THÈSE
Pour obtenir le grade de
DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE
Spécialité : Informatique
Arrêté ministériel : 7 août 2006

Présentée par
« Imene BEN HAFAIEDH »

Thèse dirigée par « Susanne GRAF »

préparée au sein du Laboratoire VERIMAG
dans l'École Doctorale MSTII

Systèmes à base de Composants: du Design à l'Implémentation

Thèse soutenue publiquement le « 03/02/2011 »,
devant le jury composé de :

Mr. Jean-Claude FERNANDEZ
Professeur , Université de Grenoble I (Président)

Mr. Elie NAJM
Professeur, Telecom ParisTech (Rapporteur)

Mr. Iulian OBER
Maître de conférence HDR, Université de Toulouse (Rapporteur)

Mr. Gerardo SCHNEIDER
Professeur, Université de Gothenburg (Membre)

Mme. Susanne GRAF
Directeur de recherche au CNRS (Membre)
Abstract

The goal of the thesis is to provide theory, methods and tools for the design and implementation of component-based systems.

To master the complexity of systems of components, we first propose a contract-based design and verification approach which is both compositional and incremental. Then we provide a distributed implementation of these systems allowing to preserve some global properties.

The proposed verification approach uses contracts as a means to constrain, refine and implement systems. It is based on a generic contract framework that we instantiate for a component framework allowing to express progress properties. We also extend the approach to reason about systems of arbitrary size and we show its usefulness for proving safety and progress properties in networked systems.

In the context of distributed settings, these systems must later be executed in a distributed fashion. We also propose in this thesis a protocol that allows executing systems in a distributed way while preserving some global requirements namely priorities and synchronizations and where components interact by message exchange. Then, we provide an implementation of this protocol in a particular platform.

Key words: Component-based design, contract, compositional verification, distributed control, synchronization, priority.
Résumé

Dans cette thèse, nous nous sommes intéressés aux design, vérification et implémentation des systèmes à base de composants.

Nous proposons d’abord une méthodologie de design et de vérification compositionnelle et incrémentale à base de contrats pour les systèmes de composants. Nous proposons ensuite une implémentation distribuée qui permet de préserver certaines propriétés globales de ces systèmes.

La méthodologie de design proposée utilise les contrats comme un moyen de contraindre, raffiner et d’implémenter les systèmes. Elle est basée sur un formalisme de contracts générique, que nous instancions pour un formalisme de composants permettant la description des propriétés de progrès.

Nous étendons cette méthodologie pour raisonner sur des systèmes de taille arbitraire et nous prouvons son utilité pour vérifier des propriétés de sûreté et de progrès d’un réseau de noeuds distribués.

Dans le contexte des systèmes distribués, les systèmes doivent être implémenter de manière distribuée. Nous proposons dans cette thèse un protocole qui permet l’exécution distribuée des systèmes tout en préservant certaines propriétés globales à savoir des synchronisations et des priorités et où les composants interagissent par échange de messages. Nous proposons également une implémentation du protocole pour une plateforme particulière.

Mots clés: Design à base de composants, contrat, vérification compositionnelle, contrôle distribué, synchronisation, priorité.
# Table of contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of contents</td>
<td>3</td>
</tr>
<tr>
<td>List of Figures</td>
<td>7</td>
</tr>
<tr>
<td><strong>Context</strong></td>
<td>9</td>
</tr>
<tr>
<td><strong>Introduction</strong></td>
<td>11</td>
</tr>
<tr>
<td>1.1 Problems and needs</td>
<td>11</td>
</tr>
<tr>
<td>1.2 Design and verification of complex systems</td>
<td>13</td>
</tr>
<tr>
<td>1.2.1 Compositional reasoning</td>
<td>13</td>
</tr>
<tr>
<td>1.2.2 Using contracts</td>
<td>14</td>
</tr>
<tr>
<td>1.2.3 Property verification</td>
<td>15</td>
</tr>
<tr>
<td>1.2.4 Our contribution</td>
<td>16</td>
</tr>
<tr>
<td>1.3 Distributed systems with rich interaction models</td>
<td>18</td>
</tr>
<tr>
<td>1.3.1 Distributed control</td>
<td>18</td>
</tr>
<tr>
<td>1.3.2 Our contribution</td>
<td>19</td>
</tr>
<tr>
<td>1.4 Organization of the thesis</td>
<td>21</td>
</tr>
<tr>
<td><strong>Preliminaries</strong></td>
<td>23</td>
</tr>
<tr>
<td>2.1 The BIP modeling framework</td>
<td>23</td>
</tr>
<tr>
<td>2.1.1 Labeled Transition Systems</td>
<td>23</td>
</tr>
<tr>
<td>2.1.2 Component-based design with BIP</td>
<td>25</td>
</tr>
<tr>
<td>2.1.3 Basic concepts of BIP</td>
<td>27</td>
</tr>
<tr>
<td>2.1.4 Glues in BIP: Interaction models</td>
<td>29</td>
</tr>
<tr>
<td>2.1.5 Composition of components in BIP</td>
<td>32</td>
</tr>
<tr>
<td>2.1.6 Priorities in BIP</td>
<td>34</td>
</tr>
<tr>
<td>2.2 Contract framework concepts</td>
<td>36</td>
</tr>
<tr>
<td>2.2.1 Contract frameworks</td>
<td>38</td>
</tr>
<tr>
<td>2.2.2 Dominance</td>
<td>40</td>
</tr>
</tbody>
</table>
II  A Contract Framework for Reasoning about Safety and Progress  43

3  Contract-Based Verification Approach  45
   3.1 Design and verification methodology  46
      3.1.1 Methodology  46
      3.1.2 Extension to recursively defined systems  50
   3.2 Soundness of the methodology  51
      3.2.1 Soundness  52
      3.2.2 Compatibility of glues  52
   3.3 Proving dominance  53
      3.3.1 Circular reasoning  53

4  A Contract Framework for Components with Data  59
   4.1 Components with data  59
      4.1.1 Semantics  61
   4.2 Glues: Rich interaction model  64
      4.2.1 Connectors  64
      4.2.2 Composition  66
      4.2.3 Composition of interaction models  69
   4.3 Progress description  71
      4.3.1 Progress in components  71
      4.3.2 Progress of a composition  74
   4.4 Relations of the contract framework  76
      4.4.1 Refinement  76
      4.4.2 Conformance  80
   4.5 Proofs  81
      4.5.1 Well-definedness of the contract framework  81

5  An Application: Resource Sharing in a Networked System  87
   5.1 Sharing resource system  87
      5.1.1 The top-level requirement  $\phi$  90
      5.1.2 Methodology  90
      5.1.3 Interaction models and contracts  91
   5.2 Implementation and experimental results  96
      5.2.1 Refinement checker module  96
      5.2.2 Composition module  98
      5.2.3 Results  99

III Building Distributed Controllers for Systems with Priorities  101

6  Controllers for Systems with Priorities  103
   6.1 Systems and controlled systems  104
## TABLE OF CONTENTS

6.1.1 Components, interaction models and systems .......................... 104
6.1.2 Controllers defined by properties ........................................ 107
6.2 Synthesis of priorities for avoiding deadlocks ............................... 111

7 Distributed Controllers for Systems with Priorities .......................... 119
   7.1 Distributing systems and controllers ...................................... 119
   7.1.1 Concurrency and confusion ............................................. 121
   7.2 Implementation of a distributed controller as a protocol ............... 124
   7.2.1 Description of the protocol ............................................. 125
   7.2.2 Avoiding deadlocks due to potential decision cycles ............... 131
   7.3 Correctness of the protocol ................................................. 134

8 Implementation and Experimental Results ...................................... 141
   8.1 Sensitivity of the prototype ................................................. 141
   8.1.1 Sensitivity to prioritized conflicts ................................... 142
   8.1.2 Sensitivity to structural conflicts ..................................... 145
   8.2 The dining philosophers example ......................................... 147

IV Conclusions and Perspectives .................................................... 149

9 Conclusion and Perspectives ...................................................... 151
   9.1 Conclusions ................................................................. 151
   9.2 Perspectives ............................................................... 152

Bibliography ................................................................................. 155
List of Figures

1.1 Approach to the Design of Contract Frameworks ........................................ 17
1.2 Distributed Controllers .................................................................................. 20
2.1 BIP Layers ................................................................................................. 25
2.2 Component Composition ............................................................................. 26
2.3 Hierarchical Components ......................................................................... 26
2.4 Structuring ................................................................................................. 26
2.5 Flattening ................................................................................................. 27
2.6 Atomic Component ..................................................................................... 28
2.7 Example of Connectors ............................................................................... 29
2.8 Structured Connector .................................................................................... 31
2.9 Semantics of a composition ........................................................................... 32
2.10 Composition of Components ....................................................................... 33
2.11 Semantics of Composite Component with Priorities ................................... 35
2.12 Example: Priorities to enforce Mutual Exclusion ........................................ 36
2.13 A hierarchical component and its equivalent flattened form ...................... 38
2.14 $K \sqsubseteq_{A,gl} G$ for conformance defined as simulation ....................... 40
3.1 Step 2: Conformance .................................................................................... 47
3.2 Step 3: $K$ defined as a composition of $\{K_i\}_{i=1}^n$ using $gl_f$ ................ 48
3.3 Step 3: Dominance ($\{C_1, C_2, C_3\}$ dominates $C$ w.r.t. $gl_f$) .............. 49
3.4 Step 4: Satisfaction ..................................................................................... 49
3.5 Methodology steps ensuring that $gl\{A, gl_f \{I_1, I_2, I_3\}\} \ll \varphi$ .......... 50
3.6 $gl_{E_1}$ allows relating the glue $gl_1$ provided in $C_1$ to the actual environment for $K_1$ .................................................. 53
3.7 A counterexample to circular reasoning due to non-determinism ............... 55
3.8 A counterexample to circular reasoning due to strong synchronization .......... 55
4.1 Example: Component with data ................................................................... 62
4.2 Consistent version of a component with data .............................................. 63
4.3 Composition of components ....................................................................... 68
4.4 Composite component .................................................................................. 68
4.5 Merge of Connectors .................................................................................... 69
4.6 $\gamma \odot \gamma' = \{\gamma \bullet \gamma_1, \gamma_2\}$ ............................................................... 70
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.7</td>
<td>$\mathcal{I} \circ \mathcal{I}' = { (\gamma \cdot \gamma_1') \cdot \gamma_2' }$</td>
<td>70</td>
</tr>
<tr>
<td>4.8</td>
<td>Inferring Progress Conditions.</td>
<td>73</td>
</tr>
<tr>
<td>4.9</td>
<td>Refinement: $K^c \sqsubseteq_{K^E, \mathcal{I}} K^a$.</td>
<td>80</td>
</tr>
<tr>
<td>5.1</td>
<td>The overall structure of the application.</td>
<td>88</td>
</tr>
<tr>
<td>5.2</td>
<td>(a) Node Structure.</td>
<td>89</td>
</tr>
<tr>
<td>5.3</td>
<td>(b) Node Behavior.</td>
<td>90</td>
</tr>
<tr>
<td>5.4</td>
<td>Top-level requirement $\varphi$.</td>
<td>92</td>
</tr>
<tr>
<td>5.5</td>
<td>Assumption of the network and the node contract.</td>
<td>92</td>
</tr>
<tr>
<td>5.6</td>
<td>Guarantee of the Node Contract.</td>
<td>93</td>
</tr>
<tr>
<td>5.7</td>
<td>Guarantee of the Network Contract.</td>
<td>95</td>
</tr>
<tr>
<td>5.8</td>
<td>$\mathcal{L}<em>{\text{Net}}$ for contract $\mathcal{C}</em>{\text{Net}}$.</td>
<td>97</td>
</tr>
<tr>
<td>5.9</td>
<td>Inner structure of a network component.</td>
<td>98</td>
</tr>
<tr>
<td>5.10</td>
<td>Refinement Checker Structure.</td>
<td>98</td>
</tr>
<tr>
<td>6.1</td>
<td>An example where reducing non-determinism eliminates a deadlock.</td>
<td>112</td>
</tr>
<tr>
<td>6.2</td>
<td>A simple system not controllable with priorities.</td>
<td>113</td>
</tr>
<tr>
<td>6.3</td>
<td>A solution to the dining philosophers problem.</td>
<td>114</td>
</tr>
<tr>
<td>7.1</td>
<td>Symmetric and asymmetric confusion.</td>
<td>123</td>
</tr>
<tr>
<td>7.2</td>
<td>Prioritied confusion.</td>
<td>123</td>
</tr>
<tr>
<td>7.3</td>
<td>Structure of the protocol for $\mathcal{C}_i$.</td>
<td>126</td>
</tr>
<tr>
<td>7.4</td>
<td>An example with global priorities $c &lt; b$.</td>
<td>131</td>
</tr>
<tr>
<td>7.5</td>
<td>Scenario of possible executions of interactions $a$, $b$ and $c$.</td>
<td>132</td>
</tr>
<tr>
<td>7.6</td>
<td>An example with cycle and independence.</td>
<td>134</td>
</tr>
<tr>
<td>7.7</td>
<td>State diagram of the algorithm.</td>
<td>136</td>
</tr>
<tr>
<td>8.1</td>
<td>Implementation layers.</td>
<td>142</td>
</tr>
<tr>
<td>8.2</td>
<td>System pattern for experiments.</td>
<td>143</td>
</tr>
<tr>
<td>8.3</td>
<td>Sensitivity to the degree of prioritized conflict.</td>
<td>144</td>
</tr>
<tr>
<td>8.4</td>
<td>System pattern for experiments ($T_k$)</td>
<td>145</td>
</tr>
<tr>
<td>8.5</td>
<td>Sensitivity to the degree of structural conflict.</td>
<td>146</td>
</tr>
<tr>
<td>8.6</td>
<td>The dining philosophers problem with priorities.</td>
<td>147</td>
</tr>
</tbody>
</table>
Part I

Context
Chapter 1

Introduction

1.1 Problems and needs

With the growing of the demand about scalability and complexity of systems, it becomes more and more difficult to design correctly their models. In particular, when the major goal is to build a system which ensures a set of desired properties during its execution, such important scale and complexity not only increases potential violations of these properties but also makes them harder to detect and to handle. Some of these violations may cause loss of money, time or even human life. The construction of a system that operates reliably despite of complexity is highly needed but also not always feasible. Therefore the check of the correctness of the system is essential and important to ensure that all requirements and desired properties are respected.

Designing concurrent and distributed systems with such a complex architecture while preserving a set of high-level requirements through all design steps is not a trivial task. An approach which is both compositional and incremental is mandatory to master this complexity. Such approaches generally rely on building complex systems using components. In deed, a central idea in system engineering is that complex systems are built by assembling components defining building blocks. Components are usually characterized by abstractions that ignore implementation details and describe properties relevant to their composition, e.g., transfer functions, interfaces. This allows to split the complex system under study into a set of subsystems which are in general less complex. It is also possible to build larger components by gluing together simpler ones. Gluing or composition can be formalized as an operation that takes in components and their integration constraints, then from these, it provides a description of a new, more complex composite component.

System designers deal with a large variety of components, each having different behaviors and each highlighting different viewpoints of a system. A central problem is the meaningful composition of these components so as to ensure some global properties.

There are two main approaches for detecting property violations of a system: formal testing and formal verification. Consider a model of a system, an environment in which the system interacts, and some properties that the designed system is expected to guarantee, one can choose one of the following approaches depending on their goal.
1.1. PROBLEMS AND NEEDS

**Formal testing** [Bei90, Tre90, Mye04, FFMR07] is a method used to find defects on a system implementation, either during the development or after the complete construction of the system. To do that, testing generates some inputs from environment (test cases) and executes the system to determine whether it produces the required results. It is a quick and direct way to detect bugs or violations in the system. However, testing is not capable of covering all the possibilities that may happen while running the system in reality. The number of possible situations is usually so large that we can test a tiny proportion of them. The absence of property violations provided by testing can not conclude the correctness of the system.

**Formal verification** [UP83, BM79, QS82a] can both search for input patterns which violate the desired properties or prove the correctness of the system if such input patterns do not exist. In contrast to formal testing, formal verification covers all the possibilities that the system can behave. Hence it proves the correctness of the system in the case of the absence of property violation.

It relies on the use of mathematical techniques to prove or disprove the correctness of a design with respect to a certain formal specification. Formal verification has been successfully applied to verify both software and hardware systems. The verification of these systems is done by providing a formal proof on an abstract mathematic model of the system. The mathematic objects that are often used to model these systems are: *labeled transition systems, petri nets, finite state machines, boolean formula*, etc.

The goal of this thesis is to provide theory, methods and tools for the design, verification and implementation of *component-based* systems of arbitrary size with complex architectures preserving a set of high-level requirements through all design steps. Our approach is based on formal verification as it allows to achieve the satisfaction of properties by systems.

As mentioned above, when reasoning about complex systems, decomposing such systems into a set of simpler sub-systems may improve considerably the results of their verification. For this reason, different developments in component-based frameworks have been performed. For example, Ptolemy II [DII+99, EJL+03a] allows simulation of models but not their verification; Software framework component models based on classical concepts of Component-Based Software Engineering (CBSE) like FRACTAL [Fra] with its implementations, e.g., THINK [FSLM02] which also does not provide tools or analysis techniques, whether for simulation or verification; Metropolis [BWH+03] which provides a frontend which produces an internal representation from the meta-model which can be used for simulation or generation model used with the SPIN model-checker [Hol97]. There are also different theoretical frameworks based for example on process algebras e.g., the Pi-Calculus [Mil98] or based on automata e.g., [RC03].

When using a component-framework to model systems for verification checks afterwards, different properties have to be ensure by this framework. Indeed, this requires the framework to be founded on rigorous semantics and provide concepts supporting separation of concerns, e.g., decoupling behavior from interaction. This is particularly absent in the case of modeling, as well as for middleware and software development standards, like CORBA. They use ad hoc mechanisms for building systems from components and offer syntax level concepts only.

Moreover, such frameworks need to encompass heterogeneous descriptions, as most of the platforms and languages, support specific interaction mechanisms and computation models. For instance, software design frameworks are based on interaction by method call and do not allow direct model-
CHAPTER 1. INTRODUCTION

The description of atomic interaction mechanisms. On the contrary, other frameworks such as SystemC and Matlab/Simulink have built in mechanisms for synchronous execution, and are not adequate for describing asynchronous systems. Thus a component framework which provides rich and complex interaction models – where \textit{glues} are used for different purposes such as data transfer, synchronizations, etc, may allow large scale description possibilities. Such frameworks describing such rich \textit{glues} are for example the Kell calculus [BS03] or the \textit{glue} calculus Reo [Arb04]. Kell is, however, mainly concerned with obtaining correctly typed \textit{glues}, and Reo supposes independence amongst \textit{glues} and does not take into account constraints imposed by components. Among such frameworks, we are particularly interested in the BIP framework [BS08a, BS07a, BS07b, BBS06a, BBS06b] developed at Verimag. \textit{Glues} in BIP are interaction models (composition operators) built by a set of \textit{connectors} describing different interaction modes such as \textit{rendez-vous} and \textit{broadcast}. Furthermore, it is possible to define hierarchical connectors, which are connectors defined as a composition of other connectors. Thus BIP also addresses the problem of composition of operators and of their properties, which can be exploited for structural verification [BS07a].

