**12.04**
100	178.42	94.71	73.22	**48.1**
k	C++/Socket			
1	**0.02**	**0.02**	0.34	0.27
10	1.8	1	1.01	**0.75**
50	44	24.1	18.57	**12.3**
100	178.42	95.32	74.01	**48.7**
k	MPI			
1	**0.26**	0.8	1.15	1.16
10	1.93	2.06	**1.92**	2.03
50	44.85	24.04	**19.47**	23.08
100	179.4	95.83	**74.59**	85.37
k	Handwritten			
1			1.54	
10			2.01	
50			13.47	
100			49.67	

Table 8.2: Performance (execution time in seconds) of bitonic sorting ($n = 2$), where $k \times 10^3$ is the size of array for sorting.

BIP component with priority as described in Figure 8.18. The transformation called "KB Opt. basic" and "KB Opt. complete" are done by calling the Knowledge-Based optimization tool from Figure 7.4 and asking to obtain either a basic or a complete BIC model. This first transformation from a BIP to a BIC model is parameterized by:

- The level of optimization (no opt, basic or complete)
- The invariant used for the computation (boolean or linear invariant)

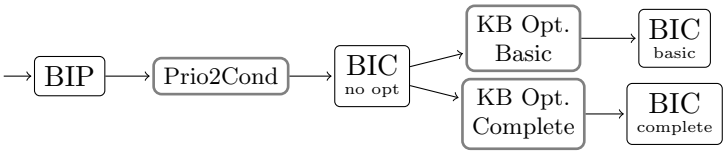

Figure 8.18: Sequence to generate BIC models.

Given a BIC model, regardless whether it is optimized or not, we consider two different implementations:

- A Condition-aware (CA) implementation obtained by directly calling the Bic2SrBip tool.
- A Multiparty-based (MB) implementation obtained by calling the tool Cond2Inter and then Bip2SrBip (See Figure 7.4).

For both cases, we use the centralized version of the conflict resolution protocol.

8.3.1 Dining Philosophers

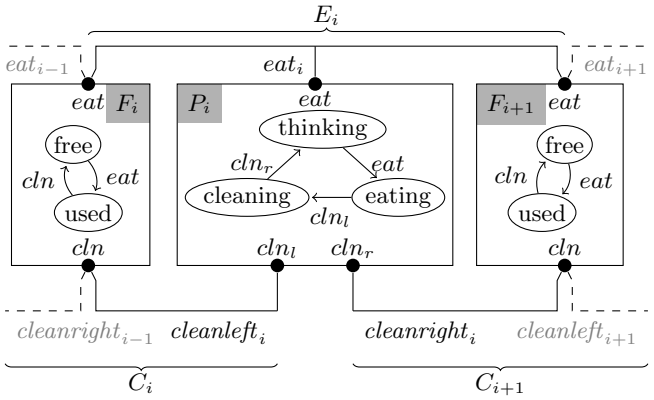

Figure 8.19: Fragment of the dining philosopher component. Braces indicate how interactions are grouped into engines.

We consider a variation of the dining philosophers problem, denoted by Philo$N$ where N is the number of philosophers. A fragment of this composite component is presented in Figure 8.19. In this component, an "eat" interaction eat_i involves a philosopher and the two adjacent forks. After eating, philosopher P_i cleans the forks one by one ($cleanleft_i$ then $cleanright_i$). We consider that each eat_i interaction has higher priority than any $cleanleft_j$ or $cleanright_j$ interaction.

This example has a particularly strong priority rule. Indeed, executing one "*clean*" interaction potentially requires to check that *all* "*eat*" interactions are disabled, that is to observe all components. This example allows the different implementations to be compared under strong priority constraints.

As explained in Section 6.3, the construction of our distributed implementation is structured in 3 layers. The second layer is parameterized by a partition of the interactions. For this example, the partition is built as follows. There is one engine E_i for every eat_i interaction and one engine C_i for every pair $cleanright_{i-1}$, $cleanleft_i$. Only the latter deals with low priority interactions and need to observe more components than the participants in the interactions it manages.

Minimizing Observed Components

As interactions are grouped into engines, we assume that they share their observations. If interactions a and b are handled by the same engine, we impose that $\mathcal{L}_a = \mathcal{L}_b$. This restriction actually assigns a set of observed components to each engine, which limits the size of the solutions space.

Minimizing the number of observed component in a complete Condition κ' is done independently for each engine. Table 8.3 shows the results, that is the number of observed components, obtained for the engine C_0 with the heuristic for ensuring completeness described in Subsection 5.4.2. The total number of atomic components in the composite component is indicated in Column $Size$. Columns $true$, BI and LI provide the cost of the solutions obtained when using respectively $true$, the boolean invariant and the linear invariant as over approximation of the global states. Note that the cost of the solution is the number of components that are observed by at least one interaction of C_0 and not participant in any of the interactions handled by C_0. For instance, the component P_0 being observed by interaction $cleanright_{N-1}$ (handled by C_0) does not increase the cost as P_0 is already participant in C_0 through the interaction $cleanleft_0$. Using $true$ as invariant does not allow actual optimization, therefore it shows the number of observed component in the initial Condition κ. The column *optimal* indicates the cost of an optimal solution.

Here, the linear invariant gives better results than the boolean invariant, which is not precise enough to allow reducing observation comparatively to the $true$ invariant. For $N = 3$, we provide the boolean and linear invariants respectively in Figures 8.20 and 8.21. As an example, consider the linear constraint (8.15). It ensures that interaction $cleanleft_0$ and interaction eat_1 cannot be enabled concurrently, otherwise, control locations $P_0.eating$ and $F_1.free$ would be active and the sum in constraint (8.15) would be equal to 2. Thus, the priority $cleanleft_0 \,\pi\, eat_1$ never forbids execution of $cleanleft_0$. A

Eng.	**Component**	*Size*	*true*	*BI*	*LI*	*optimal*
C_0	Philo3	6	3	3	1	1
	Philo4	8	5	5	2	2
	Philo5	10	7	7	3	3
	Philo10	20	17	17	8	8
	Philo20	40	37	37	18	18
	Philo100	200	197	197	108	98

Table 8.3: Minimal observation for completeness.

Component	*Size*	*true*	*BI*	*LI*
Philo3	6	9	9	0
Philo4	8	20	20	4
Philo5	10	35	35	6
Philo10	20	170	170	23

Table 8.4: Minimal observation for baseness.

related boolean constraint, that is constraint (8.3) of boolean invariant guarantees that at least one of these locations is active. However, this constraint is not strong enough to discard the case where two of them are active.