BIP is related to process algebras such as CCS [Mil80] or CSP [Hoa84, Mil83] by its rendez-vous-like interaction mechanism and the restriction to a strictly local notion of state. In this thesis we focus in a first time on the design and verification of these complex systems. Then, in a second time, we study the implementation of such systems. Thus, this thesis is organized mainly in two parts. First, in Part II we propose a design and verification approach based on \textit{contracts}. This approach allows to verify top-level properties of component-based systems of arbitrarily size. Second, in Part III we focus on properties of systems specified by their \textit{glues} namely \textit{synchronizations} and \textit{priorities}. We transform the satisfaction of these properties into a problem of control where properties defined by the \textit{glues} are seen as \textit{memoryless} controllers. Then, we propose a distributed implementation of these controlled systems such that the additional properties induced by their \textit{glues} hold.

1.2 Design and verification of complex systems

As we are interested in formal verification to reason about complex systems, \textit{model checking} [QS82a, CE, CGP99] is one of the most well-known fully automated verification approaches. However, model checking is not always scalable, in particular for systems becoming nowadays more and more complex. Indeed, model checking techniques examine all possible paths through the system’s model to determine whether or not the property being verified is violated. However, to check all possible executions of a model, \textit{model-checking} has to deal with the well-known problem of \textit{state space explosion}. A lot of work has been done to overcome this problem. The major goal is to make the formal verification scalable in order to increase the size of the systems that can be handled. They can in general be categorized into two approaches: optimization/improvement of model-checking’s algorithms [McM93, BCM+90] and compositional reasoning.

1.2.1 Compositional reasoning

\textit{Compositional reasoning} [CLM89, CMP94, Lon93, dRdBH+01a, dRdBH+01b] allows to verify each component of the system in isolation and allows global properties to be inferred about the entire
system. The basic idea is that the system is decomposed into subsystems and these subsystems are analyzed individually. Since subsystems are smaller than the whole system, the individual analysis of the subsystems reduce the effects of the state space explosion problem. The guarantee of global property is then determined by composing the results of these individual analysis. Since through this thesis, we propose a compositional method for the verification of component-based systems, we present below several existing compositional methods.

**Abstraction** [CC77, Lon93, CGL94, DF95, LGS+95a] is a popular technique which verifies properties on a system by first simplifying it. The simplification is often based on the conservative aggregation of states. The simplified system, which is called abstract system, is usually smaller than the original system (concrete system), so the state space is reduced. For a system obtained from the parallel composition of a set of components, i.e \( S = K_1 \parallel \cdots \parallel K_n \), the compositional abstraction first computes, for each component \( K_i \), an abstract component \( K^a_i \), then it composes the abstract components \( S^a = K^a_1 \parallel \cdots \parallel K^a_n \) to obtain an abstract system \( S^a \) of \( S \). The abstraction is required to be exact, i.e. the properties that hold on the abstract system also hold on the concrete system. However, the abstraction is often not complete, i.e. not all true properties of the concrete system are also true on the abstract system so that a process of abstraction refinement may be necessary to guarantee property preservation throughout the abstraction process.

**Assume-Guarantee** [Pnu85a, Jon83a, HQR98, GPB02, CGP03] is a compositional approach that decomposes properties into two parts. One is an assumption about the global behavior of the environment and the other is a property guaranteed by the component when the assumption about its environment holds. The assumption is needed since when a subsystem is verified it may be necessary to assume that the environment behaves in a certain manner. Often the behavior of a subsystem depends on the subsystems with which it interacts, thus we need to provide an assumption about its environment to verify properties of that subsystem. Consider a system \( S \) which is decomposed into two subsystems \( S_1 \) and \( S_2 \). \( P \) is a property to be verified on the parallel composition of \( S_1 \) and \( S_2 \), denoted by \( S_1 \parallel S_2 \). The basic assume-guarantee rule is: if under assumption \( A \), subsystem \( S_1 \) satisfies property \( P \) and \( A \) is satisfied by subsystem \( S_2 \), then the system resulting from the parallel composition \( S_1 \parallel S_2 \) satisfies the property \( P \). Despite of being largely advertised, many issues make the application of assume-guarantee rules difficult. They are discussed in detail in a recent paper [CAC08a]. The paper provides an evaluation of automated assume-guarantee techniques. The main difficulty is finding decompositions into sub-systems in the case of many parallel sub-systems \( S_1 \parallel \cdots \parallel S_n \). The verification performance depends on the way of decomposition but finding a good decomposition is not always feasible. Another problem is choosing adequate assumptions for a particular decomposition. The assumption should be weak enough to be satisfied by a sub-system but also be strong enough to prove the global property.

### 1.2.2 Using contracts

Contract-based design is an expressive paradigm for a modular and compositional specification of systems. The use of contracts has been advocated for a long time in computer science [Hoa69, AL93]
and, more recently, has been successfully applied in object-oriented software engineering [Mey97].

Like in contract-based design [Mey92, NMO09, LMS07], in our approach we use contracts to constrain, reuse and replace implementations.

The basic idea of design-by-contract is to consider the service or the property provided by a component as a contract between this component and its environment. Thus a contract is usually expressed as a pair of an assumption, or a property that the environment must satisfy, and a guarantee, the properties that must be satisfied by each particular component. As in Assume/Guarantee reasoning [GGTG10] and in [GQ07], such a separation between assumptions and guarantees allows more flexibility in finding compatibility relations between components. Moreover, defining multiple contracts, thus multiple views, of a given system allows better isolation between systems and hence better compositionality.

In the theory of interfaces [dAH01a, LNW06a, GLS96], they offer a notion of contracts to check interface compatibility between reactive systems. In that context, it is irrelevant to separate the assumptions from guarantees and only one contract needs to be and is associated with a given system. Separation and multiple contracts become of importance in a more general-purpose software engineering context.

In [MM04b, BM09], a notion of synchronous contracts is proposed. The contracts described are executable specifications (synchronous observers). Such an approach is satisfactory to verify safety properties of individual modules (synchronous) but can hardly be applied to the modeling of globally asynchronous architectures.

In the context of software engineering, this notion of assertion-based contract has been adapted for a wide variety of languages and formalisms but the notion of rich exogenous composition operators and interaction models needed to represent abstractions of protocols, middleware components and orchestrations is not always taken into account. In [QG08a], a first framework generalizing interface theories by adding a structural part to contracts is proposed.

In interface theories [LNW07], then in [QG08a], authors use modal specifications [Lar89, LX90] to enrich their contract frameworks. Modal specification are interesting to deal with loose specifications and properties implying some progress insurance in absence of input enabledness: in a modal transition system, a must-transition represents a progress guarantee of all its implementations while may-transitions define safety properties as in usual transition systems.

### 1.2.3 Property verification

In this thesis, we use contracts to reason about safety and progress properties. We are interested in the description of progress properties. During the last decade an important progress in the ability of tools to verify properties of hardware and software systems has taken place [CGJ+03, Hol97]. This success has in a great part concerned safety properties such as absence of run-time errors, deadlocks etc. However, the progress in verification of progress properties has been less prominent as they are harder to verify than safety properties. Indeed, in [MCM08], authors prove that deciding liveness of a set of components is NP-hard.

Progress properties are also called liveness properties. A component is considered to be live if it will repeatedly participate in some step of the system, independently of how the global system evolves.
1.2. DESIGN AND VERIFICATION OF COMPLEX SYSTEMS

Among the methods dealing with this kind of properties, we mention invisible invariant [FMPZ06, FPPZ04], counter abstraction [AFK88, FK84, FMPZ06, FPPZ04] which are based on a set of fairness requirements (weak or strong) that enable proofs of liveness properties of a parameterized systems [PPR]. Invariant generation [CS02] which handles termination of sequential programs and Backwards Reachability [AIRS06] presents complement to other methods for proving termination, in that it transforms a termination problem into a simpler one with a larger set of terminated states.

1.2.4 Our contribution

A first contribution of this thesis is providing a scalable methodology for design and verification of component-based complex systems. We focus on systems of arbitrary size and we preserve a set of high-level properties (requirements) through all design steps. Like in contract-based design [Mey92], we use contracts to constrain, reuse and replace implementations. As described above, one of the shortcomings of such reasoning is that it does not take into account the interaction model between components and how they are composed.

Thus our notion of contract has a structural part, which makes this definition very general by encompassing any composition model, in particular rendez-vous like composition. A more practical advantage is related to system design: it allows separating the architecture and the properties (requirements) of the system under construction, which evolve separately during the development process. In particular, in frameworks where interaction is rich, refinement can be ensured by relying heavily on the structure of the system and less importantly on the behavioral properties of the environment.

This structural part of our contracts encode rich exogenous composition operators which allow to represent abstractions of protocols, middleware components and orchestrations whereas assumptions and guarantees should constrain peers at the same or at an upper layer.

Interfaces [dAH01b] as described previously cannot encode such rich composition operators as they are based on a fixed rather than a generic model of composition — usually synchronous Input/Output (I/O) composition. In this work we are more expressive than the previously described approaches.

Other formalisms for describing such rich composition operators abstractly have been proposed, e.g., the Kell calculus [BS03] or the connector calculus Reo [Arb04]. Kell is, however, mainly concerned with obtaining correctly typed connectors, and Reo supposes independence amongst connectors and does not take into account constraints imposed by components. Thus we choose in this thesis to use a subset of the rich composition operators of the BIP component framework [BBS06a] as they provide the required expressiveness, define interaction with component behaviors and handle conflicting connectors. Using a formalism like BIP for wide area systems allows using layered specifications where different layers can be analyzed separately by abstracting complex lower layer protocols by an atomic multi-party interaction. Using connectors as abstractions of lower level protocol stacks leads to clearly structured models.

In our contracts, we also keep assumptions and guarantees separate and moreover we describe them on different alphabets which allows improving reusability.

To apply our verification methodology, we formalize and extend the framework introduced
in [QG08a] to distributed component systems of arbitrary size and we show its usefulness for proving safety and progress properties in networked systems.

Figure 1.1: Approach to the Design of Contract Frameworks.

In this thesis, in addition to a compositional verification methodology using a contract framework, we provide a design approach that uses the results of this methodology to verify new properties of different applications with no further proofs. More precisely, this generic design approach consists in two phases:

1. define a general notion of contract framework stating the necessary ingredients to apply our verification methodology. Then rules for establishing validity conditions for these relations are provided.

2. for any particular application, one only has to define instantiations of these generic notions and check the validity conditions. Once the concrete framework has been defined, the rules and the verification methodology can be applied without any further proofs (see Figure 1.1).

The generic contract framework, we use, is based on some notion of refinement, which allows to define the following relations: conformance which is refinement with respect to a given specification, dominance which is refinement between contracts and satisfaction which is refinement of an implementation with respect to a contract. To prove the validity conditions for these relations we use circular reasoning which allows to derive an interesting rule to prove dominance, with no need to compose contracts.

We apply the proposed design and verification methodology to an application, where we have focused on progress properties. To reason about progress, we propose a formalism similar to symbolic transition systems as introduced in [MP91], which we extend in several ways. We define progress constraints close to the usual strong and weak fairness [AFK88, FK84, FMPZ06, FPPZ04] and we decorate control states with invariants on state variables.
1.3. DISTRIBUTED SYSTEMS WITH RICH INTERACTION MODELS

We also consider an explicit interaction model \((\text{glue})\) represented by sets of \(\text{connectors}\). Each connector defines a set of interactions and a transformation on (non persistent) port variables, where \(\text{ports}\) name transitions of the local components involved in the interaction.

For achieving scalability, we base verification on an abstract semantics in which explicit values of state variables are abstracted by the defined state invariants. Given the complexity of the specifications, not having to prove the correctness of the proof rules in this concrete setting is very helpful.

We apply the methodology to a resource sharing algorithm in a networked system of arbitrary size. The different verification steps and proofs are automated in a tool developed for this purpose.

Although this is not presented in this thesis, we have also addressed the problem model-based design and validation \([\text{PBH}^+09, \text{IBHR}09]\). We have applied in \([\text{IBHR}09]\) a design approach using the OMEGA-RT profile \([\text{OGO}03, \text{GOO}05]\) for a design phase and the IF toolset \([\text{OGY}06, \text{BGO}^+04a]\) for a validation phase.

1.3 Distributed systems with rich interaction models

In Part III, we are interested in giving a distributed implementation of component-based systems. These systems represent rich interaction models defining a set of properties. We propose a distributed implementation with respect to these properties, which are seen as controllers. The satisfaction of these properties defined by the interaction model, can be generalized to the problem of controlling an existing system in order to force it to satisfy some additional safety constraints \([\text{RW}92b]\).

Component-based systems are likely to be implemented in a distributed fashion, as each component can be hosted by a different site. When the constraints added by the interaction model are global, distributing the controlled system is not a trivial task. Indeed, this is proven in \([\text{PR}90]\) to be undecidable for concurrent systems. We focus on two types of properties namely synchronization and priority. Typical frameworks used to express synchronization-based specifications are (prioritized) Petri nets, process algebras \([\text{Mil}80, \text{Hoa}84, \text{BB}87]\) or their UML incarnation, namely activity diagrams. BIP \([\text{GS}05, \text{BBS}06a]\) generalizes the basic concepts of these formalisms (see \([\text{BS}08b]\)). Interaction by synchronization is an expressive modeling paradigm as it encompasses all commonly used communication and interaction primitives. Specifying priorities amongst a set of alternative synchronizations is interesting in many contexts. For example, it is likely that amongst a set of enabled synchronizations amongst subsets of components, one will prefer those involving larger subsets. Another typical example of the use of priorities are components which for different activities require one or more resources amongst a shared pool of resources. There exist several abstract frameworks allowing to represent priorities such as process algebras with priorities or prioritized Petri nets \([\text{BBBS}08, \text{BBPS}, \text{GS}04]\).

1.3.1 Distributed control

At the implementation level, a distributed system is defined by a set of components that interact using the communication mechanisms provided by a distributed platform. Typically, such a platform allows communication via message passing and the level of abstraction that is provided determines the properties that can be guaranteed by the message passing mechanism. The platforms defined by
component frameworks such as Corba, JavaBeans or .NET aim at making transparent their distributed nature to a designer using standard (sequential) programming languages, where interaction is through method call. In this context, one generally distinguishes between active components — threads or processes — driving the computation and passive ones which get activated temporarily when needed. A requirement (i.e., a property to be ensured) is in general a property of complete executions of some thread. In service-oriented systems, different processes (initiated for example by different clients) are often considered to be functionally independent: they may share resources (data and platform capacities), however the problems resulting from this are often ignored at early design stages and the primitives provided for specifying systems consisting of several interacting threads (or ongoing activities) are often low-level mechanisms such as semaphores or time-based scheduling.

Different algorithms have been proposed to describe message passing distributed protocols. For instance, in [Bag89b, Hoa78] a first algorithm ensuring binary synchronization between components has been described. Then algorithms handling multiparty synchronizations have been proposed in [PCT04, Bag89a, FF96]. Distributed implementations preserving some global properties are not trivial to achieve as, we cannot determine the exact global state of a distributed system, we can only approximate it (see [CL85, Tri04, Thi05]). Some approaches propose the use of statically computed knowledge about the possible global states to decide about the satisfaction of some global constraints in a distributed system [BBPS, GPQ10, RR00a, RW92a].

1.3.2 Our contribution

We study the design and implementation of systems in which correct interaction between components is essential in order to achieve functional properties defining the services to be provided by the system and/or non-functional properties specifying some constraints on their quality. For this purpose, we view a system as a set of active components which interact by synchronizing certain activities. A synchronization between a set of components is the abstraction of a sequence of messages between these components that results in the atomic execution of some local transition in each of these components. The motivation for specifying interaction using synchronizations is that these may represent some global activity that is in fact distributed over several components. Thus, it is possible to abstract away the detailed specification of a particular order in which the local activities have to be executed or how atomicity is achieved on some actual platform — which may offer message-based communication, but which might as well be a CAN-bus or a multi-core processor where communication is by shared memory. We introduce an abstract representation of components and systems where components are identified with an abstraction of their behavior and represented by transition systems labeled by actions (which are also called ports). In a first phase, we totally abstract from how such a synchronization is realized; then, we provide a distributed implementation for it as a message-passing protocol. We define synchronizations between components and priorities defined on the set of their interactions as memoryless controllers.

We propose also to use controllers defined by priorities to avoid deadlocks in a given specification. More precisely, we propose that instead of systematically asking the user to rework a given specification when a reachable deadlock is detected, we propose to restrict the possible executions to those avoiding deadlock by means of a priority order. Why do we choose priorities as a means for
1.3. DISTRIBUTED SYSTEMS WITH RICH INTERACTION MODELS

avoiding deadlocks? One reason is that we suppose that specifications explicitly specify a (close to) maximal degree of concurrency and may therefore have a high degree of non-determinism. Adding buffers and reordering messages as proposed in [SB09] has the inconvenient of being tied to a lower level of abstraction — at which state explosion is a big issue for verification — and moreover, it is not adequate when interaction is by synchronization and each component already exhibits its maximal potential of concurrency. But restricting non determinism may be very useful, and priorities are an interesting means for doing so.

Another reason is that, when it is guaranteed that prioritized executions are deadlock free, a set of priority rules is a memoryless controller, that is, deciding which next transition is possible requires neither history nor look-ahead. In fact, the construction of the priority rules does eliminate statically the look-ahead required to avoid deadlocks without additional memory, thus keeping the specification small at that level of abstraction.

Our problem is that of synthesizing a distributed memoryless controller, where we define what it means to distribute a controller, and what a correct implementation should be. Furthermore, this general presentation shows that our approach applies not only to interaction models and priorities, but to all memoryless properties. We distribute these memoryless controllers by proposing a protocol that transforms a (global) system specification into a distributed implementation consisting of a set of components communicating through message passing where we suppose that the underlying communication platform ensures reliable and order-preserving transmission of messages. As already described, there exist several protocols achieving such implementations. However, specifications requiring in addition global priorities to be respected have rarely been considered. [BBPS, GPQ10] address the problem of distributing prioritized Petri nets, but for an underlying platform on which synchronization is provided as a primitive, whereas we try here to improve the efficiency of the resulting implementation by means of a combined algorithm. In [BBPS, GPQ10] they consider the same problem but with a different progress property, namely deadlock freedom, whereas our protocol ensures what we call maximal progress, which is also the progress criterion adopted in [RW92a, RR00a, PCT04]. Given a system $S$ of components, we propose to build, from a global memoryless controller defined by an interaction model $(I)$ and a priority order $(<)$, a set of local controllers (denoted $C_i$) associated to each component so as to ensure the constraints defined by $I_{<}$ in a completely distributed way (see Figure 1.2).