In general, the approximations of reachable states provided by boolean and linear invariants are not comparable. Consider the global state $P_0.cleaning \wedge F_0.used \wedge P_1.cleaning \wedge F_1.used \wedge P_2.cleaning \wedge F_2.used$. This state satisfies all the constraints of the linear invariant, but does not satisfy the constraint 8.8 of the boolean invariant.

$$\forall i \in \{0,1,2\}\ (at_{F_i.free} \vee at_{F_i.used}) \tag{8.1}$$
$$\wedge\quad \forall i \in \{0,1,2\}\ (at_{P_i.thinking} \vee at_{P_i.eating} \vee at_{P_i.cleaning}) \tag{8.2}$$
$$\wedge\quad (at_{P_1.eating} \vee at_{P_0.eating} \vee at_{P_0.cleaning} \vee at_{F_1.free}) \tag{8.3}$$
$$\wedge\quad (at_{P_2.eating} \vee at_{P_1.eating} \vee at_{P_1.cleaning} \vee at_{F_2.free}) \tag{8.4}$$
$$\wedge\quad (at_{P_0.thinking} \vee at_{F_0.used} \vee at_{P_0.cleaning} \vee at_{P_2.thinking}) \tag{8.5}$$
$$\wedge\quad (at_{P_0.thinking} \vee at_{F_1.used} \vee at_{P_1.cleaning} \vee at_{P_1.thinking}) \tag{8.6}$$
$$\wedge\quad (at_{P_2.cleaning} \vee at_{F_0.free} \vee at_{P_2.eating} \vee at_{P_0.eating}) \tag{8.7}$$
$$\wedge\quad (at_{F_1.free} \vee at_{F_2.free} \vee at_{F_0.free} \vee at_{P_1.eating} \vee at_{P_2.eating} \vee at_{P_0.eating}) \tag{8.8}$$
$$\wedge\quad (at_{F_2.used} \vee at_{P_2.cleaning} \vee at_{P_1.thinking} \vee at_{P_2.thinking}) \tag{8.9}$$
$$\wedge\quad (at_{F_2.used} \vee at_{P_2.cleaning} \vee at_{P_1.thinking} \vee at_{F_0.free} \vee at_{P_0.eating}) \tag{8.10}$$
$$\wedge\quad (at_{F_1.free} \vee at_{P_1.eating} \vee at_{F_0.used} \vee at_{P_0.cleaning} \vee at_{P_2.thinking}) \tag{8.11}$$
$$\wedge\quad (at_{P_0.thinking} \vee at_{F_2.free} \vee at_{F_1.used} \vee at_{P_2.eating} \vee at_{P_1.cleaning}) \tag{8.12}$$

Figure 8.20: Boolean invariant for the Dining Philosophers example with $N = 3$.

The results for computing basic solutions are presented in Table 8.4. The column

$$(at_{P_0.thinking} + at_{P_0.eating} + at_{P_0.cleaning} = 1) \quad (8.13)$$
$$\wedge \quad \forall i \in \{0,1,2\}(at_{F_i.free} + at_{F_i.used} = 1) \quad (8.14)$$
$$\wedge \quad (at_{P_1.eating} + at_{P_0.eating} + at_{P_0.cleaning} + at_{F_1.free} = 1) \quad (8.15)$$
$$\wedge \quad (at_{P_1.thinking} - at_{P_0.eating} - at_{P_0.cleaning} + at_{F_1.used} + at_{P_1.cleaning} = 1) \quad (8.16)$$
$$\wedge \quad (at_{P_2.eating} - at_{P_0.eating} - at_{P_0.cleaning} + at_{F_1.used} + at_{P_1.cleaning} - at_{F_2.used} = 0) \quad (8.17)$$
$$\wedge \quad (at_{P_2.cleaning} + 2 * at_{P_0.eating} + at_{P_0.cleaning} - at_{F_1.used} - at_{P_1.cleaning} + at_{F_2.used} - at_{F_0.used} = 0) \quad (8.18)$$
$$\wedge \quad (at_{P_2.thinking} - at_{P_0.eating} + at_{F_0.used} = 1) \quad (8.19)$$

Figure 8.21: Linear invariant for the Dining Philosophers example with $N = 3$.

Size contains the total number of atomic components in the composite component. The columns *true*, *BI* and *LI* contains respectively the cost of the solutions obtained when using respectively *true*, the boolean invariant and the linear invariant. For Philo3, baseness is achieved when each engine observes only the components involved in the interactions it handles (i.e. no additional atomic component), therefore the cost is 0.

Comparing Obtained Implementations

The goal of this subsection is to compare the different implementations that we obtained for the dining philosophers example. First, we consider different levels of optimization for the Condition operator:

- **No optimization**: the Condition operator is the direct rewriting of priorities rules, we do not apply any knowledge-based optimization.
- **Basic**: observation required by the Condition operator is minimized while still ensuring baseness.
- **Complete**: observation required by the Condition operator is minimized while still ensuring completeness.

As showed in the previous subsection, the Boolean invariant is not strong enough to reduce the number of observed components comparatively to the non-optimized version. Therefore, the basic and complete version of the Condition operator have been computed using the linear invariant. For each optimization level considered, we generate a Multiparty-based (MB) and a Condition-aware (CA) implementation. Once we have built the distributed components, we use a code generator that generates a standalone C++ program for each atomic component. These programs communicate by using Unix sockets.

The obtained code has been run on a UltraSparc T1 that allows parallel execution of 24 threads. For each run, we count the number of interactions executed and messages exchanged in 60 seconds, not including the initialization phase. For each instance we consider the average values obtained over 20 runs. The number of interactions executed

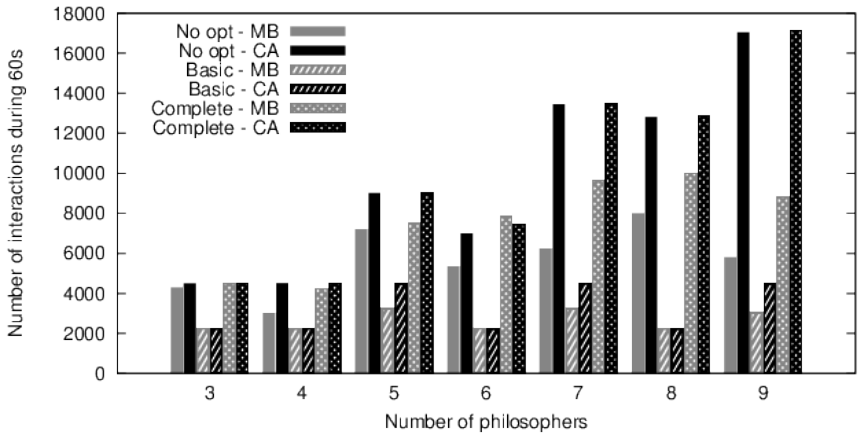

Figure 8.22: Number of interactions executed in 60s for different implementations of the dining philosophers example. MB: Multiparty-based. CA: Condition-aware.

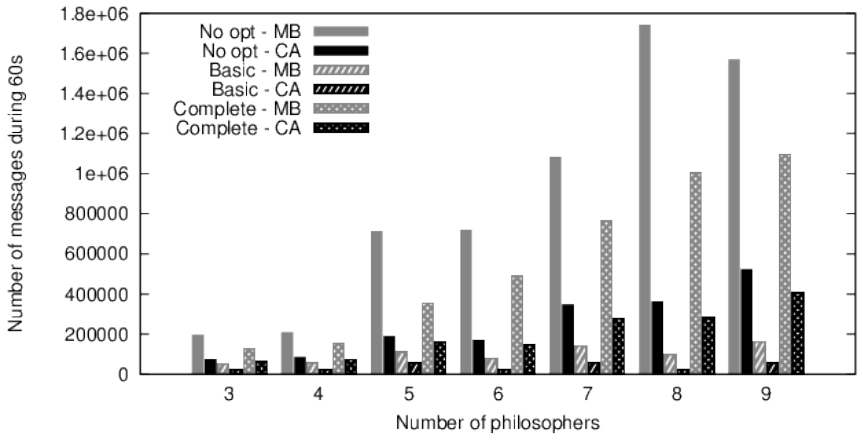

Figure 8.23: Number of messages exchanged in 60s for different implementations of the dining philosophers example. MB: Multiparty-based. CA: Condition-aware.

by each implementation is presented in Figure 8.22. The total number of messages exchanged for the execution of each implementation is presented in Figure 8.23.