![Figure 1.2: Distributed Controllers.](image-url)
We give an implementation of this protocol using the Message Passing Interfaces (MPIs) \cite{GLS99, SOHL96}, and we give an evaluation of different performance metrics. We also compare our algorithm to the existing $\alpha$-core algorithm presented in \cite{PCT04}.

### 1.4 Organization of the thesis

The thesis is split into four parts, Part I presents the context and the needed concepts (Chapter 2). Part II describes the proposed design and verification methodology \cite{IBHQ10c, IBHQ10b}, the framework used to apply it (Chapter 3 and 4) and its application to a case study using a tool developed for this purpose (Chapter 5). Part III proposes a distributed implementation of systems with priorities \cite{IBHQ10a, QBHG09, IBH10} (Chapter 6, 7 and 8). The last Part IV draws the conclusions and future work (Chapter 9). The details of all chapters are as follows:

- Chapter 2 presents preliminaries and concepts used in the sequel of this thesis. It describes the basic notions about component-based frameworks, in particular the version we use of the BIP component framework, its interaction models based on connectors and how it allows to express priorities. This chapter also provides the different basic notions of the contract-framework that we need in our approach.

- Chapter 3 presents a formal description of the different design steps of our verification approach based on a generic notion of contract framework. In particular, it discusses relations and properties that contract frameworks have to provide so that our approach could be applied. We also detail how this methodology can be applied in the case of systems of arbitrary size once these systems can be built using a grammar of components.

- In Chapter 4 we formalize the different ingredients of the contract framework used to apply the methodology described in Chapter 3. We focus on progress properties, thus we provide rich specification that allows expressing such kind of properties. Progress properties are described by progress conditions which adapt the usual weak and strong fairness conditions to component systems. The proposed contract-framework handles variables and data exchange between components. We give proofs that such contract-framework allows indeed to apply the proposed methodology.

- In Chapter 5 we apply our methodology to a networked system for sharing resources. We verify a top-level progress requirement of this case-study and we detail the different verification phases applied. We also provide a tool that implements the main verification checks of our methodology and we use this tool to validate the case-study results.

- Chapter 6 focuses on the properties defined by the interaction models of component-based systems, where these properties are seen as memoryless controllers. It also discusses the synthesis of such controllers.

- In Chapter 7 we present the protocol that transforms a system (with binary synchronizations) and its memoryless controller into a distributed implementation based on message-passing. We
give proofs of its correctness with respect to the usual properties of distributed algorithms and we describe through an example how we deal with deadlocks due to circular configurations.

- Chapter 9 concludes the thesis and hints at worthwhile future developments.
Chapter 2
Preliminaries

In this chapter, we first recall some basic definitions about labeled transition systems and their usual refinement relations, which are used to describe behaviors of components throughout this thesis. Then, we present the BIP framework [GS05, BBS06a, BS07a], which is one of the motivations for our work. We briefly discuss BIP in its generality. Then we present a variant of this framework that we use in this thesis. Finally, we review the basic notions of the generic contract framework proposed in [QG08a], from which our work on contract frameworks is inspired. In particular, we emphasize the properties of such a framework allowing to reason about contracts in a component framework with rich interaction models such as BIP.

2.1 The BIP modeling framework
2.1.1 Labeled Transition Systems

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

Definition 2.1.1 (Labeled Transition System (LTS)) A labeled transition system is a tuple \( TS = (Q, q^0, P, \rightarrow) \), where: \( Q \) is a set of states, \( q^0 \in Q \) is the initial state, \( P \) is a set of labels (actions). \( \rightarrow \subseteq Q \times P \times Q \) is a set of transitions each labeled by an action.

As usual, for any pair of states \( q, q' \in Q \) and an action \( p \in P \), we write \( q \xrightarrow{p} q' \), iff \( t = (q, p, q') \in \rightarrow \). The state \( q \) is called the start state of \( t \) and \( q' \) its target state. If such \( q' \) does not exist, we write \( q \xrightarrow{p^c} \). These labeled transition systems could be enriched with variables for example, and thus called extended labeled transition systems which we denote ELTS.

Definition 2.1.2 (Extended Labeled Transition System (ELTS)) An extended labeled transition system is a tuple \( (TS, X, g, f) \) where:
2.1. THE BIP MODELING FRAMEWORK

- \( TS = (Q, q^0, P, \rightarrow) \) is a labeled transition system: \( Q \) is a set of control states, \( q^0 \in Q \) is the initial state, \( P \) is a set of labels. \( \rightarrow \subseteq Q \times P \times Q \) is a set of transitions each labeled by an element of \( P \). Elements of \( P \) are ports;

- \( X \) is a set of variables. Some variables are associated with a (unique) port;

- \( g \) associates with every transition \( t \) a guard \( g_t \), i.e. a predicate on \( X \);

- \( f \) associates with every transition \( t \) a function \( f_t \) defined on \( X \) and corresponding to local state transformations.

If \( t = (q, p, q') \in \rightarrow \), \( g_t \) is a pre-condition for interaction through \( p \), and \( f_t \) is a computation step consisting of local state transformations. \( g_t \) is also known as the guard of the transition and the transition can be executed if the guard is true.

**Definition 2.1.3 (Execution, Extended prefix, Reachable states, Deadlocks)** Let be \( TS = (Q, q^0, P, \rightarrow) \) an LTS, Then:

- an execution \( \sigma \) is a (possibly infinite) maximal (i.e. cannot be extended) sequence \( q_0 \cdot p_0 \cdot q_1 \cdot p_1 \cdot \ldots \) starting in the initial state \( q_0 \) and such that for any \( i \geq 0 \) such that \( q_i \) and \( q_{i+1} \in \sigma \), it holds that \( q_i \xrightarrow{p_i} q_{i+1} \). Thus \( \sigma \) could be also written a sequence of states and transitions, that is \( \sigma = q_0 \cdot t_0 \cdot q_1 \cdot t_1 \cdot \ldots \), where \( t_i = q_i \xrightarrow{p_i} q_{i+1} \).

  We suppose without loss of generality, that executions are infinite, that is always extended by some \( \epsilon \) transitions. We denote by \( \text{exec}(TS) \) the set of infinite executions of \( TS \).

- a prefix on \( P \) and \( Q \) which we denote \( \beta \) is a finite sequence \( q_0 \cdot p_0 \cdot q_1 \cdot \ldots \cdot q_\beta \) with \( q_i \in Q \) and \( p_i \in P \);

- an extended prefix on \( P \) and \( Q \) is a pair \( (\beta, \Lambda) \) where: \( \beta \) is a prefix on \( P \) and \( Q \) and \( \Lambda \) is a set of labels in \( P \) which we call an acceptance set.

- A state \( q \in Q \) is reachable if there exists an execution \( \sigma \) containing \( q \)

- \( q \subseteq Q \) is a deadlock if \( \nexists (q', p) \in Q \times P \), \( q \xrightarrow{p} q' \)

\( TS \) is called deadlock free if it has no reachable deadlock state.

We now introduce simulation [MH89] which is a preorder on LTS and allowing to compare them.

**Definition 2.1.4 (Simulation)** Let \( S_1 \) and \( S_2 \) be two LTS. A relation \( \mathcal{R} \subseteq Q_1 \times Q_2 \) is a simulation relation of \( S_2 \) by \( S_1 \) iff \( q_1^0 \mathcal{R} q_2^0 \) and for any pair \( (q_1, q_2) \in Q_1 \times Q_2 \) and any \( q_1' \in Q_1 \):

\[
q_1^1 \mathcal{R} q_2^1 \text{ and } q_1 \xrightarrow{\alpha} q_1' \text{ implies that there exists } q_2' \in Q_2 \text{ such that } q_2 \xrightarrow{\alpha} q_2' \text{ and } q_1' \mathcal{R} q_2'
\]

\( S_1 \) simulates \( S_2 \) if and only if there exists such a relation.
CHAPTER 2. PRELIMINARIES

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

We now define an equivalence relation between LTS called bisimulation.

**Definition 2.1.5 (Bisimulation)** Let $K_1$ and $K_2$ be two LTS. A relation $R \subseteq Q_1 \times Q_2$ is a bisimulation if it is a simulation and furthermore $R^{-1}$ is a simulation of $K_1$ by $K_2$.

### 2.1.2 Component-based design with BIP

BIP [BS08a, BBS06a, BJS09a, BBS06b] is a component framework for designing component-based systems with complex interactions. Its main principle is that there should be a clear separation between the behavioral and the architectural parts of systems. Indeed, such a separation allows efficient structural verification techniques.

BIP framework supports a component construction methodology based on the thesis that components are obtained as the superposition of three layers (see Figure 2.1). The lower layer contains atomic components described by their *behavior*. The intermediate layer includes *glues* which are represented by *interaction models* built as a set of *connectors* linking ports of different components. The upper layer is a set of *priority* rules describing scheduling policies and preferences between interactions.

![Figure 2.1: BIP Layers.](image)

In BIP, hierarchical components are defined inductively from atomic ones:

- **atomic** components defined as basic element that only represents *behavior* which means an LTS or an extended LTS.

- **composite** or **compound** defined as a composition of a set of components using glues. A composite component could be flat or hierarchical. A composite component is flat if it is a composition of only atomic components and it is hierarchical if it is not flat.

A component is denoted graphically by a box with a well defined interface allowing to interact with its environment. Interfaces are defined as a set of *ports*. The box of an atomic component contains behavior inside and the box of a composite component contains other components and glues.

Given a set of atomic components $\{K_1, K_2, \ldots K_n\}$ and a glue $GL$. The composition of $\{K_1, K_2, \ldots K_n\}$ using $GL$ produces a composite component $K$, as shown in Figure 2.2 where

$$K = GL(K_1, K_2, \ldots K_n)$$
2.1. THE BIP MODELING FRAMEWORK

Hierarchical components are obtained as composition of such composite components. This is allowed by the fact that glues in BIP could be also composed. For example, in Figure 2.3 components $K_1$ and $K'_1$ are composed with glue $GL_1$, and the resulting composite component is integrated with $K_2$ by $GL_{12}$ to produce a hierarchical component. In BIP, glues and their composition operators provide a set of properties, namely incrementality, compositional reasoning and composability.

**Incrementality.** means that composite systems can be considered as the composition of smaller parts. Incrementality provides flexibility in building systems by simply adding or removing components and the result of construction is independent of the order of integration. It is necessary for progressive analysis and the application of compositionality rules. Incrementality allows coping with the complexity of the heterogeneous and large-scale systems in both construction and verification phases.

Incrementality means the two following properties:

- **Flattening**: means that any given structure can be flattened to a component which is the composition of its atomic components by using a single glue operator.

$$GL_1 (K_1, GL_2 (K_2, \ldots)) = GL (K_1, \ldots, K_n)$$
CHAPTER 2. PRELIMINARIES

An example is shown in Figure 2.5. Flattening allows to avoid reasoning on hierarchical components by only reasoning on their flattened version. Note that flattening is a weak property that is in general satisfied by component frameworks.

- Structuring: means that an n-ary glue operator could be obtained by successive application of a binary glue operator, as shown in Figure 2.4. In general, we should be able to write

\[ GL(K_1, \ldots, K_n) = GL_1(K_1, GL_2(K_2, \ldots, K_n)) \]

That is, any composite component can be obtained by successive composition of its atomic components. This property is useful when one wants to build the glue that relates some particular component to the rest of the composite component (the system). Notice that structuring is a very strong property. For the definition of contract frameworks which we will propose, we will require a weaker condition, inspired by this one which is easier to satisfy and sufficient for contract-based reasoning.

By the above two mechanisms, a given system of behavior can be partitioned into any required structure.

**Compositional reasoning.** Compositional reasoning rules allow inferring global system properties from the local properties of the sub-systems. (e.g inferring global deadlock-freedom from the deadlock-freedom of the individual components). Compositional reasoning is necessary for obtaining correctness-by-construction. This is interesting when applying incremental verification approaches.

**Composability.** Composability rules guarantee that, under some conditions, essential properties of a component will be preserved after integration. Composability means stability of previously established component properties across integration, e.g. a deadlock-free component will remain deadlock-free after gluing together with other components. Composability is essential for incremental construction as it allows building large systems without disturbing the behavior of their components.

### 2.1.3 Basic concepts of BIP

In this section as well as in the sequel of this thesis, components are denoted \( K_1, K_2, \) etc and given a set of ports \( \text{Ports} \), the interface of an atomic component \( K \) is defined by a set of ports
denoted $\mathcal{P} \subseteq \text{Ports}$ defining what can be observed from the component by its environment and then used for synchronization with other components. We describe now the variant of BIP that we use in this thesis. As mentioned previously, several semantics could be provided to BIP components. In this thesis, in particular in Part II, the syntactic description of a component is represented by an extended labeled transition system (see Definition 2.1.2), and its semantics is described as an LTS (see Definition 2.1.7).

**Definition 2.1.6 (Atomic component)** An atomic component $K$ on an interface $\mathcal{P}$ is an extended labeled transition system $(\mathcal{TS}, X, g, f)$, where $\mathcal{TS} = (Q, q^0, \mathcal{P}, \rightarrow)$.

Note that some variables of $X$ are associated to the ports $\mathcal{P}$ of $K$. We now define the semantics of an atomic component.

**Definition 2.1.7 (Semantics of atomic component)** The semantics of $K = (\mathcal{TS}, X, g, f)$, where $\mathcal{TS} = (Q, q^0, \mathcal{P}, \rightarrow)$, is a labeled transition system $((Q \times \mathcal{V}_X), (q^0 \times v^0_X), \mathcal{P}, \hookleftarrow \rightarrow)$ such that:

- $(Q \times \mathcal{V}_X)$ is a set of states, where $\mathcal{V}_X$ denotes the set of valuations of variables $X$.
- $\hookleftarrow \rightarrow$ is the set including transitions $(q, v_x) \xrightarrow{\mathcal{P}} (q', v_{x'})$ where, $v_x, v_{x'} \in \mathcal{V}_X$ such that $g_t(x) \land (x' = f_t(x))$ holds for some $t = (q, p, q') \in \rightarrow$.

At the semantic level some transitions of the ELTS may disappear in the computed labeled transition system and thus some control states of the ELTS may become unreachable. This is due to the fact that the semantics is computed taking into account the possible valuations of the variables and the soundness of the guards associated to transitions.

To distinguish the syntactic and the semantic level of a given component, in the rest of this thesis, we denote by control states, the states of the extended labeled transition systems and by $\rightarrow$ its set of transitions. Then we denote by only states the states of the corresponding computed labeled transition system and by $\hookleftarrow \rightarrow$ the set of its transitions.

Figure 2.6 shows an example of an atomic component, with an interface consisting of two ports in with $x$ as associated variable and the port out with $y$ as associated variable. The behavior of the component represents two control states $s_1$ and $s_2$. The transition labeled by the port in takes place if the component is in state $s_1$ and if $x > 0$. Then the variable $y$ is computed from $x$ according to the function $f$.

![Atomic Component](image)

Figure 2.6: Atomic Component.
2.1.4 Glues in BIP: Interaction models

In BIP, the interaction layer is defined by a set of possibly hierarchical connectors \[\text{[BS07a, BS08b]}\]. Connectors are used to specify possible interaction patterns between the ports of components. Interactions in BIP are used to allow synchronization and communication between components.

Connectors

Composition of components allow to build a system as a set of components that interact by respecting constraints of an interaction model. In BIP interactions are structured by connectors. A connector is a macro notation for representing sets of related interactions in a compact manner. To specify the interactions of a connector, two types of synchronizations are defined:

- strong synchronization or rendez-vous, when the only interaction of a connector is the maximal one, i.e., it contains all the ports of the connector.
- weak synchronization or broadcast, when interactions are all those containing a complete port which initiate the broadcast.

To characterize these two types of synchronizations, a connector may associate to the ports it connects two types. A trigger port of a connector is a complete port which can initiate an interaction without synchronizing with other ports of the connector. It is represented graphically by a triangle. The second type is synchron port of a connector which is an incomplete port, hence needs synchronization with other ports, and is denoted by a circle.

![Figure 2.7: Example of Connectors.](image)

Let \( \gamma \) be a connector connecting a set of ports \( \{p_i\}_{i=1}^{n} \), then \( \gamma \) is defined as follows:

**Definition 2.1.8 (Connector)** A connector \( \gamma \) is defined as a tuple \( (p_\gamma[x], P_\gamma, \delta_\gamma) \) where:

- \( p_\gamma[x] \) is a port called the exported port of \( \gamma \) with \( x \) as associated variable,
- \( P_\gamma = \{p_i[x_i]\}_{i=1}^{n} \) is the set of connected ports called the support set of \( \gamma \). These ports are typed by the information whether they are trigger or synchron. \( x_i \) is a variables associated to \( p_i \).
- \( \delta_\gamma = (G, U, D) \) where,
2.1. THE BIP MODELING FRAMEWORK

- $G$ is a guard of $\gamma$, an arbitrary predicate $G(\{x_i\}_{i \in I})$.
- $U$ is an upward update function of $\gamma$ of the form, $x := F^u(\{x_i\}_{i \in I})$.
- $D$ is a downward update function of $\gamma$ of the form, $\cup_{p_i} \{x_i := F^d_{x_i}(x)\}$.

The intuition behind the notion of exported port, as illustrated in Figure 2.7, is that a connector allows to relate a set of inner ports (of connected components) to a new port (the exported port) which allows to provide a notion of encapsulation by defining the interface of the obtained composite component. Besides, the notion of the exported port, allows the connector to be used as a port by other connectors, and thus create hierarchical components. The exported port represents set of interactions rather than a single interaction.

Note that the support set defines ports of distinct components, which means that a connector connects at most one port of each component. The variables associated to ports of a connector $\gamma$ as well as its upward and downward functions defined on these variables allow the exchange of data between components connected by $\gamma$. Local variables of $\gamma$ can be associated with its exported port $p_\gamma$.

The interactions of a connector are all the subsets of its ports which contain at least one trigger, and in addition the set of all its ports, representing the maximal interaction of the connector, taking into account the soundness of the guard associated to the connector and the guards associated to transitions of the connected components.

**Definition 2.1.9 (Interaction of a connector)** Given a set of components $\{K_i\}_{i=1}^n$, where $K_i = (Q_i, q_i^0, P_i, \longrightarrow, X_i, g_i, f_i)$. A connector $\gamma = (p_\gamma[x], \mathcal{P}_\gamma, \delta_\gamma)$, $\delta_\gamma = (G, U, D)$ connecting these components, such that $\mathcal{P}_\gamma \cap \mathcal{P}_i = \{p_i\}$. An interaction $a$ defined on a set of ports $\{p_j\}_{j=1}^J \subseteq \mathcal{P}_\gamma$ is an interaction of $\gamma$ if one of the following conditions holds:

- $\exists j \in J$ such that $p_j$ is trigger;
- $\forall j \in [1, J]$, $p_j$ is sychron and $\{p_j\}_{j=1}^J = \mathcal{P}_\gamma$.

In both cases the interaction $a$ has a guard, an upward and a downward functions denoted $(G_a, U_a, D_a)$, where we suppose that they are respectively obtained by a projection of $G$, $U$ and $D$ on the variables of the ports involved in $a$.

We denote by $I(\gamma)$ the set of interactions of $\gamma$.

Thus like a connector, an interaction $a$ is of the form $(p_a, \mathcal{P}_a, (G_a, U_a, D_a))$, where $p_a$ is the exported port of $a$, $\mathcal{P}_a$ the set of ports involved in $a$. In this thesis to simplify notation, an interaction $a$ defined on the set of ports $\{p_1, \ldots, p_J\}$ is denoted by the expression $a = p_1, \ldots, p_J$, where we abstract away its guard and function if we do not need to explicit them.

**Example 2.1.10** Example of connectors is depicted in Figure 2.7 In (a), the connector $\gamma_a$ relates the ports $p_1$ and $p_2$, which it defines as sychron, to an exported port $p_{\gamma_a}$. In this connector, if we suppose that there is no variables, then the only feasible interaction is $p_1p_2$ which is also the interaction represented by the exported port $p_{\gamma_a}$. In (b), the interaction between $p_1$ and $p_2$ is asymmetric as $p_1$ is a trigger and can occur alone, even if $p_2$ is not possible. Nevertheless, the occurrence of $p_2$ requires
the occurrence of $p_1$. Thus, if no variables are defined, the interactions defined by $\gamma_b$ are $p_1$ and $p_1p_2$. Thus $p_{b_1}$ represents both interactions.

A connector, as well as a set of connectors, define a structural description of a possible set of interactions. As interaction models are defined as a set of interactions, then a structural description of interaction models could be given as a set of connectors.

**Definition 2.1.11 (Interaction model)** A set of disjoint connectors with distinct exported ports and disjoint interfaces $\{P_i\}_{i=1}^n$ defines an interaction model $\mathcal{I}$ that is defined on $\bigcup_{i=1}^n P_i$ which we call the support set of $\mathcal{I}$ and we denote $S_{\mathcal{I}}$ and which defines an interface $P_{\mathcal{I}} = \{p_\gamma | \gamma \in \mathcal{I}\}$.

The interface of the component resulting from a composition using $\mathcal{I}$ is defined by $P_{\mathcal{I}}$. The variables associated to an interaction model, denoted $X_{\mathcal{I}}$, are the set of variables associated with the ports $P_{\mathcal{I}}$ of the interface of $\mathcal{I}$. The interactions of $\mathcal{I}$, denoted $I(\mathcal{I})$, is the set of interactions defined by the set of its connectors, i.e., $I(\mathcal{I}) = \bigcup_{\gamma \in \mathcal{I}} I(\gamma)$.

Note that connectors in an interaction model are not required to have pairwise disjoint support sets, this means that a port may be connected by several connectors.

**Structured connectors**

So far we have seen a notation for connectors, which are essentially flat, i.e., having types (triggers and synchrons) associated to the individual ports (support set) only. However, connectors sometimes need to be structured, i.e., having types associated to groups of ports. This is necessary to represent some interactions, which otherwise cannot be represented by a flat connector. Structured connectors are created by the combined mechanism of exporting port from a connector and instantiating connectors, where a port of the connector is an exported port of another instantiated connector. Figure 2.8 shows an example of a structured connector. The connector $\gamma_0$ relates the port $p_0$ (trigger) of the *Sender* component with the port $p_1$ (synchron) of the *Receiver* 1 component, and exports the port $p_2$. It represents a set of interactions involving respectively the port sets $\{\{p_0\}, \{p_0p_1\}\}$. $\gamma_1$ is a structured connector joining the port $p_2$ (trigger) of connector $\gamma_0$ with the port $p_3$ (synchron).
of the Receiver\textsuperscript{2} component and exports the port \( p \). It represents the set of interactions involving respectively the port sets \( \{ \{ p_0 \}, \{ p_0, p_1 \}, \{ p_0, p_3 \}, \{ p_0, p_1, p_3 \} \} \).

### 2.1.5 Composition of components in BIP

Composition of components in BIP is performed using interaction models.

**Definition 2.1.12 (Composite component)** A composite component consists of a composition of a set of components \( \{ K_i \}_{i=1}^n \) with disjoint interfaces \( \{ \mathcal{P}_i \}_{i=1}^n \) using an interaction model \( \mathcal{I} \) on \( \bigcup_{i=1}^n \mathcal{P}_i \). Such a component is denoted \( \mathcal{I} \{ K_i \}_{i=1}^n \).

Note that the interface of the composite component is defined by the interface of the used interaction model \( \mathcal{P}_I \).

**Definition 2.1.13 (Flat and Hierarchical component)** A component is called flat if it is atomic or of the form \( \mathcal{I} \{ K_i \}_{i=1}^n \), where all \( K_i \) are atomic components. A component is called hierarchical if it is not flat.

To compose a set of components \( \{ K_i \}_{i=1}^n \), it is assumed that their respective interfaces are pairwise disjoint, i.e., for any two \( i \neq j \) from \( 1..n \) we have \( \mathcal{P}_i \cap \mathcal{P}_j = \emptyset \).

![Figure 2.9: Semantics of a composition.](image)

We now provide how to compute the semantics of a composite component. First, we propose in Definition 2.1.14 a syntactic composition, allowing to transform a composite component \( \mathcal{I} \{ K_i \}_{i=1}^n \), given by a set of atomic components defined by extended labeled transition systems, into a new atomic component \( K \) defined also as an extended labeled transition system. Then, the semantics of the composition is computed as the semantics of the atomic component \( K \). In this thesis, as we are interested in describing the behavior of components as extended labeled transition systems and our interaction models, in particular in Part II, are defined on ELTS, thus we choose to compute the semantics only at the level of atomic components (ELTS) (see Figure 2.9).

**Definition 2.1.14 (Syntactic composition)** The syntactic composition of \( n \) components \( \{ K_i \}_{i=1}^n \), where \( K_i = (TS_i, X_i, g_i, f_i) \), with respect to an interaction model \( \mathcal{I} \) with an interface \( \mathcal{P}_I \) and a set of variables \( X_I \), is an ELTS \( K = (TS, X, g, f) \) where:
• \( TS = (Q, q^0, \mathcal{P}, \rightarrow) \) is an LTS where:
  - \( Q \) is set of control states, which is the cartesian product of the sets of control states of the composed components \( Q = \prod_{i=1}^{n} Q_i \),
  - \( q^0 = (q^0_1, q^0_2, \ldots, q^0_n) \),
  - \( \mathcal{P} \), the interface of \( K \), defined by the interface of the interaction model, \( \mathcal{P} = \mathcal{P}_I \)
  - \( \rightarrow \) a set of transitions of the form \( t = (q, p_\gamma, q') \), where:
    - \( t = (q_1, \ldots, q_n), q' = (q'_1, \ldots, q'_n) \), \( q_i \) and \( q'_i \) being control states of the \( i^{th} \) component.
    - \( p_\gamma \in \mathcal{P}_I \) is an exported port of a connector \( \gamma \in \mathcal{I} \), corresponding to an interaction \( \alpha \) of \( \gamma \), such that there exists a subset \( J \subseteq \{1, \ldots, n\} \) of components with transitions \( \{(q_j, p_j, q'_j)\}_{j \in J} \) and \( \alpha = \{p_j\}_{j \in J} \).
    - if \( j \notin J \), \( q'_j = q_j \). That is, the control states from which there are no transitions labeled with ports in \( \alpha \), remain unchanged.

• \( X = \bigcup_i^n X_i \cup X_I \) the union of the sets of variables of the composed components and the interaction model,

• \( g \) associates with every transition \( t = (q, p_\gamma, q') \), corresponding to an interaction \( \alpha = \{p_j\}_{j \in J} \) of a connector \( \gamma \in \mathcal{I} \), a guard defined by \( g = (\bigwedge_{j \in J} g_j) \land G_\alpha \). Intuitively, the guard of the new obtained transition takes into account the guards of the composed transitions as well as the guard of the connector used to compose them.

• \( f \) associates with every transition \( t = (q, p_\gamma, q') \), corresponding to an interaction \( \alpha = \{p_j\}_{j \in J} \) of a connector \( \gamma \in \mathcal{I} \), a function \( f = U_\alpha; D_\alpha; [f_j]_{j \in J} \). That is, the computation starts with the execution of \( U_\alpha \) then \( D_\alpha \) followed by the execution of all the functions \( f_j \) in some arbitrary order. The result is independent of this order as components have disjoint sets of variables.

Note that components which are not involved in an interaction do not move when it is fired.

\[ \gamma = (p, \{p_1, p_2, \ldots, p_k\}, G, U, D) \]
\[ \alpha \in \gamma \land \alpha = p_1p_2 \ldots p_k \]

Figure 2.10: Composition of Components.
2.1. THE BIP MODELING FRAMEWORK

Figure 2.11 depicts an example of composition of BIP components. The connector \( \gamma \) connects the set of ports \( \{p_i\}_{i=1}^k \). The example depicts the transition resulting from an interaction \( \alpha \) of \( \gamma \). \( \alpha \) corresponds to a rendez-vous between all ports of \( \gamma \). The obtained transition is then labeled by \( p \) the exported port of \( \gamma \), its guard is then computed from the guard \( G_\alpha \) and the guards of the transitions labeled by the ports involved in \( \alpha \). Similarly, the function of transition labeled by \( p \) is computed by applying first the upward function of \( \gamma \), then its downward function and finally the functions corresponding to the synchronized transitions.

2.1.6 Priorities in BIP

Previously, we have focused on the first two layers of the BIP framework, namely behavior and interaction. Now let us focus on priority.

Priorities are a powerful tool for restricting nondeterminism. They allow straightforward modeling of urgency and scheduling policies for distributed systems. For example, execution constraints like run to completion and synchronous execution can be modeled by priority models on threads.

**Definition 2.1.15 (Priority)** A priority order \( \prec \) on a set of interactions is a strict partial order on these interactions.

A priority order \( \prec \) on an interaction model \( \mathcal{I} \) is a strict partial order on the interactions of \( \mathcal{I} \).

We define now the notion of composite component taking into account priority.

**Definition 2.1.16 (Composite component)** A composite component consists of a composition of a set of components \( \{K_i\}_{i=1}^n \) with disjoint interfaces \( \{P_i\}_{i=1}^n \) using an interaction model \( \mathcal{I} \) on \( \bigcup_{i=1}^n P_i \) and a priority order \( \prec \) on \( \mathcal{I}(\mathcal{I}) \). Such a component is denoted \( \mathcal{I}, \prec \{K_i\}_{i=1}^n \).

The pair \((\mathcal{I}, \prec)\) is denoted \( \mathcal{I}, \prec \) and it is a glue as it describes how to compose a set of components so as to make them interact.

In this thesis, we only use priorities in Part III where components are defined as simple labeled transition systems with neither variables nor guards. Thus the semantics of an atomic component is itself. Similarly the semantics of a composite component built as a composition of a set of LTS using an interaction model \( \mathcal{I}, \prec \) is also an LTS obtained by applying first \( \mathcal{I} \) then \( \prec \) to the obtained component.

**Definition 2.1.17 (Semantics of a priority order)** A priority order \( \prec \) defines an operator that associates with an LTS \( TS = (Q, q^0, P, \rightarrow) \) on an LTS \( TS' = (Q, q^0, P, \rightarrow_\prec) \) where \( \rightarrow_\prec \) is the least transition relation satisfying the following rule:

\[
q^1 \xrightarrow{a} q^2 \quad \text{if } b \in P, (a < b \land q^1 \xrightarrow{b}) \quad q^1 \xrightarrow{a \prec} q^2
\]

Figure 2.11 depicts an example of composition of BIP components with priorities. Two interactions between the set of \( k \) components are defined. In this example to simplify notation we suppose
that transitions are labeled by interactions, and not by the exported port of the corresponding connector. In the semantics of the composition and with respect to priorities, stating that $\alpha_2 \prec \alpha_1$, only the transition corresponding to the interaction with higher priority appears.

Note that the priorities we use in this thesis are called static. In BIP another type of priorities called dynamic is also defined [BBBS08, GS04], where priorities between interactions of a given component $K$, depend on its state.

We now provide an example showing the application of priorities to enforce execution constraints on a composite component.

**Example 2.1.18 (Mutual Exclusion)**

In this example, we show the enforcement of mutual exclusion, a very common execution constraint, needed when we have multiple components (tasks) sharing a single resource similar to the famous example of the dining philosophers. In the example shown in Figure 2.12, we have two identical tasks, $T_1$ and $T_2$, modeled as BIP components. The control states of the $i$-th task are $I_i$ (Idle), $R_i$ (Ready) and $E_i$ (Executing). The actions (ports) are $a_i$ (activate), $b_i$ (begin) and $f_i$ (finish). Each task is initialized to state $I_i$, from where it can activate through the action $a_i$ and become ready ($R_i$) for execution. The start of execution is marked by the action $b_i$, by which the task acquires the resource and moves to the execution state ($E_i$).

The interaction model allows every action to be activated independently. Under mutual exclusion, a task cannot acquire the resource if the other task is already in the execution state. As in this simple example we suppose that only unitary actions can occur, violation of mutual exclusion would mean that after the occurrence of $b_1$, $b_2$ occurs before $f_1$. Then we can enforce mutual exclusion by the priority model, which assigns the actions to obtain the resource, lower priority than the actions to release the resource, i.e., $b_1 \prec f_2$ and $b_2 \prec f_1$. For $T_1$, the priority $b_1 \prec f_2$ prevents it from obtaining the resource, unless $T_2$ releases it (by the action $f_2$). Similarly, $T_2$ is restricted by $b_2 \prec f_1$ as long we suppose that $f_i$ is always enabled after $b_i$. 

35
2.2 Contract framework concepts

A main part of this thesis aims at the definition of a scalable design and verification methodology for systems of components based on contracts. Contracts are design constraints for implementations which are maintained throughout the development and life cycle of the system. We describe in this section the basic concepts about contracts. In particular, we introduce a formal definition of the generic contract framework presented in [QG08a] and a description of the component framework on which it is based. We also provide key relations and properties required to reason about contracts namely satisfaction and dominance.

The generic notion of contract framework, described in [QG08a], relies on a notion of component framework supporting hierarchical components as well as some powerful mechanisms to reason about

Figure 2.12: Example: Priorities to enforce Mutual Exclusion.

When both tasks are at state $R_i$, either of them can acquire the resource in a non-deterministic fashion. However, static priority rules can be used to prioritize the tasks in any desired order. For example, $b_2 \prec b_1$ sets a higher priority for $T_1$ to acquire the resource, compared to $T_2$.

We can also define mutual exclusion in the case where we suppose that actions of different processes may occur independently. This can be expressed by an interaction model that allows any pairs of actions involving the two processes except the pairs $b_1 b_2$ of course. The same priority rules can be used. If we suppose that $f_1$ may be disabled for a while after the occurrence of $b_i$, priorities are not sufficient anymore. We need the so-called timed priorities as they have been proposed in [BS00].

Maximal Progress Priority in Connectors A particular priority rule, that favors, among the enabled interactions of a connector, the maximal one, i.e., the one with maximum number of ports, is known as maximal progress priority. This can be explicitly represented through priority rules amongst the interactions, of the form $p_1 \prec p_1 p_2$, where $p_1$ and $p_1 p_2$ are interactions of the same connector. As an example, maximal progress is necessary to model a broadcast. Maximal progress is implicitly assumed in connectors for their compact and natural representation.
composition. We therefore introduce next the notions a component framework must define.

**Definition 2.2.1 (Component framework)** A component framework is a structure of the form \((\mathcal{K}, \mathcal{GL}, \circ, \cong)\) where:

- \(\mathcal{K}\) is a set of components — describing their behavior or properties. Each component \(K \in \mathcal{K}\) has as its interface a set of ports, denoted \(\mathcal{P}_K\).

- \(\cong \subseteq \mathcal{K} \times \mathcal{K}\) is an equivalence relation. In general, this equivalence is derived from equality or equivalence of semantic sets.

- \(\mathcal{GL}\) is a set of glue (composition) operators. Operators \(gl \in \mathcal{GL}\) are partial functions \(2^\mathcal{K} \rightarrow \mathcal{K}\) transforming a set of components into a new component. Each \(gl\) is defined on a set of ports \(S_{gl}\) — of the original set of components, called its support set — and defines a new interface \(\mathcal{P}_{gl}\) — on the new component, called its exported interface (see Definition 2.1.11). Thus, \(K = gl\{K_1, \ldots, K_n\}\) is defined if \(K_1, \ldots, K_n \in \mathcal{K}\) have disjoint interfaces, \(S_{gl} = \bigcup_{i=1}^n \mathcal{P}_{K_i}\) and the interface of \(K\) is \(\mathcal{P}_{gl}\), the exported interface of \(gl\).

- \(\circ\) is an operation on \(\mathcal{GL}\) allowing to compose glues. It is such that \((\mathcal{GL}, \circ)\) is a commutative monoid. Formally, \(gl \circ gl'\) is defined on \((S_{gl} \cap S_{gl'}) \cup (S_{gl} \setminus \mathcal{P}_{gl'}) \cup (S_{gl'} \setminus \mathcal{P}_{gl})\) and defines as interface \((\mathcal{P}_{gl} \cup \mathcal{P}_{gl'}) \setminus S_{gl \circ gl'}\). Note that \(gl \circ gl'\) must be defined even if \((S_{gl} \cap S_{gl'}) \neq \emptyset\). Furthermore, this operation must be consistent with \(\cong\) in the sense that \(gl\{gl\{K^1\}, K^2\} \cong (gl \circ gl')\{K^1 \cup K^2\}\) for any sets of components \(K^i\) such that all terms are defined.

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

Note that the operation \(\circ\) over glues is commutative not in the order of applying composed glues but in the sense that the resulted glue of their composition is the same. In this definition of component framework, the notion of component is intentionally kept abstract to encompass different frameworks. It may be for example a labeled transition system or an extended labeled transition system as defined previously. For example, components are defined in the framework defined in [QG08a] as Modal Transition Systems (MTS) and they could be also defined as BIP components which include a structural layer. Note that in this definition, there is no explicit distinction between atomic and hierarchical components, if there is a way to represent an hierarchical component as an atomic one. In many frameworks that work at the semantic level, they are the same.

Figure 2.13 shows how hierarchical components (the colored ones) are built from atomic ones (the white components). It also illustrates the flattening property the composition model. Note that the representation of glues is just one among other possible sets of glues. Dashed lines show how the exported interface is defined based on inner ports and glues.

Note that BIP framework as defined previously represents such a component framework with its definition of components and glues. In this thesis, we are interested in a variant of BIP defining such a component framework.
2.2. CONTRACT FRAMEWORK CONCEPTS

2.2.1 Contract frameworks

We give in this section the different notions presented in [QG08a] allowing to use contracts to reason about components and their properties. In particular, how to build a contract framework which provides basic rules to reason about systems of components.