First, remark that switching from a Multiparty-based (gray) to a Condition-aware (black) implementation improves performance, that is the number of interactions executed in 60 seconds. Furthermore, it always reduces the number of messages exchanged. The improvement is very visible with the unoptimized version (No opt). This can be explained as follows. Evaluation Condition predicates requires to observe all components for executing a $cleanleft_i$ or a $cleanright_i$ interaction. In the Multiparty-based implementation, observed components must synchronize to execute some interaction $cleanleft_i$ or $cleanright_i$. Between two "*clean*" executions, each component has to receive a notification and to send a new offer. This strongly restricts the parallelism. In the observation-aware implementation, a component offer is still valid after execution of an interaction observing that component. After a "*clean*" interaction, only components that participated may need to send a new offer before another "*clean*" interaction can be executed. This explains the speedup.

Second, when comparing Multiparty-based (gray) implementations, one sees that the Condition operator ensuring completeness gives the best performance. The basic implementations exhibit poor performance because restricting observation in that case also restricts parallelism. For the example with 9 philosophers, Multiparty-based implementation with optimized Condition (Complete - MB) shows a significant gain in performance compared to the non optimized version (No opt - MB). The performance gained by optimizing the Condition operator into a complete one is not visible anymore when switching to Condition-aware (black) implementation. However, the optimization remains interesting in that case since it reduces the number of messages needed for 60 seconds of execution.

8.3.2 Jukebox

The second example is a jukebox depicted in Figure 8.24. It is a more advanced version than our previous running example that has two parallel jukeboxes allowing two discs to be loaded at a time. It represents a system, where a set of readers $R_1 \dots R_4$ access data located on 3 discs D_1, D_2, D_3. Readers may need to access any disc. Access to discs is managed by jukeboxes J_1, J_2 that can load any disc to make it available to the connected readers. The interaction $load_{i,k}$ (respectively $unload_{i,k}$) allows loading (respectively unloading) the disc D_i in the jukebox J_k. Each reader R_j is connected to a jukebox through the $read_j$ interaction. Once a jukebox has loaded a disc, it can either take part in a "read" or "unload" interaction. Each jukebox repeatedly loads all N discs in a random order.

If unload interactions are always chosen immediately after a disc is loaded, then readers may never be able to read data. Therefore, we add the priority $unload_{i,k} \,\pi\, read_j$, for all i, j, k. This ensures that "read" interactions will take place before corresponding discs are unloaded. Furthermore, we assume that readers connected to J_1 need more often disc 1 and that readers connected to J_2 need more often disc 2. Therefore, loading these discs in the corresponding jukeboxes is assigned higher priority: $load_{i,1} \,\pi\, load_{1,1}$ for $i \in \{2, 3\}$

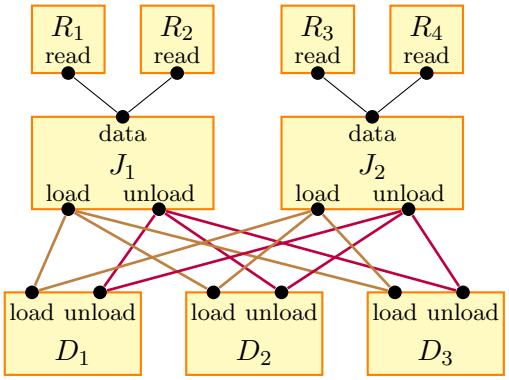

Figure 8.24: Jukebox component with 3 discs.

and $load_{i,2} \, \pi \, load_{2,2}$ for $i \in \{1, 3\}$. Each interaction is handled by a dedicated engine.

The main difference with the dining philosopher examples is that here priority rules do not restrict parallelism since they are expressed between interactions that are in Interaction conflict. Here a priority rule is used to express a scheduling policy that aims to improve the efficiency of the system, in terms of "*read*" interactions. Removing this priority rule results in a system that does less "*read*" interactions.

Minimizing Observed Components

Results of the simulated annealing heuristic are presented in Table 8.5. Engines handling a "read" interaction do not need to observe additional atomic components since there is no interaction with higher priority. The boolean invariant allows removing some observed atomic components, in the basic solution. As for Philo$N$ components, the linear invariant is stronger than the boolean invariant. Therefore, attaining the same level of detection requires less observed atomic components.

Interaction	*true*	BI(basic)	BI(complete)	LI(basic)	LI(complete)
$unload_{i,k}$	5	$3(k = 1)$ or $5(k = 2)$	5	2	2
$load_{i,k}$	1	0	1	0	1

Table 8.5: Minimal observation cost to ensure baseness or completeness.

Comparing Obtained Implementations

For this example, knowledge using the Boolean invariant (BI) allows the number of observed components to be reduced and yields implementations to be evaluated. We consider the optimization levels: No optimization, Basic ($\mathfrak{I}$), and Complete ($\mathfrak{I}$), where $\mathfrak{I}$ is either the boolean invariant BI or the linear invariant LI. For each optimization level, we compare Multiparty-based and Condition-aware implementations. The number of

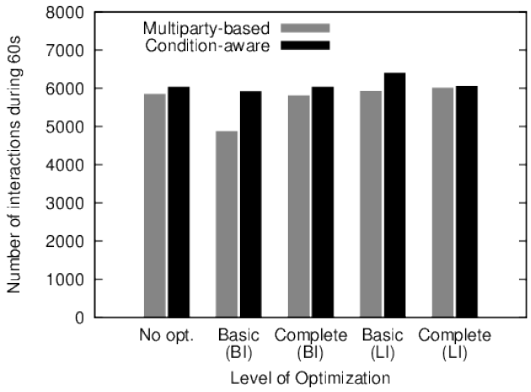

Figure 8.25: Number of interactions executed in 60s for the jukebox example.

interactions executed during 60 seconds is presented in Figure 8.25. Here the performance of Condition-aware implementation is not significantly better than performance of Multiparty-based implementation. The best results are obtained with the basic optimization level using linear invariant. These results come from the fact that no parallelism is allowed between low priority interactions since they are in interaction conflict. More precisely, the only gain in performance consists in time involving actually sending and receiving messages, not in waiting unneeded offers.

Figure 8.26 shows that significantly fewer messages are exchanged with the Condition-aware implementation. Intuitively, this difference corresponds to the notifications and subsequent offers to and from observed components, that are not necessary with the Condition-aware implementation. Interestingly, the implementation giving the best performance (Basic (LI) optimization with Condition-aware implementation) is also the one requiring the least number of messages.

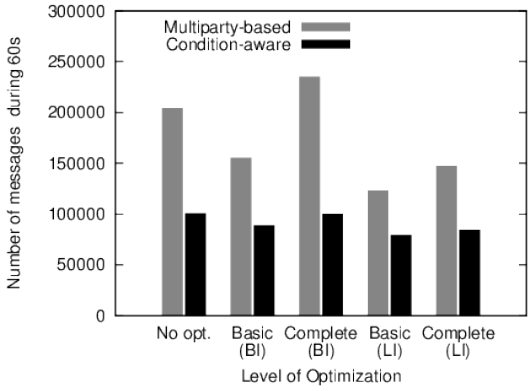

Figure 8.26: Number of messages exchanged in 60s for the jukebox example.