Before giving the formal definition of contract framework, we introduce the notion of context, to describe how a component may be connected to the rest of the system and to express a property of the environment. Thus, a context limits the way in which a component may be further composed.

Definition 2.2.2 (Context) A context for an interface $P$ is a pair $(E, gl)$ where $E \in \mathcal{K}$ a component defined on $P_E$, and where we suppose $P \cap P_E = \emptyset$ and $gl$ is defined on $P \cup P_E$.

Definition 2.2.3 (Contract framework) A contract framework is a tuple $(\mathcal{K}, GL, \circ, \sim, \{\sqsubseteq_{E,gl}\}, \preceq)$ where:

- $(\mathcal{K}, GL, \circ, \sim)$ is a component framework.
- $\preceq \subseteq \mathcal{K} \times \mathcal{K}$ is a conformance relation relating components with the same interface. Given $K_1, K_2 \in \mathcal{K}$, for $K_1 \preceq K_2$ we may say $K_1$ conforms to $K_2$.
- $\{\sqsubseteq_{E,gl}\}$ is a refinement under context relation parameterized by a context. Given a context $(E, gl)$ for an interface $P$, $\sqsubseteq_{E,gl}$ is a preorder over the set of components on $P$ which is expected to be compositional. It is a preorder such that for any $K_1, K_2$ on the same interface $P$ and for any context $(E, gl)$ for $P$, $K_1 \sqsubseteq_{E,gl} K_2 \implies gl\{K_1, E\} \preceq gl\{K_2, E\}$.

The notion of refinement and substitutability is recognized as being a fundamental requirement [DHJP08] in the context of component and contract-based approaches. Conformance is a kind of refinement with respect to a given specification and it relates properties of closed systems, where a closed system is a component that we cannot or do not want to further compose. Open systems are components that may be composed with an unknown environment. Even though a closed system is not intended to be composed anymore, it has an interface allowing not to observe its behavior and thus to define properties of a closed system on ports of this interface.
Refinement under context is usually considered as a derived relation and chosen as the weakest relation implying conformance.

**Example 2.2.4** Typical notions of conformance \( \preceq \) are trace inclusion and simulation. For these notions of conformance, refinement under context (denoted \( \sqsubseteq \)) is usually defined as the weakest pre-order included in \( \preceq \) that is compositional:

\[
K_1 \sqsubseteq_{E, gl} K_2 \triangleq gl\{K_1, E\} \preceq gl\{K_2, E\}
\]

Note that there are cases where a stronger notion of refinement under context allows more powerful reasoning.

**Example 2.2.5** Conformance itself is another candidate for refinement under context if it is preserved by composition, that is, if: \( K_1 \preceq K_2 \) implies \( gl\{K_1, E\} \preceq gl\{K_2, E\} \) for any \( E \).

However, defining refinement under context as \( K_1 \sqsubseteq_{E, gl} K_2 \triangleq K_1 \preceq K_2 \) means in fact not taking the environment into account, thus it is of limited interest. In some cases, as the frameworks presented in [QG08a], conformance actually corresponds to refinement in any context. In the contract framework we propose in Chapter 4, our definition of conformance is defined as refinement in a particular context.

Now we can define the notion of contract which allows to describe properties that a component should offer in a given context.

**Definition 2.2.6 (Contract)** A contract \( C \) for an interface \( P \) consists of:

- a context \( E = (A, gl) \) for \( P \) where \( A \) is called the assumption
- a component \( G \) on \( P \) called the guarantee

We write \( C = (A, gl, G) \) rather than \( C = ((A, gl), G) \). \( gl \) implicitly defines the interface of the environment while \( A \) expresses a constraint on it and \( G \) a constraint on the refinements of \( K \). The “mirror” contract \( C^{-1} \) of \( C \) is \( (G, gl, A) \).

A contract \( C = (A, gl, G) \) defines a closed system namely \( gl\{A, G\} \) as a composition of its assumption and guarantee using its glue. The exported interface of \( gl \) defines the interface of the closed system on which properties are expressed.

**Definition 2.2.7 (Satisfaction of contract)** A component \( K \) satisfies a contract \( C = (A, gl, G) \), denoted \( K \models C \), if and only if \( K \sqsubseteq_{A, gl} G \). \( K \) is called a possible implementation of \( C \).

Intuitively, a component \( K \) satisfies a contract \( C = (A, gl, G) \), if \( K \) and its environment are composed by the glue \( gl \) and \( K \) behaves according to \( G \) provided that the environment behaves according to \( A \).
2.2. CONTRACT FRAMEWORK CONCEPTS

Example 2.2.8 Suppose that components are LTS, conformance is simulation and refinement under context is the usual derived notion defined in Example 2.2.4. Suppose also that composition gl is defined as the synchronization between actions with the same letter and interleaving of others. Then Figure 2.14 shows $K$, $A$ and $G$ such that $K$ satisfies the contract $(A, gl, G)$. Indeed, even though $K$ does not simulate $G$ (after $b$ is fired it offers $c$ instead of $a$), it still behaves like $G$ in the context of $(A, gl)$, which prevents $b$ from taking place.

2.2.2 Dominance

To reason about contracts, in the proposed contract framework, an additional relation called dominance is introduced. Dominance means for contracts what refinement means for components. Contract $C$ is said to dominate contract $C'$ if every implementation of $C$ — i.e., every component satisfying $C$ — is also an implementation of $C'$. Intuitively, this is achieved by a $C$ that has a stronger promise or a weaker assumption than $C'$.

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

Definition 2.2.9 (Binary dominance) Let $C$ and $C'$ be two contracts for the same interface $\mathcal{P}$ and the same glue $gl$, with $C = (A, gl, G)$ and $C' = (A', gl, G')$. $C$ dominates $C'$ iff:

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

A component which satisfies the concrete contract $C$ does also satisfy the more abstract contract $C'$.

There are several ways to “refine” a contract $C'$. A typical way consists in strengthening the guarantee $G$. Another typical situation is the one of a hierarchical component. Then $C'$ would be contract associated with the higher level of hierarchy and $C$ is defined implicitly by a set of contracts and the glue $gl$ defining their composition.

In this case, we have two options: define for each glue $gl$ a composition of contracts $\tilde{gl}$ or define dominance explicitly for such a situation. Here we choose the second option, as composition of contracts is not possible in all cases of interest for us, and moreover not even desirable for achieving efficient verification. We thus need a broader notion of dominance than the binary version, that is
dominance defined directly for a set of contracts \( \{C_i\}_{i=1}^n \) and a contract \( C \) to be dominated w.r.t a composition operator \( gl_I \).

In order to hide ports of the lower-level contracts which do not appear at the interface of the top-level contract, the constraints on the composition operators are relaxed by only requiring that they agree on their common ports. For this, a notion of projection of a component \( K \) onto a subset \( \mathcal{P}' \) of its interface is needed, denoted \( \Pi_{\mathcal{P}'}(K) \), which is quite natural and must preserve the following properties.

**Definition 2.2.10 (Projection)** If \( \Pi \) is a projection, then for any components \( K_i \) and \( A \) on disjoint interfaces \( \mathcal{P} \) and \( \mathcal{P}_A \), and any composition operator \( gl \) on \( \mathcal{P} \cup \mathcal{P}_A \):

1. for \( \mathcal{P}' \subseteq \mathcal{P} \) and for any \( gl_1, gl_2 \) with \( S_{gl_1} = \mathcal{P}' \cup \mathcal{P}_A, S_{gl_2} = \mathcal{P} \setminus \mathcal{P}' \) s.t. \( gl = gl_1 \circ gl_2 \):

   \[
   (K_1 \subseteq_{A,gl} K_2 \land \Pi_{\mathcal{P}'}(K_2) \subseteq_{A,gl_1} G) \implies \Pi_{\mathcal{P}'}(K_1) \subseteq_{A,gl_1} G
   \]

   Note that \( G \) is defined on \( \mathcal{P}' \).

2. \( \forall \mathcal{P}'_A \subseteq \mathcal{P}_A \) and for any \( gl_1, gl_2 \) on \( \mathcal{P}_A \), \( S_{gl_1} = \mathcal{P} \cup \mathcal{P}_A, S_{gl_2} = \mathcal{P}_A \setminus \mathcal{P}'_A \) s.t. \( gl = gl_1 \circ gl_2 \):

   \[
   (K \subseteq_{\Pi_{\mathcal{P}'_A}(A),gl_1} G) \implies K \subseteq_{A,gl} G
   \]

   Note that \( G \) is defined on \( \mathcal{P} \).

These properties state that ports of the component (and symmetrically of the environment) which do not appear in interactions with the environment (resp. the component) may be abstracted away when checking refinement.

We now provide the definition of dominance relating a set of contracts.

**Definition 2.2.11 (Dominance)** Let \( C \) be a contract for \( \mathcal{P} \), \( \{C_i\}_{i=1}^n \) a set of contracts defined on disjoint interfaces \( \{\mathcal{P}_i\}_{i=1}^n \) and a glue defined on \( \bigcup_{i=1}^n \mathcal{P}_i \). Then \( \{C_i\}_{i=1}^n \) dominates \( C \) w.r.t. \( gl_I \) iff for any set of components \( \{K_i\}_{i=1}^n \):

\[
(\text{if for every } i \in [1,n], K_i \models C_i, \text{ then } \Pi_{\mathcal{P}}(gl_I\{K_1,\ldots,K_n\}) \models C)
\]

Intuitively, a set of contracts \( \{C_i\}_{i=1}^n \) dominates a contract \( C \) w.r.t. a glue \( gl_I \) if and only if any set of components satisfying the contracts \( C_i \), when composed using \( gl_I \), makes a component satisfying \( C \).

The definition we have given is semantic, and concretely one does not want to manipulate implementations in order to establish dominance. More generally, what are the additional proof rules and properties that one needs in order to reason within contract frameworks? The following chapter answers those questions.
Part II

A Contract Framework for Reasoning about Safety and Progress
Chapter 3

Contract-Based Verification Approach

It is a well-known fact that formal verification based on model-checking suffers from the state space explosion problem. Compositional design and verification are mandatory for making verification feasible. One possibility is to infer global properties of a system from properties of its subsystems. Instead of verifying globally the entire system, compositional approach first decomposes it into small subsystems and verifies each of them individually. The size of a subsystem is often quite smaller compared to the size of the whole system, hence there is less risk of explosion of state space. Then, properties of the global system are inferred from the verified properties of its subsystems.

In this chapter, we present a compositional design and verification methodology. In particular, we focus on an approach allowing to verify some top level requirement of a given system represented by a composition of a set of components (subsystems). We propose to infer this global property to a set of “sub-properties” associated to each of the components.

The idea is to deduce the satisfaction of this global property from the satisfaction of the “sub-properties” at the level of each component. Such an approach is in general possible if the set of subsystems are independent. However, if it is not the case, when building their “sub-properties”, one would take into account the interference of this subsystem with the rest of the system. Thus, we have chosen to use contracts, so as one could describe properties of the subsystems and also properties of their environment that is the rest of the system.

Moreover, to describe how each subsystem interacts with the rest of the system, and as mentioned previously, we rely on a notion of contract allowing to describe the glues used to compose components. Thus, the proposed approach does not only encompass properties of systems but also how they interact and how they are composed to their environment.

This chapter is organized as follows: Section 3.1 describes the different steps of our methodology. In particular how we use the notion of contract framework and it relations. Then, we describe how to extend our methodology to verify recursively defined systems, that is systems which can be built according to a grammar of components. This allows to encompass systems of arbitrary size. Section 3.2 details first how we can establish the soundness of our methodology. Then, properties that the component framework has to ensure so that we can apply the methodology. In Section 3.3 we describe the property that we use to prove dominance which is a key notion in our methodology as
it allows to infer properties of some top-level contract from the properties of a set of inner contracts with respect to some glue operator.

### 3.1 Design and verification methodology

In this section we propose first a design methodology which is compositional and incremental and defined for the generic contract framework described in Section 2.2.1. In particular, it uses its notion of contracts and its relations of conformance, dominance and satisfaction.

Then, we describe how we extend this methodology to verify properties of systems with arbitrary size. The idea is to describe these systems by a grammar of components, allowing to describe the system by a set of rules, then we adapt the steps of the methodology to these rules.

#### 3.1.1 Methodology

We present now our compositional and incremental design and verification methodology defined for any contract framework that is an instance of the generic notion of contract framework as introduced in Section 2.2.

We present the methodology in a top-down fashion. However, it could be also applied in a bottom-up fashion to achieve the same verification purposes but in a different order by using the same reasoning and relations. Note that in general design approaches allow both top-down and bottom-up reasoning.

We consider here the simple case of a unique top-level requirement – that will be denoted \( \varphi \) throughout this section – that is pushed progressively from the overall system into atomic components which we call implementations.

In the entire section we use the notions introduced in Section 2.2 in particular, the notion of contract and the relations conformance, satisfaction and dominance. We first present an overview of the methodology steps, then we detail the different steps separately. Our methodology is given by the following steps:

1. We first provide the property \( \varphi \), which we want that the system \( K \) satisfies in a given context. That is a property of the closed system obtained by the composition of \( K \) through its interface \( \mathcal{P} \) to its environment \( E \).

2. Then, we define a contract \( C \) for \( \mathcal{P} \) which conforms to the property \( \varphi \). That is defining the following:
   - the glue \( gl \) connecting \( K \) to its environment which allows to define the interface of the environment denoted \( \mathcal{P}_A \);
   - a property \( A \) on the environment \( E \) of \( K \) defined on \( \mathcal{P}_A \);
   - a property \( G \) on \( \mathcal{P} \) which \( K \) has to satisfy in the context \((A, gl)\).
3. We define $K$ as a structural composition of a set of subcomponents $\{K_i\}$ using a glue $gl_i$ (see Figure 3.2). Then we associate a contract $C_i$ for each subcomponent $K_i$ and we prove that any set of implementations (components) for $K_i$ satisfying these contracts $C_i$, when composed using $gl_i$, satisfies the top-level contract $C$ and thus guarantees $\varphi$. This corresponds to the verification of the relation of dominance between contracts (see Figure 5.8). This step can possibly be iterated if needed by similarly decomposing each $K_i$;

4. Finally, for components $K_i$, which we do not want to further decompose because for example they are sufficiently simple, we provide implementations $I_i$ and we prove that these implementations satisfy the contracts $C_i$.

![Figure 3.1: Step 2: Conformance.](image)

The first step provides a description of the property $\varphi$, which the designer wants to verify. The designer has to precise in which context, the system $K$ under study has to ensure this property. Indeed, $\varphi$ has to be ensured by $K$ together with its environment and thus the description of $\varphi$ has to be defined on the interface of the closed system defined by the composition of $K$ and its environment.

For this purpose, the next step is based on the conformance relation (see Definition 2.2.3) which is a relation defined on closed systems with the same interface.

In [Qui11], closed systems do not have interfaces as they are not supposed to be connected and their properties are described on the set of internal interactions. Here we propose to define an interface for closed systems in which we export the set of interactions and possibly variables that one wants to use to describe the desired property. Thus, if one wants to describe a property on the set of all internal interactions of a closed system, it has to export by means of an “exported interface” all these interactions. For example, in the Input/Output (I/O) contract framework proposed in [Qui11], input completeness is verified by checking that whenever an output $o$ occurs, a corresponding input $i$ occurs as well, that is only interaction $o.i$ occurs. This is possible by exporting to the interface of the closed system the interaction involving only an output $o$ and verifying that this interaction will never take place.

As previously described, a contract allows to define a closed system built by the composition of $K$ and its environment. A contract on the interface $P$ of $K$, defines a $gl$ describing how $K$ is connected
to its environment, and thus defines the interface of this environment \( P_A \). The contract, describes then a property of the environment \( A \) defined on \( P_A \) and a property \( G \) on the interface \( P \) that \( K \) has to provide in the context \((A, gl)\). In Figure 3.1, \( \phi \) and the closed system defined by the contract \((A, gl, G)\) are defined on the same interface \( P_\phi \). This relation has to be reflexive and transitive. In the frameworks presented in [QG08a] for example, the relation of conformance is defined as inclusion of traces between MTS.

![Figure 3.2: Step 3: \( K \) defined as a composition of \( \{K_i\}_{i=1}^n \) using \( gl_I \).](image)

The third step of our methodology uses the characteristics of the component framework used to build the system in particular its set of glues. In the first part of this step, see Figure 3.2, the designer provides a description of the system \( K \) as a composition of a set of components \( \{K_i\} \) with respect of a given glue \( gl_I \). \( \{K_i\} \) are also called subsystems of \( K \). If the component framework provides the so-called property of flattening, then it is possible to build from the glues \( gl \) and \( gl_I \), a glue \( gl_{\text{flat}} \) relating directly the components \( \{K_i\} \) to the environment of \( K \). That is \( gl_{\text{flat}}(A, K) \cong gl(A, gl_I\{K_1, \ldots, K_n\}) \). In the second part of third step, we define for each subsystem \( K_i \) a contract \( C_i \) describing a property of its environment \( A_i \) and a property \( G_i \) that it is supposed to guarantee in the context of such an environment. Thus, to define such contracts for each subsystem, the designer has to provide a property of the environment of this subsystem, that is the rest of the system, and in particular how it is composed to this environment that is the glue \( gl_i \). These glues \( \{gl_i\} \) must provide a property called structural compatibility with the glue \( gl_{\text{flat}} \) obtained by flattening. This property is needed to relate the glue \( gl_i \) provided in each \( C_i \) to the actual environment of the component \( K_i \). For example, as illustrated in Figure 5.8, the actual environment of the subcomponent \( K_1 \) consists of components \( K_2, K_3 \) and \( A \), the top-level assumption. However, the glue \( gl_1 \) provided in the contract for \( K_1 \) has been defined for an abstract environment \( A_1 \), hence the need for a glue representing the environment of \( K_1 \) as a single component with the same interface as \( A_1 \). More details about this property are provided in Section 3.2.

The preservation of the top level requirement \( \phi \) by the set of contracts \( \{C_i\} \) is guaranteed by the properties of the relation of dominance that must hold between these contracts \( \{C_i\} \) and the global contract \( C \).

Thus a key issue of the third step of the proposed approach is dominance relation (see Definition 2.2.11), as shown in Figure 5.8 once we define a contract, for each subsystem, a dominance relation.
relation has to be checked between these contracts \( \{C_1, C_2, C_3\} \) and the top-level Contract \( C \). Intuitively, dominance means for contracts what refinement means for components as the goal of contract frameworks in general is to reason about contracts and dominance rather than about components and refinement.

The last step of our methodology, is to provide implementations that satisfy contracts of components which are not further refined, at least at the current stage of design (see Figure 3.4). This means, giving a concrete specification of the different subsystems of the global system. As our methodology is incremental, this step can be pushed as far as possible to lower levels. Indeed, for a given subsystem \( K_i \) we can either provide an implementation, which can be described using the same formalism as properties or using any language if the implementation is provided as a program, or it can be further described as a composition of a set of components and for which we apply again the third step of our methodology.