8.4 Optimizing conflict resolution

We present experimental results relative to the computation of the support automata for participants, as presented in Section 3.2. We then present the performance gained when embedding these automata inside the α-core participants, as presented in Section 6.5.

8.4.1 Examples

Our first example is the modified dining philosophers depicted in Figure 8.19, but without priority rules.

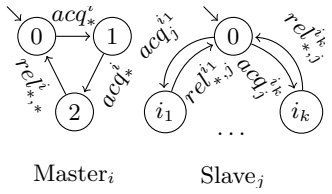

Figure 8.27: Master Slave example

The second example, called Master/Slave, is presented as a system of processes as introduced in Chapter 2. We assume a set of N masters and M slaves. Each master wants to perform a task for which it needs two slaves that it can chose amongst a pool of size K. We denote msNMK such an instance. If the slave j is in the pool of the master i, then the interaction acq^i_j allows master i to acquire slave j, which brings the slave in state i so that it remembers that i acquired it. On completion on the task, the master i releases simultaneously the two acquired slaves j_1 and j_2 through the $rel^i_{j_1,j_2}$ interaction. Figure 8.27 shows respectively, the behavior of a master and a slave. Since the actual number and labels of transitions between two states depends on the parameters of the example, we represented only one transition with a label representing all possible transitions between these two states. Note that the automata of the slave is such that only the master who acquired it can release it.

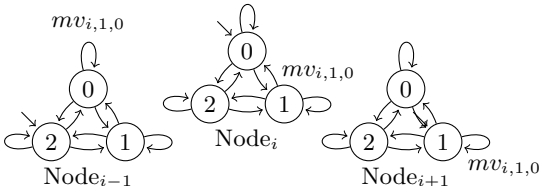

Figure 8.28: Three consecutive nodes of the transmission protocol.

The third example models a transmission protocol that propagates values amongst a chain of memories. At every time, each memory node stores a single value. A fragment of this example is shown in Figure 8.28, expressed as a system of processes. The rule is to propagate (copy) the new value (from the left) only if the memory on the right has already copied the local value. Propagation steps are implemented as ternary interactions

denoted by mv_{i,v_1,v_2}, which correspond to the case where memory i changes its value from v_1 to v_2. As an example, the interaction $mv_{i,1,0}$ in the Figure 8.28 changes the value in Node_i from 1 to 0 if Node_{i+1} already changed its value to 1 and the next value (in Node_{i-1}) is 0. For our experiment, the memories form a ring, thus the sequence of values seen by each memory depends only on the initial state of the system. Note that propagation is enabled at places where the ring contains two consecutive nodes holding the same value. We denote by tp$N$ (resp. tpN') an example with N nodes and one (resp. two) enabled propagations.

8.4.2 Building Support Automata for Participants

We compute the support automaton for each participant by using the corresponding tool. In Table 8.4.2, we present the results of this analysis by giving the average number of states in the original automata and in the support automata. The number of states gives an indication on the size needed to store the knowledge, and the memory needed to execute of the support automata.

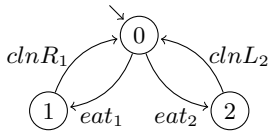

Figure 8.29: Support automaton for participant Fork_2.

For the philo$N$ instances, the support automaton of philosophers is the same as the original automaton. For the forks, there is only one additional state, as shown in Figure 8.29. The added state distinguishes who acquired the fork (left or right). Consequently, only one offer is sent, avoiding unneeded conflict resolution.

In the Master/Slave example, the automaton describing a master is very generic. The corresponding support automaton contains all the possible sequences for acquiring two slaves and then releasing them. In particular, after having acquired two slaves, there is only one possible release interaction, thus only one offer is sent.

Finally, in the transmission protocol example, the size of each support automaton is much larger since it depends on the number of nodes in the chain, that is on the sequence of values seen by each node. If two propagations are possible, then the size of the support automaton is slightly increased, since the two propagations may conflict.

8.4.3 Performance of Distributed Implementation

We obtain a distributed BIP model by embedding the α-core participants and coordinators. From this model, we generate a set of C++ programs communicating through POSIX sockets. We ran the obtained code for both standard α-core and knowledge-optimized α-core on a UltraSparcT1 allowing parallel execution of 24 processes. In Table 8.4.2, we provide the number of interactions executed during 60 seconds of execution (not including initialization) for both standard and optimized version of each

		Average number of states		Number of interactions during 60s	
Name	Components	in B^i	in $\mathcal{K}_i$	Standard	Optimized
philo3	6	2.5	3	1129	2251
philo4	8	2.5	3	1811	2499
philo5	10	2.5	3	2261	4448
philo6	12	2.5	3	2624	4542
philo7	14	2.5	3	3093	4603
ms232	5	2.6	3	1491	1504
ms233	5	3	4.6	1128	1129
ms342	7	2.7	3.1	642	1885
ms343	7	3.1	4.9	1278	1265
ms344	7	3.6	7	1256	1251
tp3	3	3	6	750	1499
tp6	6	3	15	750	1500
tp6'	6	3	16	1498	1557
tp9	9	3	24	750	1509
tp9'	9	3	28	1497	3725
tp12	12	3	33	749	1513

Table 8.6: Results: average size of original and support automaton and performance of the obtained implementation, for each test instance.

test instance. On the dining philosopher instances, the optimized version is up to twice faster than the standard version. On the Master/Slave instances, except for one, the performance is the same for both versions. On the transmission protocol instances, we have a speedup of at least two, except for the tp6′ example.

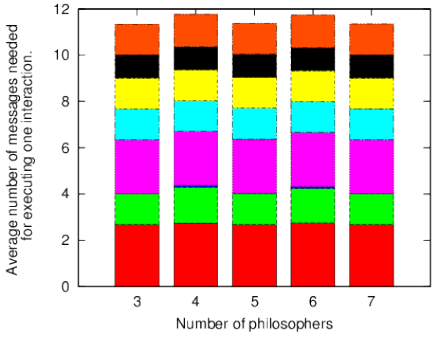

Figure 8.30: Dining philosophers: messages per interaction, standard version.

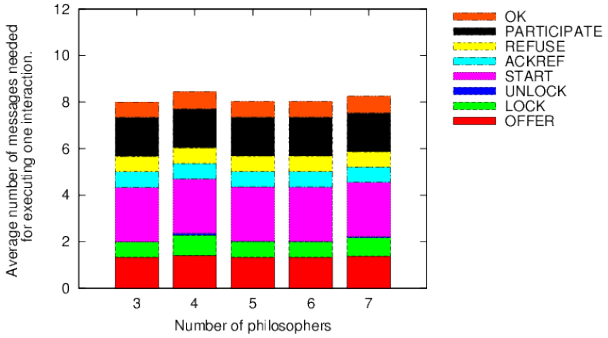

Figure 8.31: Dining philosophers: messages per interaction, optimized version.

In order to evaluate the distributed execution of standard *vs.* optimized versions, we compare the average number of messages needed to perform an interaction for the three examples. For the dining philosophers, these average numbers are shown in Figures 8.30 and 8.31. We can observe a reduction of approximatively 25%, mainly because some **OFFER** messages from the fork participants are transformed in **PARTICIPATE** messages. In turn, this reduces the number of participants to lock, and thus the number

of messages.

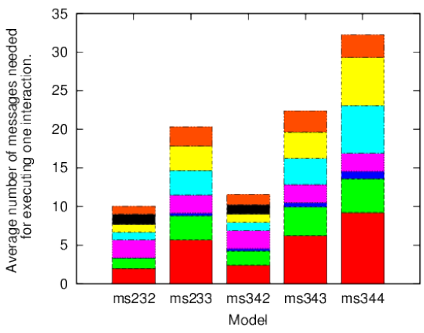

Figure 8.32: Master/Slave: messages per interaction, standard version.

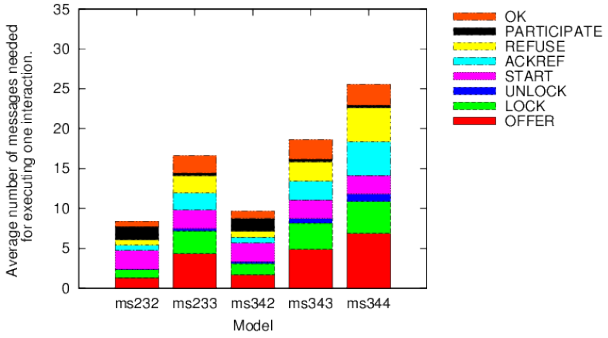

Figure 8.33: Master/Slave: messages per interaction, optimized version.

For the Master/Slave, the average number of messages needed to complete one interaction for standard and optimized α-core are shown in Figures 8.32 and 8.33. Here the number of conflicts depends on the size of the pool of slaves assigned to each master. Since there are many conflicts, the number of offers sent to execute an interaction is quite big. Recall that on this example, performance of both versions is comparable. However, the number of exchanged message is smaller in the optimized version, because less offers are sent.

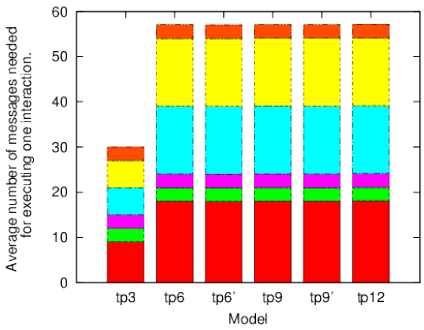

Figure 8.34: Transmission protocol: messages per interaction, standard version.

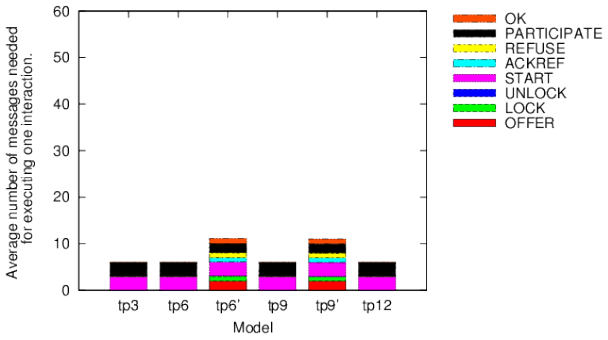

Figure 8.35: Transmission protocol: messages per interaction, optimized version.

For the transmission protocol, the average number of messages exchanged to execute one interaction for standard and optimized executions is shown in Figures 8.34 and 8.35. For the non-primed versions, since there is no dynamic conflict, each participant sends only **PARTICIPATE** messages and each coordinator can directly answer a **START** message. This reduces drastically the number of exchanged messages (6 per interaction,

since they are ternary interactions). For the primed version, in some cases a node may participate in two interactions and thus send two **OFFER** messages, which is still much less than in the original version.

8.5 Discussion

In order to evaluate the diverse transformations and optimizations presented in this thesis, several experiments have been conducted. The Section 8.1 tries to explore the different parameters that are involved in the flow from the centralized model to the distributed implementation. These parameters includes the partition of the interactions and the choice of the conflict resolution protocols. Adding delays to artificially create several situations reveals that there is no best choice. Depending on the conditions in which the model is executed (i.e. cost of computation between cost of communication), the same parameter choice may range from the worst solution to the best one.

In Section 8.2, the examples are more computationally intensive and no delays are added. This section shows that speedup is indeed achieved when distributing this kind of applications. Moreover, the difference between code generated for sockets and code generated for MPI is very interesting. In the first case, receiving a message triggers an interruption whereas in the second case active polling is required. Having more processes than processors does not incur overhead for sockets but severely limits MPI performance. Using the BIP framework, this problem can be solved by either changing the interaction partition to limit the number of processes or by merging some components at the Send/Receive BIP level.

In Section 8.3, we evaluate the use of Condition to implement the priority. There are two aspects to evaluate: the use of a Condition-aware protocol and the possible optimization of the Condition predicates. Note that using a regular multiparty interaction protocol requires to transform each component by adding observation ports, as explained in Subsection 4.2.5. The results of the experiments show that the Condition-aware protocol is much more efficient than the Multiparty-based implementation. Furthermore, optimizing the Condition predicates allows a significant performance gain when using the Multiparty-based implementation. However, this optimization only reduces the number of messages with the Condition-aware implementation.

The last optimization proposed is evaluated with the α-core protocol in Section 8.4. The optimization consists in reducing as much as possible the offers, to avoid unneeded conflict resolution mechanisms. Results show that the number of messages needed to execute an interaction can be significantly reduced. Indeed, sending less interaction offers requires to cancel less offers when an interaction is actually executed. Furthermore, coordinators have more chances to succeed executing an interaction when it is detected enabled, because the corresponding offers were not discarded by knowledge with perfect recall.

9 Conclusion

9.1 Achievements

Rigorous flow for Distributed Implementation

In this thesis, we presented a part of a rigorous design flow, whose goal is to transform a centralized model into a decentralized one. This task is not performed at once but through a sequence of transformations, each of them taking into account a particular aspect of the decentralization. Some of these transformations are enhanced by optimizations. This decomposition into several transformations addresses in the following way the challenges listed in the introduction.

Correctness Each transformation is simple enough to be fully formalized. As both input and output models are expressed using the BIP semantics, one can prove their functional equivalence, either as observational equivalence or as trace equivalence. This guarantees that the final distributed model preserves the functionality of the original high-level model. Checking that the original model meets the desired properties is done through the verification tool of the framework.

Performance/Efficiency First, the solutions provided in this thesis allow for a broad range of decentralization levels. Each atomic component yields an autonomous program in the final implementation. After decentralization, the distributed model contains additional components, named engines, that execute interactions of the original model. The partitioning of interactions into engines is a parameter of the transformation, leaving to the designer the choice that best suits his needs.

Second, several optimizations have been proposed to increase performance of the system or reduce utilization of the network. In particular, introducing the Condition operator allows priorities to be implemented in an efficient manner. The Condition operator can be optimized, by using distributed knowledge, to minimize the number of exchanged messages which may in turn increase the speed of the system. A lower-level optimization, based on knowledge with perfect recall, allows unneeded conflict resolution to be avoided.

Productivity Each transformation and optimization is implemented as a prototype tool. As all transformations are described using the same semantic model, they can be chained to provide a complete flow from the original model to the implementation. Different flows can be defined to fit different platforms. Furthermore, other tools from the toolbox provide additional functionalities. The verification tool permits the validation

the original model, while the flattening tool allows models with hierarchical connectors to be handled. Finally, the merging tool reduces the number of distributed processes in the final implementation by merging several components into a single one in the distributed model.

Knowledge-Based Optimization Techniques

The outline of the thesis first presents the model with a centralized engine, then the decentralization of the engine. In practice, the model with the centralized engine is never built, but has been used for the clarity of the presentation. From a high-level model, our tool constructs directly the decentralized model with distributed engines. Similarly, the two optimizations presented in this thesis derive from artifacts computed on the original model, but which are introduced at the decentralization level for which they are intended.