Given a set of implementations \( \{I_i\} \) satisfying the contracts \( \{C_i\} \), then if the different steps of our approach have been applied successfully, means that conformance, dominance and satisfaction have been successfully checked, then the system obtained as a composition of these implementations, that is \( gl_{flat}\{A, I_1, \ldots, I_n\} \), guarantees the property \( \varphi \). The soundness of these results rely on the properties of the relations of the contract framework we use to apply this methodology. These properties will be detailed in the next section.
An additional step is represented on the right hand side: it allows the integration of the system $K$ in an actual environment $E$ while preserving $\varphi$. For this purpose, $E$ must satisfy the mirror contract of $C$ or any contract dominating this mirror contract. This step can be added if the contract framework allows a property called *circular reasoning*, which intuitively allows to infer that if $K$ conforms to $\varphi$ in an environment $A$, than it also conforms to $\varphi$ in any environment refining this environment. Details about this property and its usefulness are described in Section 3.3.1.

![Diagram](image.png)

Figure 3.5: Methodology steps ensuring that $gl\{A, I_1, I_2, I_3\} \preceq \varphi$.

### 3.1.2 Extension to recursively defined systems

In this Section, we extend our design and verification methodology to hierarchically structured and recursively defined systems so that we can handle systems representing component networks of arbitrary size. We propose to apply our methodology to systems that can be defined by a component grammar as follows:

- a set of terminal symbols $\{A, I_1, \ldots, I_k\}$ representing implementations;
- a set of nonterminal symbols $\{S, K_0, K_1, \ldots, K_n\}$ representing hierarchical components; $S$, which defines the top-level closed system, is the axiom;
- a set of rules corresponding to design steps which define each non-terminal either as a composition of subsystems or as an implementation:
CHAPTER 3. CONTRACT-BASED VERIFICATION APPROACH

- $S \rightarrow gl\{A, K_0\}$.

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

Where $S$ is a hierarchical component (a system) and $\{K_i\}_{i=1,...,n}$ are either hierarchical or leaf components for which an implementation is to be given.

Terminal symbols correspond to actual implementations, nonterminal symbols to hierarchical components and rules to design steps. In particular, $K_0$ represents the global system under design, $A$ is a property of its real environment. Thus, $S$ stands for the system along with its environment. Note that a decomposition rule of some non-terminal $K_i$ may also contain a set of the same non-terminal. Thus, to ensure the unicity of ports, we suppose that the name of each port is prefixed by its path from the root symbol and indexed if the rule contains a set of the same non-terminals. Unlike classical network grammars, we use rich composition operators and are not limited to flat regular networks, as for example in [SG89].

We now instantiate the methodology of Figure 3.5 for such component networks. We choose again a top-down presentation. To do so,

The same four steps are presented, namely conformance, decomposition, dominance and satisfaction.

1. formulate a top-level requirement $\varphi$ characterizing the closed system $S$ defined by the system $K_0$ and a property $A$ of its environment;

2. define a contract $C = (A, gl, G)$ associated with $K_0$ and prove that $gl\{A, G\} \preceq \varphi$;

3. define for every non terminal $K_i$ a contract $C_{K_i} = (A_{K_i}, gl_{K_i}, G_{K_i})$ such that for every rule $K_l \rightarrow gl_{\Sigma_l}\{K_j\}_{j \in \Sigma_l}$ having an occurrence of $K_i$ on the right hand side, there exists $gl_{A_i}$ such that $gl_{K_i} \circ gl_{\Sigma_l} = gl_{K_l} \circ gl_{A_i}$. Then, for each $K_i \rightarrow gl_{\Sigma_i}\{K_j\}_{j \in \Sigma_i}$, show that $\{C_{K_j}\}_{j \in \Sigma_i}$ dominates $C_{K_i}$ w.r.t $gl_{\Sigma_i}$.

4. prove that implementations satisfy their contracts: $K_i \rightarrow I_j \implies I_j \models C_{K_i}$.

The steps 3 and 4 can be combined, by proving for each non terminal $K_i$ either satisfaction if it is described by an implementation, or dominance, if it is described by a composition rule.

In the next section we provide a proof that, given an grammar of components, for which all the methodology steps have been successfully applied to verify a property $\varphi$, then any component system built as a derivation tree accepted by this grammar satisfies also $\varphi$.

3.2 Soundness of the methodology

In Section 2.2 we have described the definitions of the different relations used in our methodology, namely conformance, dominance and satisfaction. In this section, we describe how we prove soundness of this methodology using the definitions of these relations provided in the definition of
contract framework. We then, describe a condition a designer has to ensure on glues when defining the contracts for dominance.

3.2.1 Soundness

To prove the soundness of the methodology described in Section 3.1.1, we rely on the properties of the contract framework defined in Section 2.2.

**Theorem 3.2.1** Suppose for a system $K$, a top level property $\varphi$, and a contract $C = (A, gl, G)$ defined on $K$ together with its environment such that $C$ conforms to $\varphi$. Suppose a decomposition of $K$ into $gl_I\{K_1, \ldots, K_n\}$, where for each $K_i$ we associate a contract $C_i = (A_i, gl_i, G_i)$ such that the contracts $\{C_i\}$ dominate $C$ w.r.t $gl_I$ and we provide an implementation $I_i$ satisfying $C_i$. Then, we have:

$$gl\{A, gl_I\{I_1, \ldots, I_n\}\}$$ conforms to $\varphi$

**Proof** According to the definition of dominance (see Definition 2.2.11), $\{C_i\}$ dominate $C$ w.r.t $gl_I$ implies that any components satisfying the contracts $\{C_i\}$ when composed together using $gl_I$ satisfies $C$. As the implementations $\{I_i\}$ satisfy their corresponding contracts, we obtain $gl_I\{I_1, \ldots, I_n\} \sqsubseteq_{A, gl} G$ (according to the Definition 2.2.7 of satisfaction). As we have that refinement implies conformance in our contract framework, this implies that $gl\{A, gl_I\{I_1, \ldots, I_n\}\}$ conforms to $gl\{A, G\}$. Then as $gl\{A, G\}$ conforms to $\varphi$ and as conformance is transitive, we obtain $gl\{A, gl_I\{I_1, \ldots, I_n\}\}$ conforms to $\varphi$. \qed

We can now based on this Theorem 3.2.1 prove the soundness of the extended methodology.

**Theorem 3.2.2** Let $G$ be a grammar such that all methodology steps have been completed to guarantee a requirement $\varphi$. Any component system corresponding to a word accepted by $G$ satisfies $\varphi$.

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

3.2.2 Compatibility of glues

As previously described, the soundness of our methodology relies on the definition of dominance. However, before using dominance, we need first to associate contracts $\{C_i\}$ for subsystems $\{K_i\}$ of $K$, which requires to define the glues $\{gl_i\}$ allowing to compose each subsystem to the rest of the system. As already mentioned in the methodology, these glues have to provide a property of structural compatibility with the glue $gl \circ gl_I$.

**Definition 3.2.3 (Structural compatibility of glues)** Consider two glues $gl$ and $gl_2$. Now suppose $P \subseteq S_{gl}$. $gl$ and $gl_2$ are called compatible if there exists $gl_1$ with $P_{gl} = P_{gl_1}$ and $S_{gl_2} = P \cup P_{gl_2}$.
such that \( gl_1 \circ gl_2 \cong gl \) where by abuse of notation, this equivalence between glues means that \( gl_1 \circ gl_2 \{ K^1 \} \cong gl \{ K^1 \} \) for any set of components \( K^1 \) such that all terms are defined.

In the context of the dominance problem, the goal is to relate the glue \( gl_i \) provided in each \( C_i \) to the actual environment of component \( K_i \), as illustrated in Figure 3.6: \( gl \) is the glue defined in the top-level contract and \( gl_I \) defines how subcomponents are composed. Thus, the actual environment of subcomponent \( K_1 \) consists of components \( K_2 \) to \( K_4 \) and \( A \), the top-level assumption. However, the glue \( gl_1 \) provided in the contract for \( K_1 \) has been defined for an abstract environment \( A_1 \), hence the need for a glue \( gl_{E_1} \) representing the environment of \( K_1 \) as a single component with the same interface as \( A_1 \).

![Figure 3.6: \( gl_{E_1} \) allows relating the glue \( gl_1 \) provided in \( C_1 \) to the actual environment for \( K_1 \).](image)

### 3.3 Proving dominance

The steps of the methodology described in Section 3.1 consists in proving basically three relations, namely conformance, dominance and satisfaction. In general, proving conformance and satisfaction uses analytical methods. For example, when components are LTS and conformance is inclusion of traces, then to prove conformance it is sufficient to prove simulation.

However, to prove dominance, according to the Definition 2.2.11, one has to manipulate concrete implementations of contracts. To avoid this, we propose to use a particular property called circular reasoning in order to prove dominance. Thus, in this section, we first discuss this property and we provide examples for which circular reasoning is not sound. Then, we describe how this property allows to deduce a sufficient condition to prove dominance.

#### 3.3.1 Circular reasoning

Circular reasoning is a powerful property which allows to derive the property of independent implementability. Which means that we do not need to prove that an implementation refines its specification in the actual context in which it is used. This is highly undesirable for at least two reasons: one is that implementations are expected to be very complex, thus manipulating them is likely to be intractable; the other is that whenever a small change occurs in the implementation of a part of the system, all the proofs have to be started all over again. This is why, we have already
3.3. PROVING DOMINANCE

mentionned this property of circular reasoning in the additional step of our methodology related to the refinement of the environment (see the top right-hand side of Figure 3.5). Using circular reasoning, we can prove that a set of given implementations conforms to a given property in any context refining the mirror contract of the top level contract.

To avoid this situation, circular reasoning allows proving refinement using the abstract environment provided by the specifications rather than the concrete one provided by the implementations. Thus, circular reasoning is a property provided by the relation of refinement under context and it is defined as follows:

**Definition 3.3.1 (Circular Reasoning)** If a component $K$ refines $G$ in an abstract context $(A, gl)$ and if $E$ refines $A$ in the abstract context $(G, gl)$, then $K$ refines $G$ in the concrete context $(E, gl)$.

$K \sqsubseteq_{A, gl} G \quad E \sqsubseteq_{G, gl} A \quad \therefore K \sqsubseteq_{E, gl} G$

This property can be proved in a given framework by an induction based on the semantics of composition and refinement [McM99a, Mai03c].

However, circular reasoning is not sound in general. In particular, it is unsound when composition is based on synchronizations (as they exist in e.g. Petri nets or process algebras) or instantaneous mutual dependencies between inputs and outputs (as they exist in synchronous formalisms). For example, in the framework based on Input/Output automata defined in [QG08a], circular reasoning is sound because exactly one behavior has control over each interaction.

Example 3.3.2 explains two reasons for the non validity of circular reasoning for $\sqsubseteq$ which are illustrated in Figures 3.7 and 3.8.

**Example 3.3.2** Suppose, as in the previous examples, that components are LTS and composition $gl$ is defined as the synchronization between actions with the same letter and interleaving of others. Suppose also that conformance is simulation and refinement under context is the usual derived notion:

$K_1 \sqsubseteq_{E, gl} K_2 \triangleq gl\{K_1, E\} \sqsubseteq gl\{K_2, E\}$

The examples in Figures 3.7 and 3.8 are both counterexamples to the circular rule, that is: $K \sqsubseteq_{A, gl} G$ and $E \sqsubseteq_{G, gl} A$ but $K \not\sqsubseteq_{E, gl} G$. Figure 3.7 shows that non-determinism of the abstract environment is a problem. In Figure 3.8 both the assumption $A$ and the guarantee $G$ forbid $b$ to occur. This allows their respective refinements according to $\sqsubseteq_{E}$, $E$ and $K$, to offer $b$ — since they can rely on $G$ respectively $A$ to forbid its actual occurrence. However obviously, the composition of the implementations $gl\{E, K\}$ now allows $b$.

Because circular reasoning is not sound for all refinements under context, it may be useful to use a stronger (more restrictive) definition of refinement under context in order to make circular reasoning sound which is the case of the refinement under context relation proposed in the contract framework that will be provided in Chapter 4.
We first provide the sufficient condition to prove dominance when circular reasoning is sound in the case of binary dominance, we extend this condition to the more general case for a set of contracts.

**Theorem 3.3.3 (Binary dominance)** Let $C$ and $C'$ be two contracts for the same interface $P$ and the same glue $gl$, with $C = (A, gl, G)$ and $C' = (A', gl, G')$. If circular reasoning is sound, then to prove that $C$ dominates $C'$ it is sufficient to prove that:

$$G \models C' \land A' \models C^{-1}$$

**Theorem 3.3.4 (Sufficient Condition for Dominance)** If circular reasoning is sound and $\forall i, \exists gl_{E_i}, gl \circ gl_I = gl_i \circ gl_{E_i}$, then to prove that the set $\{C_i\}_{i=1..n}$ dominates $C$ w.r.t. $gl$, it is sufficient to prove that:

$$\left\{ \begin{array}{l}
\Pi_P(gl_I(G_1, \ldots, G_n)) \models C \\
\forall i, \Pi_{\mathcal{P}_A_i}(gl_{E_i}(A, G_1, \ldots, G_{i-1}, G_{i+1}, \ldots, G_n)) \models C_i^{-1} \end{array} \right.$$  

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

1. $\forall i, \exists gl_{E_i}, gl \circ gl_I = gl_i \circ gl_{E_i}$
2. $gl_I(G_1, \ldots, G_n) \subseteq A, gl G$
3. $\forall i, gl_{E_i}(A, G_1, \ldots, G_{i-1}, G_{i+1}, \ldots, G_n) \subseteq G_i, gl_i A_i$
3.3. PROVING DOMINANCE

4. \( \forall i, K_i \sqsubseteq_{A_i, gl_i} G_i \)

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

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

with \( K^J = \{ K_j \}_{j \in J}, G^K = \{ G_k \}_{k \in K} \) and with \( E_i^{J,K} = K^J \cup (G^K \setminus \{ G_i \}) \).

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

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

\[
\begin{cases}
  K_q \sqsubseteq_{A_q, gl_q} G_q \text{ from (4)} \\
  gl_{E_q} \{ A, E_q^{J,K} \} \sqsubseteq G_q, gl_q A_q
\end{cases}
\]

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

\( K_q \sqsubseteq gl_{E_q} \{ A, E_q^{J,K} \}, gl_q A_q \)

As refinement under context is preserved by composition, we obtain :

\( gl_I \{ K_q, E_q^{J,K} \} \sqsubseteq_{A, gl} gl_I \{ G_q, E_q^{J,K} \} \)

This is equivalent to \( gl_I \{ K^J' \cup G^{K'} \} \sqsubseteq_{A, gl} gl_I \{ K^J \cup G^K \} \).

Finally, by using the recurrence hypothesis: \( gl_I \{ K^J' \cup G^{K'} \} \sqsubseteq_{A, gl} G \).

**Step 2** We now have to prove that:

\( \forall i \in K', gl_{E_i} \{ A, E_i^{J',K'} \} \sqsubseteq_{G_i, gl_i} A_i \)

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

\( K_q \sqsubseteq gl_{E_q} \{ A, E_q^{J,K} \}, gl q G_q \)

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

\( gl_{E_i} \{ K_q, A, E_i^{J,K}\setminus\{i\} \} \sqsubseteq G_i, gl_i, gl_{E_i} \{ G_q, A, E_q^{J,K}\setminus\{i\} \} \)

This boils down to \( gl_{E_i} \{ A, E_i^{J',K'} \} \sqsubseteq_{G_i, gl_i} gl_{E_i} \{ A, E_i^{J,K} \} \).

Hence, using the recurrence hypothesis: \( gl_{E_i} \{ A, E_i^{J',K'} \} \sqsubseteq_{G_i, gl_i} A_i \).
CHAPTER 3. CONTRACT-BASED VERIFICATION APPROACH

Conclusion  By applying our property to $I = \ldots n$, we get:

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

Theorem 3.3.4 shows that the proof of a dominance relation boils down to a set of refinement checks, one for proving refinement between the guarantees, the second for discharging individual assumptions. This result is particularly useful as it allows to check dominance without having to compose neither implementations nor contracts.

The proof of Theorem 3.3.4 relies, in addition to the circular reasoning, on a second property of the relation of refinement, that is compositionality, and which means preservation of refinement by composition. Compositionality is usually based on the rule that: if an implementation $I$ conforms to its specification $S$, then whenever composed with any component $E$ it still conforms to $S$. However, for refinement in a given context we cannot compose with any component as the context is already fixed by the relation. Thus we suppose a composite context and we integrate a part of this context in $I$ and $S$ to obtain refinement in the remaining part of the composite context. This property is in general easily satisfied by refinement relations in contract frameworks, where its always relevant to choose a refinement relation which can be preserved by composition.

**Definition 3.3.5 (Compositionality)** A set of refinement under context relations $\{ \subseteq_\omega \}_{\omega \in \Omega}$ is said to be preserved by composition if and only if the following rule applies whenever all terms are defined:

$$I \subseteq_{E, gl} S \quad E = gl_I \{ E_1, E_2 \} \quad gl \circ gl_E = gl_2 \circ gl_1$$

In the next chapter, we propose a component framework as well as a contract framework, allowing to apply the proposed methodology, with a refinement relation ensuring circular reasoning and compositionality.
3.3. PROVING DOMINANCE
Chapter 4

A Contract Framework for Components with Data

In this Chapter, we first define a component framework. That is a notion of component enriched with variables, and a set of glue operators describing how to compose these components so as to obtain systems of components. We choose to use and extend the set of composition operators of the BIP interaction model [Sif05, BS07a, BJS09b], defined in Section 2.1.2. In particular, we define a notion of observation connectors useful for our verification purposes. Second, we propose, based on this component framework, a contract framework, that is, we define conformance and a notion of refinement under context and show that they have the required properties. In particular, that the relation of refinement under context allows the property of circular reasoning. To allow to encompass progress, our definition of components is extended with a notion of progress conditions.

In this chapter, we detail in Section 4.1 our definition of components and its semantics. Then we define in Section 4.2 how to compose components using our rich interaction model and how hierarchical components are obtained by the composition of these glues. In Section 4.3, we focus on progress description and how we deal with their composition. Finally, in Section 4.4, we provide relations of the proposed contract framework and the proofs of the conditions it has to ensure.

4.1 Components with data

Let us first recall the ingredients required to define a component framework as described in Definition 2.2.1, that is, a notion of atomic component, representing abstract behavior or properties of the components we want to design, an equivalence relation ∼ on components and a set of glues GL defining composition operators which can be composed by an associative operation ◦.

In this section, we first focus on the two first ingredients, that is the definition of components and the equivalence relation between them.

We first provide the syntactic definition of components (see Definition 4.1.1), then their semantics. In the variant of BIP component framework described in Section 2.1.2, components are defined as extended labeled transition system (ELTS). Here, we also define atomic components as extended...
labeled transition systems but we yield a more abstract notion of components that we want to use for the expression of properties, not only implementations.