The first optimization proposed relies on invariants to approximate the global states and compute distributed knowledge associated with a set of components. This optimization only modifies the Condition layer of a high-level model. In the case of complete detection, the input and output models are strictly equivalent, and their centralized execution gives the same results. However, when switching to a distributed context, the optimized model requires less communication. Furthermore, some decisions are taken earlier which increase performance.

The second optimization is done in two steps. The first step consists in computing a support automaton for each component of the original model. Composing the support automata yields the same control behavior as the original model. Therefore there is no gain in a centralized execution. To generate the distributed model, these support automata are embedded within alpha-core participants during a second phase. The optimized version sends offers containing fewer interactions, which reduces the workload of the conflict resolutions protocol.

Implementation and Evaluation on Case Studies

The various transformations and optimizations proposed in this thesis have been implemented. Tools for computing knowledge, based both on invariants and support automata, rely on functions provided by the verification tool DFinder. These prototype implementations allow the performances of the different implementations as well as the influence of the parameters to be compared. Initial results focus on models without priority and mainly study the influence of the partition and the conflict resolution protocol chosen. Other experiments study the efficiency of the optimizations.

Simulations show that depending on the target platform characteristics, the best partition may range from fully centralized to fully decentralized. Similarly, depending on the platform simulated, the best performance is obtained with different conflict resolution protocols. The conclusion is that all these parameters should be available to the designer in order to optimize the distributed model for the target platform.

Running computation intensive code shows that depending on the platform used, the

number of parallel processes has a significant impact on the performances. For instance, POSIX sockets give better performance with more processes than available cores whereas MPI gives better performance with exactly one process per core. If it is not achieved with the partition, using the merging tool can limit the number of processes in the final implementation. For the token ring and dining philosophers protocols, where each conflict resolution component corresponds to an interaction, one can merge each conflict resolution component with the engine managing the corresponding interaction.

Experiments concerning the first optimization actually evaluated two different parameters. The first parameter is the level of detection for the optimization. The second parameter is the expressiveness allowed by the conflict resolution protocol, that is whether it handles Condition or only multiparty interactions. Regardless of the optimization level, using a Condition-aware protocol gives the best performance. Nonetheless, the optimization allows a reduction in the number of messages which sometimes increases performance.

Finally, experiments concerning the second optimization showed that it reduces the number of messages needed to execute an interaction. In turn, the optimization increases the performance as well.

Interestingly, handling Condition efficiently in a distributed protocol seems to be the source of the most important speedup obtained. Thus, it seems fitting to consider the extension of multiparty interactions where their execution is guarded by a predicate depending both on participants and non-participants in the interactions. This extension increases the expressiveness because it encompasses priorities, can be easily implemented, and it gives good performances.

9.2 Future Works

Here, we present some leads for following up this work. The first one is related to the choice of the partition that parameterizes the decentralization process, which is left to the designer. Two other leads are considering extensions of the BIP model, namely real-time and stochastic extension, for which distributed execution is challenging. Finally, considering other primitives than message-passing in the distributed model, such as shared memory is another direction.

Partitioning and Mapping

The partition of the interactions is an important parameter of our transformation. A first issue is whether there exists a partition that is better than the others. Intuitively, the choice of the partition is a tradeoff between the parallelism between interactions and the message complexity needed to solve conflicts.

Conflict-free partitions are of interest because they guarantee the lowest message complexity, namely an offer and a notification per participating component in an interaction. Such partitions are interesting when communications are expensive. There is a single conflict-free partition that maximizes the number of engines.

By contrast to the conflict-free approach, one can consider partitions that preserve maximal parallelism. Such partitions do not allow to group in the same engine interactions that are not directly conflicting. These partitions correspond to clique decomposition of the conflict graph. Hence, there may be several such partitions that minimize the number of engines.

Note that conflict-free partitions might be further separated by using knowledge from support automata. Indeed, these automata may reveal that interactions statically conflicting are actually not conflicting when considering the dynamics of the system. In the case of parallelism-preserving partition, one could group together interactions that are never enabled concurrently according to an invariant of the system.

The tradeoff between lower message complexity and higher parallelism might be guided by considering a specific target platform. Hence, a second question is to find the best partition for a specific platform. This question is related to the problem of mapping an application on a platform, as in [37]. The platform is modeled as a graph whose nodes are processors and edge are connections. An application consists of a set of processes with communications between them. Mapping aims to associate each process to a processor while balancing the load between processors and minimizing communication volume.

Searching for an optimal partition while taking the platform into account corresponds to an extended mapping problem. The extension allows the merging or splitting of some processes of the application by modifying the partition. In order to evaluate the cost of a given mapping, one has to provide the communication and computation loads of the application. These loads can be obtained through a centralized execution of the model. However, a centralized execution does not take parallelism into account. Therefore, the search space for partitions can be limited by some criteria related to parallelism and conflict resolution.

Real-Time Systems

The BIP framework provides a real-time semantics for BIP models [1]. Each interaction is equipped with a timing constraint consisting of a time zone expressed over a set of clocks and an urgency [26]. The time zone indicates when the interaction can be executed, the urgency indicates whether the model authorizes the time to progress before executing the interaction. For instance, an interaction with urgency *eager* has to be executed at the instant it becomes possible; time should not progress while such an interaction is enabled. The timing constraint associated to an interaction is actually obtained from the timing constraints associated to the corresponding transitions in the atomic components.

Models with timing constraints can be executed using a single-thread implementation relying on a real-time engine. More recently, a multi-thread implementation with a centralized engine has been proposed in [83]. As described in Chapter 5, such an engine works with a partial view of the system based on the offers received so far. Before scheduling an interaction, the engine has to ensure that no interaction with an earlier deadline is enabled. This case is similar to executing a low priority interaction where the engine has to ensure that no higher priority interaction is enabled. In the case of priority, waiting for more offers is always safe, though sometimes inefficient, whereas

waiting for more offers in the context of a timed system might lead to a deadline miss. The solution chosen in [83] is to rely on an oracle that is true whenever no interaction with an earlier deadline than the current one might be enabled by a subsequent offer. If the oracle is false, the engine waits for more offers before scheduling any interaction. Note that the centralized engine detects and stops execution whenever a deadline miss occurs.

Decentralizing the engine is even more challenging. Before scheduling an interaction, one has to ensure that no *conflicting* interaction with an earlier deadline is enabled. Even if an interaction is enabled and urgent, it cannot be executed directly. Indeed, executing that interaction will disable all conflicting interactions, which may hide a deadline miss for them. Detecting deadline misses a posteriori requires the history of offers to be remembered and to know which sets of offers correspond to an actual coherent observation of the system.

The problem of implementing multiparty interactions with real-time constraints in a distributed setting is difficult. When implementing multiparty interaction with priority, waiting for more information does not break the semantics. With timing constraints, waiting might sometimes be necessary, and sometimes could be harmful. To distinguish between the two cases, deeper analysis techniques of the system are required.

A more pragmatic approach, where an enabled interaction is executed whenever its deadline is met, is not safe. This approach authorizes some interactions not enabled in the original model. Furthermore, the system may fail to detect a posteriori that the current execution is incorrect.

Distributed Stochastic Execution

Another direction is to consider stochastic models that are equipped with probabilities. For instance, a Markov decision process executes a step by making a non deterministic choice followed by a probabilistic choice. In our case such an execution is obtained by first choosing a component, non deterministically, then choosing probabilistically one interaction in which the component participates. Each component defines a discrete probability for each possible subset of interactions in which it participates. The probabilistic choice is done according to the probability associated to the subset of enabled interactions.

This semantics can be executed in a distributed manner. Each step starts by choosing a master, so that two components involved in a common interaction are not both masters. A master then locks all its slaves, which ensures a stable view of the global state for computing the enabled interactions in which the master participates. The master then chooses and executes an interaction according to the probability associated to the current subset of enabled interactions. Several components may be master at the same time, provided they are not involved in a common enabled interaction.

This approach enables the distributed execution of Markov decision processes, which can be used for simulation purposes. The decision to execute an interaction is based on a partial state comprising all components involved in at least one common interaction with the master. A question is to see what kind of properties, beyond correctness, can

be ensured by such executions. Fairness is a candidate property that could be ensured by such semantics.

Beyond Message-Passing

In this thesis, we consider that a distributed system relies on message-passing for communication. This assumption aims at generality, because most platforms provide message-passing even if they have other means of communications. Manycore platforms often provide shared memories, at different levels. Such platforms might be divided in tiles, that incorporate several cores with a shared memory between them. Communication between these tiles is asynchronous message-passing.

A future direction is to handle shared memory for communication between processes. Similarly to the Send/Receive models, one could define a class of model representing applications communicating through shared memory. Then one could envision hybrid models allowing both message-passing and shared memory communication, for describing applications intended to run on platforms providing both primitives.

List of Figures

Bibliography

[1] T. Abdellatif, J. Combaz, and J. Sifakis. Model-based implementation of real-time applications. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, pages 229–238, New York, NY, USA, 2010. ACM.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983.

[3] K. R. Apt, N. Francez, and S. Katz. Appraising fairness in distributed languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 189–198, New York, NY, USA, 1987. ACM.

[4] F. ARBAB. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14:329–366, 5 2004.

[5] R. Bagrodia. A distributed algorithm to implement n-party rendevouz. In *Foundations of Software Technology and Theoretical Computer Science, Seventh Conference (FSTTCS)*, pages 138–152, 1987.

[6] R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering (TSE)*, 15(9):1053–1065, 1989.

[7] J. Barwise. Scenes and other situations. *Journal of Philosophy*, 78(7):369–397, 1981.

[8] A. Basu, S. Bensalem, D. Peled, and J. Sifakis. Priority scheduling of distributed systems based on model checking. *Formal Methods in System Design*, 39:229–245, 2011.

[9] A. Basu, P. Bidinger, M. Bozga, and J. Sifakis. Distributed semantics and implementation for systems with interaction and priority. In *Formal Techniques for Networked and Distributed Systems (FORTE)*, pages 116–133, 2008.

[10] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Software Engineering and Formal Methods (SEFM)*, pages 3–12, 2006.

[11] A. Basu. *Component-based Modeling of Heterogeneous Real-time Systems in BIP*. PhD thesis, Université Joseph Fourier, 2008.

[12] A. Basu, S. Bensalem, D. Peled, and J. Sifakis. Priority scheduling of distributed systems based on model checking. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 79–93. Springer Berlin Heidelberg, 2009.

[13] A. Basu, L. Mounier, M. Poulhiès, J. Pulou, and J. Sifakis. Using bip for modeling and verification of networked systems – a case study on tinyos-based networks. In *NCA*, pages 257–260, 2007.

[14] K. E. Batcher. Sorting networks and their applications. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[15] S. Bensalem, M. Bozga, A. Legay, T.-H. Nguyen, J. Sifakis, and R. Yan. Incremental component-based construction and verification using invariants. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 257 –256, oct. 2010.

[16] S. Bensalem, M. Bozga, D. Peled, and J. Quilbeuf. Knoweldge-based transactional behavior. In *HVC*, 2012. To appear.

[17] S. Bensalem, M. Bozga, J. Quilbeuf, and J. Sifakis. Optimized distributed implementation of multiparty interactions with observation. In *AGERE*, 2012. To appear.

[18] S. Bensalem, M. Bozga, B. Delahaye, C. Jegourel, A. Legay, and A. Nouri. Statistical model checking QoS properties of systems with SBIP. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 327–341. Springer Berlin Heidelberg, 2012.

[19] S. Bensalem, M. Bozga, S. Graf, D. Peled, and S. Quinton. Methods for knowledge based controlling of distributed systems. In *Proceedings of the 8th international conference on Automated technology for verification and analysis*, ATVA'10, pages 52–66, 2010.

[20] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis. D-finder: A tool for compositional deadlock detection and verification. In *CAV*, pages 614–619, 2009.

[21] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis. Compositional verification for component-based systems and application. *IET Software*, 4(3):181–193, 2010.

[22] S. Bliudze and J. Sifakis. The algebra of connectors - structuring interaction in bip. *IEEE Trans. Computers*, 57(10):1315–1330, 2008.

[23] S. Bliudze and J. Sifakis. A notion of glue expressiveness for component-based systems. In *CONCUR*, pages 508–522, 2008.

[24] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis. Automated conflict-free distributed implementation of component-based models. In *IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 108 – 117, 2010.

[25] M. Bonani, V. Longchamp, S. Magnenat, P. Rétornaz, D. Burnier, G. Roulet, F. Vaussard, H. Bleuler, and F. Mondada. The MarXbot, a Miniature Mobile Robot Opening new Perspectives for the Collective-robotic Research. In *International Conference on Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ*, IEEE International Conference on Intelligent Robots and Systems, pages 4187–4193. IEEE Press, 2010.

[26] S. Bornot and J. Sifakis. An algebraic framework for urgency. *Inf. Comput.*, 163(1):172–202, 2000.

[27] P. Bourgos, A. Basu, M. Bozga, S. Bensalem, J. Sifakis, and K. Huang. Rigorous system level modeling and analysis of mixed hw/sw systems. In *MEMOCODE*, pages 11–20, 2011.

[28] M. Bozga. Component-based design of real-time systems, 2009. Habilitation à Diriger des Recherches, Université Joseph Fourier.

[29] M. Bozga, M. Jaber, N. Maris, and J. Sifakis. Modeling dynamic architectures using dy-bip. In *Proceedings of the 11th international conference on Software Composition*, SC'12, pages 1–16, Berlin, Heidelberg, 2012. Springer-Verlag.

[30] M. Bozga, M. Jaber, and J. Sifakis. Source-to-source architecture transformation for performance optimization in bip. *IEEE Trans. Industrial Informatics*, 6(4):708–718, 2010.

[31] M. D. Bozga, V. Sfyrla, and J. Sifakis. Modeling synchronous systems in bip. In *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT '09, pages 77–86, New York, NY, USA, 2009. ACM.

[32] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, October 1984.

[33] K. M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[34] C.-H. Cheng, S. Bensalem, Y.-F. Chen, R. Yan, B. Jobstmann, H. Ruess, C. Buckl, and A. Knoll. Algorithms for synthesizing priorities in component-based systems. In *ATVA*, pages 150–167, 2011.

[35] M. Y. Chkouri, A. Robert, M. Bozga, and J. Sifakis. Models in software engineering. chapter Translating AADL into BIP - Application to the Verification of Real-Time Systems, pages 5–19. Springer-Verlag, Berlin, Heidelberg, 2009.