- We consider control states with an attribute \( \text{Inv} \), a predicate on the set of state variables \( X^{st} \) of the component. This predicate is guaranteed to hold in that control state. The variables which are not state variables are transient \( X^{tr} \). Their values are calculated twice during the execution of a step of the system that involves this component: first their values are assigned depending on \( X^{st} \) (we use a relation \( R \) for expressing their possible values), then they may be assigned again by the interaction between the involved components (such a function will be defined by the glues) and then be used in the data transformations operated by individual components (which we represent by predicates \( F_u \) on \( X = X^{st} \cup X^{tr} \)). Also data transformations of state variables (which may depend on the values of transient variables) are not given as a function but more abstractly by a predicate.

- We explicitly distinguish internal actions which we denote by \( \tau \). As usually, they cannot participate in interactions with other components and they are used to define weaker notions of equivalence and refinement. Modeling internal actions is very useful when we need to model “uncontrollable” actions. This is in general the case when a component has some local actions for which it does not need to synchronize or to exchange data with other components (see Example 4.1.2).

As in BIP, we may consider that the transient variables are associated to ports through which they are exported and imported and in that case we may write \( p[x_1, \ldots, x_n] \) to express that \( x_1, \ldots, x_n \) are transient variables only meaningful in steps involving an interaction on \( p \). Without loss of generality, we suppose in the following that a port is associated with exactly one variable. We suppose given a set of predicates that is closed by \( \land \) and \( \lor \).

**Definition 4.1.1 (Component)** A component is an ELTS defined by a tuple \((TS, X, \text{Inv}, g, F_u, \text{Prog})\) and an interface defined by a set of ports \( \mathcal{P} \):

- \( TS = (Q, q^0, \mathcal{P} \cup \{\tau\}, \rightarrow) \) is a labeled transition system: \( Q \) is a set of control states, \( q^0 \in Q \) is the initial state, \( \mathcal{P} \cup \{\tau\} \) is a set of labels. \( \rightarrow \subseteq Q \times \mathcal{P} \cup \{\tau\} \times Q \) is a set of control transitions. Elements of \( \mathcal{P} \) are ports and \( \tau \) labels internal transitions;

- \( X = X^{st} \cup X^{tr} \) is a set of variables. \( X^{st} \) contains state variables and \( X^{tr} \) contains transient variables. A boolean expression \( R \) on \( X \) defines the possible values of transient variables before the execution of a step depending on the current values of the state variables in the current state;

- \( \text{Inv} \) associates with every \( q \in Q \) a state invariant \( \text{Inv}_q \) that is a predicate on \( X^{st} \);

- \( g \) associates with every transition \( t \) a guard \( g_t \), i.e. a predicate on \( X^{st} \);
• Fu associates with every transition \( t \) an action \( Fu_t \) defined as a predicate on \( X^{st} \cup x_\gamma \cup X^{st}_{next} \), where as usually, we introduce the auxiliary variables \( X_{next} \) to represent the values of the state variables after the transition and the original variables \( X^{st} \) for representing the values before the transition and which are also used in the guard. \( x_\gamma \) is the transient variable associated with the port labeling \( t \);

• \( \text{Prog} \) a set of progress conditions (more details are given in Section 4.3).

If a given transition \( t \) is not labeled by a port but by \( \tau \), then, the function associated is a predicate \( Fu_t \) defined on variables \( X^{st} \cup X^{st}_{next} \). Indeed, when no interaction with the environment is performed, there is no need to transient variables which allow to import or export values, and thus the predicate \( Fu_t \) is completely local and it only depends on state variables.

Note, however, that we do not require that a port must have an associated variable. Indeed, a synchronization of a component with the environment may do not involve data transfer. Moreover, in the next section we propose a variant of connectors connecting ports without variables, which we use only to detect that some action is performed. These connectors are called observation connectors.

Invariants associated with states, represents the knowledge that the component has on its state variables in this state.

**Example 4.1.2** Figure 4.1 shows a component \( K \) with an interface consisting of \( \mathcal{P} = \{\text{in}, \text{out}\} \). The ports \( \text{in} \) and \( \text{out} \) label the transitions of \( K \), which has also an internal transition labeled by \( \tau \) and which corresponds to an internal computation performed by \( K \) and which does not synchronize with the environment to be performed. The invariants of the two states of \( K \) are defined on the state variable \( y \). Similarly, guards on transitions are also defined on state variables \( y_1 \) and \( y_2 \). However to exchange data with its environment, \( K \) uses the transient variables \( x_1 \) and \( x_2 \) associated to its ports. In fact, the transfer of data is performed within the functions predicates associated to the transitions labeled by these ports. The overall behavior of \( K \) is getting a value from the environment (other components), making some local computations using these values, then giving the resulting new values back to the environment through the port \( \text{out} \).

Our syntactic definition of components provides a description of some progress conditions which we denote \( \text{Prog} \). For the sake of clarity, and as the description of these properties depends of the semantics of our definition of components, we propose next to give the semantics of a component, then to detail our notion of progress in Section 4.3.

### 4.1.1 Semantics

We define the semantics of components, that is extended labeled transition system (ELTS) \( K \), as usually as a simple labeled transition system (LTS), where states are control states of \( K \) extended by valuations of state variables of \( K \) and transitions are labeled by ports extended by valuations of transient variables.

Moreover, we also have variables associated with ports and which can be interpreted differently depending on the semantics we are interested in.
Definition 4.1.3 (Safety Semantics of atomic component) The semantics of $K = (TS, X, Inv, g, Fu, Prog)$, where $TS = (Q, q^0, \mathcal{P} \cup \{\tau\}, \xrightarrow{} )$, is a labeled transition system $((Q \times V_{X^{st}}), (q^0 \times v_{X^{st}}), \mathcal{P} \cup \{\tau\}, \xrightarrow{} )$ such that:

\begin{itemize}
  \item $(Q \times V_{X^{st}})$ is a set of states, where $V_{X^{st}}$ denotes the set of valuations of state variables $X^{st}$;
  \item $\xrightarrow{}$ is the set including transitions $(q, v_{x^{st}}) \xrightarrow{l} (q', v_{x^{st}_{next}})$ where:
    \begin{itemize}
      \item $l \in ((\mathcal{P} \cup \{\tau\}) \times V_{X^{tr}})$;
      \item $\exists v_{x^{tr}} \in V_{X^{tr}}$ such that $Inv_q(v_{x^{st}}) \wedge g_t(v_{x^{st}}) \wedge R(v_{x^{st}}, v_{x^{tr}}, x_{next}^{st}) \wedge Fu_t(v_{x^{st}}, v_{x^{tr}}, x_{next}^{st})$ holds for some $t = (q, l, q') \in \xrightarrow{}$.
    \end{itemize}
\end{itemize}

We denote by $Sem_K$, the LTS defining the semantics of $K$.

In this thesis, we opt for a safety semantics.

We suppose that $K$ needs not to synchronize on any of its ports which means that all states in which a transition on port $p$ is enabled in $K$ have indeed this transition. On the other hand, we consider value of the port variables, which will be determined by the downward function of connectors when $K$ is composed, as unknown that is any possible value. In other words, the set of transitions we obtain is the maximal set of transitions that a component could fire in a given environment which means that when composed to its environment, the component could only have any set that is equal or a subset of these transitions.

Once we have defined the semantics of our description of components, we now provide the second ingredient to the definition of the component framework that is the equivalence relation between components.

Definition 4.1.4 (Equivalence relation) Equivalence on components is bisimulation. That is, two components $K_1$ and $K_2$ are equivalent, denoted $K_1 \cong K_2$, whenever their associated semantic transition systems are bisimilar.
The relation of equivalence that we concretely use in the application we propose in the next chapter, is stronger than such equivalence of semantics. We use a syntactic equivalence, that is the equality between the extended labeled transition systems possibly after renaming of control states and variables and after proving equivalence of expressions corresponding to predicates.

Some transitions of a component description may be unfeasible. Usually, detecting unfeasible transitions requires taking into account the dynamics of the system, at least when we suppose that no guard is trivially equivalent to false. Here we have declared invariants which are enforced, that is, for a given transition $t = (q \xrightarrow{p} q')$ whenever it is not possible to satisfy jointly the invariant of $q$, the guard of $t$ the transition predicate of $t$ and the invariant in $q'$, the transition is infeasible and thus called inconsistent. This notion of consistency is particularly relevant in the case of composite components where their syntactic composition may lead to several inconsistent transitions.

**Definition 4.1.5 (Inconsistent transition)** Let $K = (TS, X, Inv, g, Fu, Prog)$ be a component where $TS = (Q, q^0, P \cup \{\tau\}, \rightarrow)$. A transition $t = (q \xrightarrow{p} q') \in \rightarrow, p \in P \cup \{\tau\}$ is said to be inconsistent iff the predicate: $\text{Inv}_q \land g_t \land Fu_t \land \text{Inv}^{next}_{q'}$ is not Satisfiable, where $\text{Inv}^{next}_{q'}$ is applied on the next new version of variables. Satisfiable means that it exists a concrete valuation of the variables of $K$ for which this predicate holds.

We call a component consistent if it has no inconsistent transitions. Note that any component could be made consistent by removing all its inconsistent transitions.

**Example 4.1.6** Figure 4.2 gives a small example of how one can simplify the definition of component according to the proposed condition of consistency. If we suppose that all the component variables are natural numbers, then then transition labeled by the port $p_1$ is not consistent with the invariant of its target state $S_1$. Indeed, for any possible value of $x_1$, the corresponding function predicate $Fu_1$
implies $y > 0$ which is inconsistent with the invariant of $S_1$ namely $y = 0$. Similarly this is the case of the transition labeled by the port $p_4$.

4.2 Glues: Rich interaction model

As mentioned previously, the purpose of this chapter is the definition of a contract framework, and the first ingredient of its definition is a component framework. As in the previous section we have defined atomic components and a notion of equivalence. Now we focus in this section, on the two last ingredients, namely, the set of glues $GL$ and their composition operators $\circ$.

Like in BIP we define composition by means of a set of connectors defining an interaction-model used as a parameter of the composition operator. Indeed, we define composition by means of a set of specific connectors, namely rendez-vous, extended with data, and abstract observation connectors. We have made this choice, because the notion of component we have proposed handles variables, and thus what we need is an interaction model rich enough to handle data exchange between components. In this section, we first describe the variant of BIP glues we are using. Second, we describe how we use these glues to compose components and thus to build composite components from atomic ones. Then, we defined the composition of interaction models using the operator $\circ$ allowing to flatten hierarchical components.

4.2.1 Connectors

Interaction models are defined by a set of connectors. Here, we propose a variant of BIP connectors that have mainly two differences w.r.t the previously defined BIP connectors:

- First, we limit connectors to only two types of synchronizations and The first one is the usual BIP rendez-vous connector requiring all ports to be activated in order for the interaction to take place. The second type of connectors called observation connector connects only trigger ports and an interaction of this connector can take place as soon as any port is activated.

- The second difference concerns the upward and downward update functions of connectors. BIP connectors have concrete update functions which manipulate valuations of variables, however, in our connectors, these functions are simply predicates on these variables. We choose this abstract representation of connectors, because it allows like for components to provide abstract descriptions that can be instantiated in different ways. For us this is in particular important as in systems obtained as instances of a grammar, the actual concrete connectors depend on the actual instance whereas proofs are made at the level of the grammar using abstract connectors.

We now provide the syntax of rendez-vous and observation connectors.

**Definition 4.2.1 (Rendez-vous connector)** A rendez-vous connector is of the form $\gamma = (p_\gamma[x], P_\gamma, \delta_\gamma)$ where:
• $p_\gamma$, the exported port and $P_\gamma = \{p_1, \ldots, p_k\}$, the support set of ports, where $x$ is the transient variable associated with $p_\gamma$ and we denote by $\{x_1, \ldots, x_k\}$ the transient variables associated with the ports of the support set;

• $\delta_\gamma = (G, \mathcal{U}, \mathcal{D})$ where:
  
  – $G$ is the guard, that is, a predicate on $\{x_1, \ldots, x_k\}$
  
  – $\mathcal{U}$ is the upward update function defined from $\{x_1, \ldots, x_k\}$ into $x$, and we choose to represent it as a predicate on $\{x_1, \ldots, x_k\} \cup \{x\}$
  
  – $\mathcal{D}$ is a set $\{D_{x_i}\}$ downward update functions for each $x_i \in \{x_1, \ldots, x_k\}$ allowing to define $x_i$ using $x$, and which we define as a predicate on $\{x\} \cup \{x_i\}$

The definition of observation connectors does not involve data transformations, they have neither guard nor $\mathcal{U}$ nor $\mathcal{D}$ predicates.

**Definition 4.2.2 (Observation connector)** An observation connector $\gamma = (p_\gamma, P_\gamma)$ is defined by an exported port $p_\gamma$ and a support set $P_\gamma = \{p_1, \ldots, p_k\}$.

Note that this type of connectors allow to detect when a given transition is fired, which is particularly useful if one wants to prove some properties related to the occurrence of a given transition. This is relevant when dealing with safety and progress properties where to prove that a property is valid one has to detect that an interaction occurs.

To avoid cyclic connectors, we require also that $p_\gamma \notin P_\gamma$.

**Definition 4.2.3 (Disjoint connectors)** Given two connectors $\gamma_1 = (p_1, P_1, \delta_1)$ and $\gamma_2 = (p_2, P_2, \delta_2)$, $\gamma_1$ and $\gamma_2$ are said to be disjoint if $p_1 \neq p_2$, $p_1 \notin P_2$ and $p_2 \notin P_1$. Note that $P_1$ and $P_2$ may have ports in common, as a port may be connected to several connectors.

Note that each connector defines a set of interactions.

**Definition 4.2.4 (Interactions of connectors)** A rendez-vous connector $\gamma_{rde} = (p_\gamma[x], P_\gamma, \delta_\gamma)$ defines a unique interaction $\alpha = (p_\gamma, P_\gamma, \delta_\alpha)$ defined by the synchronization of all ports of $P_\gamma$ with $\delta_\alpha = \delta_\gamma$.

An observation connector $\gamma_{obs} = (p_\gamma, P_\gamma)$ defines as interactions any non-empty subset of $P_\gamma$.

An interaction model is defined by a set of interactions which are structured using connectors. Thus connector represent a syntactic description of interaction models. As in Section 2.1.2 we define an interaction model for a set of connectors as follows:

**Definition 4.2.5 (Interaction models)** An interaction model $\mathcal{I}$ is defined by a set of disjoint connectors $\{\gamma_i = (p_i[x], P_i, \delta_i)\}_{i=1}^n$ where its support set $S_\mathcal{I} = \bigcup_{i=1}^n P_i$ and its associated interface is defined by $P_\mathcal{I} = \{p_i|\gamma_i \in \mathcal{I}\}$ consisting of the set of the exported ports of its connectors.

We denote by $\mathcal{I}_{rde}$ the set of rendez-vous connectors of $\mathcal{I}$ and $\mathcal{I}_{obs}$ the set of its observation connectors.

The interface of an interaction model allows to define the interface of the component resulting from a composition using $\mathcal{I}$. The set $X_\mathcal{I}$ denotes the set of variables associated with the ports of $P_\mathcal{I}$ and which defines the set of variables of $\mathcal{I}$. 

65
4.2. GLUES: RICH INTERACTION MODEL

4.2.2 Composition

We now define how to compose a set of components, by means of an interaction model, that is how to build composite components from atomic ones. Similarly as in the definition of the BIP component framework (Section 2.1.2).

To define the semantics of a composition, we first define a syntactic composition associating with the composite component an atomic component defined on the product of control states. Then to compute semantics of composition as the semantics of this atomic component.

The definition of composition proposed here differs from the one of BIP in that; first, our set of transitions can be labeled by a port or by \( \tau \). Second, we use two particular types of connectors that we have to precise how components synchronize and exchange data for each type.

In our definition of component (see Definition 4.1.1) we do not allow sets of ports as labels of transitions. Thus we require that connectors of the interaction model \( I \) have at most one port of the same component in their support set.

Definition 4.2.6 (Syntactic composition) Let \( K_1, \ldots, K_n \) be a set of components defined on pairwise disjoint interfaces, such that \( K_i = (TS_i, X_i, Inv_i, g_i, Fu_i, Prog_i) \) where, \( TS_i = (Q_i, q_i^0, P_i \cup \{ \tau \}_i, \rightarrow_i) \). Let \( I \) be an interaction model defined on \( \bigcup_{i=1}^n P_i \), and which defines an interface \( P_I \) with the set of variables \( X_I \). Then, the syntactic composition of \( K_1, \ldots, K_n \) using \( I \) is a component \( (TS, X, Inv, g, Fu, Prog) \) such that:

- \( TS = (Q, q^0, P_I \cup \{ \tau \}, \rightarrow) \) with \( Q = \prod_{i=1}^n Q_i \), \( q^0 = (q_1^0, \ldots, q_n^0) \) and where \( \rightarrow \) a set of transitions of the form \( t = (q, p, q') \), where:
  
  - \( q = (q_1, \ldots, q_n) \), \( q' = (q_1', \ldots, q_n') \), \( q_i \) and \( q_i' \) being control states of the \( i \)-th component.
  - \( p, p' \in P_I \) is an exported port of a connector \( \gamma \in I \), corresponding to an interaction \( \alpha \) of \( \gamma \), such that there exists a subset \( J \subseteq \{1, \ldots, n\} \) of components with transitions \( \{(q_j, p_j, q_j')\}_{j \in J} \) and \( \alpha = \{p_j\}_{j \in J} \).
  - if \( j \notin J \), \( q_j' = q_j \). That is, the control states from which there are no transitions labeled with ports in \( \alpha \), remain unchanged.
  - \( X^{st} = \bigcup_{i=1}^n X_i^{st} \), \( X^{tr} = \bigcup_{i=1}^n X_i^{tr} \) and \( X = X^{st} \cup X^{tr} \)

The relation \( R \) between variables in \( X^{st} \) and \( X^{tr} \in X_I \) is defined as:

\( x^{tr} \in X_I \), then \( x^{tr} \) is associated with an exported port \( p_\gamma \) of a rendez-vous connector \( \gamma = (p_\gamma, P_\gamma, \delta) \in I_{rdv} \). Let \( I = |P_\gamma| \). \( U_\gamma \) is a predicate on \( \{x_1, \ldots, x_1\} \cup \{x^{tr}\} \), where every \( x_i \) is associated with a port of \( P_\gamma \). Without loss of generality, we suppose each \( x_i \) is a variable of component \( K_i \).