[36] F. Chu and X.-L. Xie. Deadlock analysis of petri nets using siphons and mathematical programming. *Robotics and Automation, IEEE Transactions on*, 13(6):793 –804, dec 1997.

[37] S. Cotton, O. Maler, J. Legriel, and S. Saidi. Multi-criteria optimization for mapping programs to multi-processors. In *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on*, pages 9–17, 2011.

[38] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[39] P. Dinges and G. Agha. Scoped synchronization constraints for large scale actor systems. In *Proceedings of the 14th international conference on Coordination Models and Languages*, COORDINATION'12, pages 89–103, Berlin, Heidelberg, 2012. Springer-Verlag.

[40] R. Fagin, J. Y. Halpern, Y. Moses, and V. M. Y. *Reasoning about Knowledge*. MIT Press, 1995.

[41] M. P. I. Forum. *MPI: A Message-Passing Interface Standard, Version 3.0*. High Performance Computing Center Stuttgart (HLRS), 2012.

[42] S. Frølund and G. Agha. A language framework for multi-object coordination. In *In Proceedings of ECOOP*, pages 346–360. Springer Verlag, 1993.

[43] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[44] S. J. Garland and N. Lynch. Foundations of component-based systems. chapter Using I/O automata for developing distributed systems, pages 285–312. Cambridge University Press, New York, NY, USA, 2000.

[45] C. Georgiou, N. Lynch, P. Mavrommatis, and J. Tauber. Automated implementation of complex distributed algorithms specified in the ioa language. *International Journal on Software Tools for Technology Transfer*, 11(2):153–171, 2009.

[46] S. M. German. Programming in a general model of synchronization. In R. Cleaveland, editor, *CONCUR*, volume 630 of *Lecture Notes in Computer Science*, pages 534–549. Springer, 1992.

[47] S. Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. Program. Lang. Syst.*, 26(1):47–56, January 2004.

[48] S. Graf, D. Peled, and S. Quinton. Achieving distributed control through model checking. *Form. Methods Syst. Des.*, 40(2):263–281, April 2012.

[49] I. B. Hafaiedh, S. Graf, and S. Quinton. Building distributed controllers for systems with priorities. *J. Log. Algebr. Program.*, 80(3-5):194–218, 2011.

[50] J. Y. Halpern and R. Fagin. Modelling knowledge and action in distributed systems. *Distributed Computing*, 3(4):159–177, 1989.

[51] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3):549–587, 1990.

[52] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[53] ISO/IEC. *Information Processing Systems – Open Systems Interconnection: LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*, 1989.

[54] M. Jaber. *Centralized and Distributed Implementations of Correct-by-construction Component-based Systems by using Source-to-source Transformations in BIP*. PhD thesis, Université de Grenoble, 2010.

[55] Y.-J. Joung. Two decentralized algorithms for strong interaction fairness for systems with unbounded speed variability. *Theretical Computer Science*, 243:307–338, 2000.

[56] Y.-J. Joung and S. A. Smolka. A comprehensive study of the complexity of multiparty interaction. *J. ACM*, 43(1):75–115, January 1996.

[57] Y. jzer Joung and S. A. Smolka. Coordinating first-order multiparty interactions. *ACM Transactions on Programming Languages and Systems*, pages 209–220, 1994.

[58] G. Katz and D. Peled. Code mutation in verification and automatic code correction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 6015 of *LNCS*, pages 435–450. 2010.

[59] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[60] F. Krückeberg and M. Jaxy. Mathematical methods for calculating invariants in petri nets. In *Advances in Petri Nets 1987*, volume 266 of *LNCS*, pages 104–131. Springer Berlin / Heidelberg, 1987.

[61] D. Kumar. An implementation of n-party synchronization using tokens. In *ICDCS*, pages 320–327, 1990.

[62] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[63] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. 1988.

[64] S. Magnenat, P. Rétornaz, M. Bonani, V. Longchamp, and F. Mondada. Aseba: A modular architecture for event-based control of complex robots. *Mechatronics, IEEE/ASME Transactions on*, 16(2):321–329, 2011.

[65] A. Marron, G. Weiss, and G. Wiener. A decentralized approach for programming interactive applications with javascript and blockly. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, AGERE! '12, pages 59–70, New York, NY, USA, 2012. ACM.

[66] R. Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1995.

[67] R. Milner. *A calculus of communicating systems*. Springer-Verlag, Berlin New York, 1980.

[68] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267 – 310, 1983.

[69] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 –580, apr 1989.

[70] J. Parrow and P. Sjödin. Multiway synchronizaton verified with coupled simulation. In *International Conference on Concurrency Theory (CONCUR)*, pages 518–533, 1992.

[71] Z. Pawlak and A. Skowron. Rudiments of rough sets. *Information Sciences*, 177(1):3 – 27, 2007.

[72] J. A. Pérez, R. Corchuelo, D. Ruiz, and M. Toro. An enablement detection algorithm for open multiparty interactions. In *Proceedings of the 2002 ACM symposium on Applied computing*, SAC '02, pages 378–384, New York, NY, USA, 2002. ACM.

[73] J. A. Pérez, R. Corchuelo, and M. Toro. An order-based algorithm for implementing multiparty synchronization. *Concurrency and Computation: Practice and Experience*, 16(12):1173–1206, 2004.

[74] J. Proença. *Synchronous Coordination of Distributed Components*. PhD thesis, Leiden University, 2011.

[75] J. Proença, D. Clarke, E. de Vink, and F. Arbab. Dreams: a framework for distributed synchronous coordination. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1510–1515, New York, NY, USA, 2012. ACM.

[76] K. Rajan, S. Rajamani, and S. Yaduvanshi. Guesstimate: a programming model for collaborative distributed systems. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 210–220, New York, NY, USA, 2010. ACM.

[77] S. Ricker and K. Rudie. Know means no: Incorporating knowledge into discrete-event control systems. *IEEE Trans. on Automatic Control*, 45(9):1656–1668, 2000.

[78] V. Sfyrla. *Modeling Synchronous Systems in BIP*. PhD thesis, Université de Grenoble, 2011.

[79] V. Sfyrla, G. Tsiligiannis, I. Safaka, M. Bozga, and J. Sifakis. Compositional translation of simulink models into synchronous BIP. In *SIES'10*, pages 217–220, 2010.

[80] J. Sifakis. *Rigorous System Design*, volume 6 of *Foundations and Trends in Electronic Design Automation*. Now Publishers, 2013.

[81] P. Sjödin. *From LOTOS specifications to distributed implementations*. PhD thesis, docs, 1991. Available as report DoCS 91/31.

[82] J. A. Tauber. *Verifiable Compilation of I/O Automata without Global Synchronization*. PhD thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 2005.

[83] A. Triki, J. Combaz, S. Bensalem, and J. Sifakis. Model-based implementation of parallel real-time systems. In V. Cortellessa and D. Varró, editors, *FASE*, volume 7793 of *Lecture Notes in Computer Science*, pages 235–249. Springer, 2013.

[84] R. van der Meyden. Common knowledge and update in finite environments. *Inf. Comput.*, 140(2):115–157, 1998.

[85] J. Zhou, D. Miao, Q. Feng, and L. Sun. Research on complete algorithms for minimal attribute reduction. In P. Wen, Y. Li, L. Polkowski, Y. Yao, S. Tsumoto, and G. Wang, editors, *Rough Sets and Knowledge Technology*, volume 5589 of *Lecture Notes in Computer Science*, pages 152–159. Springer Berlin Heidelberg, 2009.