Then \( R(x^{tr}, X^{st}) \) is defined iff:

\[
\exists x_1, \ldots, x_I. (\forall i \in [1, I]. R_i(x_i, X_i^{st})) \land U_\gamma(x_1, \ldots, x_I)
\]

- \( Inv \) associates with every state \( q \in Q \) an invariant \( Inv_q \) defined by \( Inv_q = \bigwedge_{i=1}^n Inv_{q_i} \).
– $g$ associates with every transition $t = (q, p, q')$, corresponding to an interaction $\alpha = \{p_j\}_{j \in J}$ of a connector $\gamma \in I$, a guard defined by $g_t = (\bigwedge_{j \in J} g_j \land R_t) \land G_\gamma$. Intuitively, the guard of the new obtained transition takes into account the guards of the composed transitions as well as the guard of the connector used to compose them. Note that if $\alpha$ corresponds to an interaction of an observation connector, then $g_t = \bigvee_{i=1}^I g_{t_i}$.

– $F_u$ associates with every transition $t = (q, p, q')$, corresponding to an interaction $\alpha = \{p_j\}_{j \in J}$ of a connector $\gamma \in I$, a predicate $F_{u_t} = U \land D \land \bigwedge_{i=1}^I F_{u_{t_i}}$. If $\alpha$ corresponds to an interaction of an observation connector, then $F_{u_t} = \bigvee_{i=1}^I F_{u_{t_i}}$.

• The set $Prog$ of progress conditions of the composition will be discussed separately in Section 4.3.2.

Our composition of components is rather technical but not surprising. It does not involve hiding of ports. Indeed the interaction model $I$ has to be defined on the set of all ports of the composed components.

Note that a rendez-vous connector leads to a new control transition, in the composed component, labeled by its exported port. This transition is enabled if all local transitions are enabled and the connector guard holds, that is the guard of a transition corresponding to a rendez-vous is the conjunction of all these guards.

For an observation connector, an observation transition is enabled if at least one of the corresponding local transitions is enabled, thus the corresponding guard is the disjunction of local guards. Remember that an observation connector has an interaction (a transition in the syntactic product) for each subset of local transitions which each one has its own guard. Note that there is no maximal progress for observations: even if all local transitions are enabled, any subset may go for an observation. This is why we call it an observation transition: we can observe at the higher level of hierarchy whether at least one of the local transitions has been executed but the execution of a transition in a connected component does not influence the behavior of any other component.

Note that this syntactic composition may lead to the appearance of inconsistent transitions, thus it could be interesting to apply the consistency condition, described in Section 4.1.5, to the obtained component.

**Example 4.2.7** Figure 4.3 depicts a set of two components $K^1$ and $K^2$ (each component represents an instance of the component given in Example 4.1.2 and an interaction model $I$ represented by two connectors $\gamma_1, \gamma_2$. The composite component $\mathcal{I}(K^1, K^2)$ obtained as a composition of $K^1$ and $K^2$ using $\mathcal{I}$ is given in Figure 4.4. As described in Definition 4.2.6, the new transitions of the composite component obtained as a synchronization between ports of $K^1$ and $K^2$ are now labeled by the exported ports 1to2 and 2to1 of the connectors. Guards and local functions of these new transitions are computed as given in Figure 4.4. Variables of $\mathcal{I}(K^1, K^2)$ are given by $X^{\mathcal{I}} \cup X^1 \cup X^2$. The transitions labeled by $\tau$ represent internal transitions which still possible locally but not exported to the interface of the new composite component.

Note, in this composition as well as in the definition of components, we did not discuss how we
4.2. GLUES: RICH INTERACTION MODEL

\[ G_2 : [x_1^2 + x_1^3 = 0] \]
\[ U_2 : (v_2 = x_1^2) \]
\[ D_2 : (x_1^7 = v_2) \land (x_2^7 = 0) \]

\[ G_1 : [x_2^7 + x_1^3 = 0] \]
\[ U_1 : (v_1 = x_2^7) \]
\[ D_1 : (x_1^7 = v_1) \land (x_2^7 = 0) \]

\[ x_{st} = \{y_1^7\} \]
\[ R^1 : (y_1^7 = x_1^7) \land (y_1^7 = x_2^7) \]

\[ \gamma_1(G_1, U_1, D_1) \]
\[ \gamma_2(G_2, U_2, D_2) \]

Figure 4.3: Composition of components.

\[ I(K^1, K^2) \]
\[ \tau \]
\[ g_1^3 : [y_1^7 > 0] \]
\[ Fu_1^1 : (y_{next}^1 = y_1^7 + x_1^7) \]
\[ S_1^1 : y_1^7 > 0 \]
\[ g_1^2 : [y_1^7 = 0] \]
\[ Fu_2^1 : (y_{next}^1 = x_2^7) \]
\[ S_2^1 : y_1^7 = 0 \]

\[ g_2^3 : [y_2^7 > 0] \]
\[ Fu_2^2 : (y_{next}^2 = y_2^7 + x_2^7) \]
\[ S_2^2 : y_2^7 > 0 \]
\[ g_2^2 : [y_2^7 = 0] \]
\[ Fu_1^2 : (y_{next}^2 = x_1^7) \]
\[ S_1^2 : y_2^7 = 0 \]

\[ \tau \]
\[ g_2^1 : [y_2^7 > 0] \]

Figure 4.4: Composite component.
describe progress conditions associated with each component and how to compose them. This will be detailed in Section 4.3.

4.2.3 Composition of interaction models

As already explained, we want to be able to define components and connectors hierarchically, and we want to be able to flatten a hierarchical component. This means that we must be able to compose connectors and transform a hierarchically defined connector into a single "flat" connector. We call this "merge" of connectors.

The merge of connectors is only defined on connectors of the same type. It is the operation that takes two connectors defining together a hierarchical connector and returns a connector of a basic type (see Figure 4.5). More precisely, the merge of two rendez-vous connectors defines a new rendez-vous connector and similarly, the merge of two observation connectors defines a new observation connector. The merge for rendez-vous connectors is already defined in [BJS09b], where it is called flattening. Here we restrict this definition so as to preserve associativity of the upward and downward functions.

\[ \gamma_1 \circ \gamma_2 = \gamma_2 \circ \gamma_1 = \gamma \]

**Figure 4.5: Merge of Connectors.**

**Definition 4.2.8 (Merge of rendez-vous connectors)** Let \( \gamma_1 = (p_1[x_1], P_1, \delta_1) \) and \( \gamma_2 = (p_2[x_2], P_2, \delta_2) \) be two rendez-vous connectors such that \( P_1 \cap P_2 = \emptyset \) and \( p_1 \neq p_2 \). The merge of \( \gamma_1 \) and \( \gamma_2 \), denoted \( \gamma_1 \circ \gamma_2 \), is defined as follows:

- if \((p_1 \not\in P_2 \text{ and } p_2 \in P_1)\) then \( \gamma_1 \circ \gamma_2 = (p[x], P, \delta) \) with:
  - \( P = P_1 \cup P_2 \setminus \{p_2\} \)
  - \( \delta = (G, U, D) \) is defined as follows:
    - \( G = G_2 \land \exists v_2, G_1[v_2/x_2] \land U_2[v_2/x_2] \)
    - \( U = \exists v_2, U_2[v_2/x_2] \land U_1[v_2/x_2] \)
    - \( D_{x_k} = \{ \exists v_2, D_{1,x_k}[v_2/x_2] \land D_{2,x_k}[v_2/x_2] \text{ if } x_k \in P_1 \setminus \{x_2\} \}
    - \( D_{x_k} = \exists v_2, D_{1,x_k}[v_2/x_2] \land D_{2,x_k}[v_2/x_2] \text{ if } x_k \in P_2 \)
  - if \((p_1 \in P_2 \text{ and } p_2 \not\in P_1)\) then \( \gamma_1 \circ \gamma_2 = \gamma_2 \circ \gamma_1 \).
**Definition 4.2.9 (Merge of observation connectors)** Let $\gamma_1 = (p_1, \mathcal{P}_1)$ and $\gamma_2 = (p_2, \mathcal{P}_2)$ be two observation connectors such that $\mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$ and $p_1 \neq p_2$. The merge of $\gamma_1$ and $\gamma_2$, denoted $\gamma_1 \bullet \gamma_2$, is defined in the following situations:

- if $(p_1 \notin \mathcal{P}_2$ and $p_2 \in \mathcal{P}_1$) then $\gamma_1 \bullet \gamma_2 = (p, \mathcal{P}, \delta)$ with:
  - $p = p_1$
  - $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2 \setminus \{p_2\}$

- if $(p_1 \in \mathcal{P}_2$ and $p_2 \notin \mathcal{P}_1$) then $\gamma_1 \bullet \gamma_2 = \gamma_2 \bullet \gamma_1$.

If $(p_1 \in \mathcal{P}_2$ and $p_2 \in \mathcal{P}_1$), then $\gamma_1 \bullet \gamma_2$ is not defined because this would result in a cyclic connector. Besides, if $(p_1 \notin \mathcal{P}_2$ and $p_2 \notin \mathcal{P}_1$), then $\gamma_1$ and $\gamma_2$ are disjoint, thus they cannot be merged.

We can now define the composition of interaction models as the union of the set of disjoint connectors of each interaction model, and the connectors obtained by a merge for connectors that are not disjoint. Note that merge of connectors and thus composition of interaction models are by definition commutative and allows flattening according to the definition of merge. They are also associative because $\land$ operator on update predicates of connectors is associative.

**Definition 4.2.10 (Composition of interaction models)** The merge of connectors $\bullet$ is extended to interaction models as follows. The composition $\circ$ of two interaction models $\mathcal{I}_1$ and $\mathcal{I}_2$ is obtained from $\mathcal{I}_1 \cup \mathcal{I}_2$ by inductively composing all connectors which are not disjoint.

Examples of composition of interaction models are depicted in Figures 4.6 and 4.7.
CHAPTER 4. A CONTRACT FRAMEWORK FOR COMPONENTS WITH DATA

4.3 Progress description

In this section, we provide a description of progress conditions associated to the syntactic definition of components. Then, we give rules allowing to derive some progress conditions of a composition.

4.3.1 Progress in components

Progress conditions are useful to exclude behaviors staying forever in some particular states or loops. We adapt usual weak and strong fairness conditions [AFK88, FK84] to component by defining a set of progress conditions $\text{Prog}$. In this section, we provide first the syntax of progress conditions. Then, their semantics defined as a constraint on infinite executions of the semantics of our components, that is a labeled transition system. As usually, we consider progress conditions as an assumption which will have to be satisfied later on by an implementation.

Definition 4.3.1 (Progress Condition) Let be $K = (TS, X, \text{Inv}, g, Fu, \text{Prog})$ a component. A progress condition $pr \in \text{Prog}$ is a pair $(T_c, T_p)$, where:

- $T_c$ is either a set of transitions of $TS$ or the symbol $\top$. $T_c$ is called the condition;
- $T_p$ is a set of transitions called promise.

The set of start control states of transitions of $T_p$ are denoted $\text{start}(T_p)$ and called progress states.

We now define the semantics of a progress condition on a component $K$ as a constraint on the set of executions $\text{exec}(Sem_K)$ of the LTS defining its semantics according to the Definition 4.1.3. We recall here that according to the semantics of a component it is supposed that its ports are not connected and that the variables of these ports could take any value meaning that they are used like inputs. When a component $K$ guarantees progress by a transition $t$ in a state $q_K$, it ensures the same progress in any environment $E$ which from some point on permanently allows the transition $t$. The environment $E$ allows a transition $t$ of $K$ in a state $q_E$ if either $t$ is a transition of $K$ that does not need to synchronize with $E$ or if $E$ has a transition synchronizing with $t$ that is enabled in $q_E$.

Definition 4.3.2 (Semantics of a progress condition) Let be a component $K$ where its semantics is given by the LTS $Sem_K = (Q, q_0, P \cup \{\tau\}, \rightarrow)$ and $\sigma$ an infinite execution in $Sem_K$. Then, $\sigma$ respects the progress condition $(T_c, T_p)$ if the following holds:

- If $T_c = \top$, $\sigma$ contains infinitely many states in $\text{start}(T_p)$ implies that it also contains infinitely many transitions in $T_p$.
- If $T_c = \{t_i\}_{i=1}^k$, $t_i \in \rightarrow$, $\sigma$ contains infinitely many $t_i$ transitions implies that it also contains infinitely many transitions in $T_p$.
- If $T_c = \emptyset$, then the progress condition is trivially satisfied.
σ is called to be a legal execution of Semₖ if it respects all progress condition in Prog. We denote by \( \text{exec}(K) \) the set of legal executions of Semₖ.

Intuitively, the set \( T_c \) defines the set of prefixes for which progress has to be ensured. No prefixes satisfy \( \emptyset \), thus if \( T_c = \emptyset \) there is no progress to guarantee. On the other hand, all prefixes satisfy \( \top \), which means that if \( T_c = \top \) any execution of the component containing infinitely many a state of \( \text{start}(T_p) \) must guarantee progress. This is what we call unconditional progress.

Our notion of progress is close to the notion of strong fairness described in [AFK88, FK84, PPR, PXZ02] and where fairness is defined on states and not transitions. Here, the choice of expressing progress using transitions is highly motivated by the properties we want to verify on our application, described in Chapter 5, and where the progress is guaranteed by the occurrence of some transitions.

We now define what it means to guarantee a progress condition by a component.

**Definition 4.3.3** Let be \( K = (TS, X, \text{Inv}, g, Fu, Prog) \) and \( (T_c, T_p) \) be a pair of sets of transitions of \( TS \). \( K \) guarantees \( (T_c, T_p) \), denoted \( K \models (T_c, T_p) \), if one the following conditions holds:

- \( (T_c, T_p) \in Prog \).
- \( \forall \sigma \in \text{exec}(K), \sigma \) respects \( (T_c, T_p) \).

Intuitively, this means that a component guarantees a progress condition if it is indeed one of its progress conditions in \( Prog \) or if it is implied by its progress conditions. A few rules guaranteeing such implication between progress conditions are given below.

Note also that a progress condition \( pr = (T_c, T_p) \) of \( K \) is trivially satisfied if no \( T_c \) transition can be fired infinitely often or when \( T_c = \top \) there is no executions containing infinitely many states in \( \text{start}(T_p) \). A progress condition that cannot be satisfied by a component is called inconsistent and it is defined as follows:

**Definition 4.3.4** A progress condition \( (T_c, T_p) \) is called inconsistent with a component \( K \), if there exist finite prefixes of \( \text{exec}(\text{Sem}_K) \) that cannot be extended to legal executions.

We suppose that all progress conditions in \( Prog \) of \( K \) are consistent with \( K \). Note that a progress condition \( (T_c, T_p) \) with a set \( T_p \) which is empty or not reachable from any sequence of transitions containing \( T_c \)-transitions is inconsistent with \( K \).

We now provide some rules, that allow reasoning about progress conditions. In particular, these rules allows to deduce new progress conditions from a set of progress conditions. These rules are of particular interest, when one wants to compare components (such as to establish refinement) and thus needs to define some implication between their progress conditions. In the following, let be a component \( K = (TS, X, \text{Inv}, g, Fu, Prog) \).

**Rule 4.3.5 (monotonicity)** \( K \models (T_c, T_p) \) implies that \( K \models (T_c, T_p \cup \{t\}) \), where \( t \) is a transition of \( K \).


Proof Suppose that $K \models (T_c, T_p)$. Then according to Definition 4.3.3 this means that either $(T_c, T_p) \in \text{Prog}$ or $\forall \sigma \in \text{exec}(K)$, $\sigma$ respects $(T_c, T_p)$.

If $(T_c, T_p) \in \text{Prog}$, then $\forall \sigma \in \text{exec}(K)$, $\sigma$ contains infinitely many $T_c$-transitions implies that it also contains infinitely many transitions in $T_p$ which means also that it also contains infinitely many transitions in $T_p \cup \{t\}$, for any transition $t$ of $K$. Similarly if $\forall \sigma \in \text{exec}(K)$, $\sigma$ respects $(T_c, T_p)$ then it also respects $(T_c, T_p \cup \{t\})$. Thus we obtain that $\forall \sigma \in \text{exec}(K)$, $\sigma$ respects $(T_c, T_p \cup \{t\})$, that is $K \models (T_c, T_p \cup \{t\})$.

The Rule 4.3.5 means that the smaller the set $T_p$, the stronger the progress constraint. Indeed, when the set $T_p$ is larger, more possibilities are given to guarantee the progress condition, as it is sufficient to fire one of the $T_p$ transitions to ensure the condition. However, the opposite implication does not hold in general. It is only guaranteed when either $T_p$ or $\{t\}$ is not enabled in $K$ when it is possible to fire infinitely many transitions in $T_c$.

Rule 4.3.6 $K \models (T_c \cup T_c', T_p)$ implies that $K \models (T_c, T_p)$; that is, the larger the set $T_c$, the stronger the progress constraint.

Proof Suppose that $K \models (T_c \cup T_c', T_p)$. Then according to Definition 4.3.3 this means that either $(T_c \cup T_c', T_p) \in \text{Prog}$ or $\forall \sigma \in \text{exec}(K)$, $\sigma$ respects $(T_c \cup T_c', T_p)$.

Both cases means that $\forall \sigma \in \text{exec}(K)$, $\sigma$ contains infinitely many $T_c \cup T_c'$-transitions implies that it also contains infinitely many transitions in $T_p$. Thus if $\sigma$ contains infinitely many $T_c$ transitions it has also to contain infinitely many transitions in $T_p$, that is $\sigma$ respects $(T_c, T_p)$.

The opposite implication of Rule 4.3.6 is only guaranteed if $\text{exec}(\text{Sem}_K)$ has no executions with loops containing transitions in $T_c'$ or if $(T_c, T_p)$ is also guaranteed by $K$.
Example 4.3.7 Figure 4.8 illustrates the Rules 4.3.5 and 4.3.6 for simple examples of progress conditions. In the left of the figure, there is a representation of the LTS describing the semantics of two components. The set of transitions $T_2$ is presented by blue transitions, and $T_3$ by red ones.

Example 4.3.8 Suppose a component $K$ whose description is given by the Figure 4.1 which we suppose has no variables and thus no guards or invariants. We may define for $T$ components. The set of transitions $pr$ with variables, guards and invariants as a possible implementation of $K$. This description whenever the transition takes place, the associated function $f_1$ will update the new value of the state variable $y$ using the value of the port variable $x_1$. As $x_1 > 0$, $y$ will take a positive value ($y > 0$) and then the guard of $in$ will be set to false and the transition could not be fired again. However, the guard corresponding to the internal $\tau$ transition will be equal to true and thus performed if we suppose that this implementation guarantees progress by their enabled transitions. This forbids in transition to be fired again, before out transition takes place.

4.3.2 Progress of a composition

The methodology of Chapter 3 requires proving properties of compositions of components. More precisely, in step 3 of the methodology, the dominance of a set of contracts over a global contract requires a set of refinement proofs showing that a composition refines a property in some context. Concerning progress, this means that one has to show, that the finer, composed property guarantees at least the progress required by the looser global property, and therefore we do not need to compute the strongest but only a sufficient progress condition for products.

Thus, we only provide some rules which allow to derive some progress conditions of a composite component from progress conditions of its sub-components. The idea is based, first, on defining a projection of this property of $K$ on the set of properties corresponding to $\{K_i\}$. Then providing conditions on these projections so as to guarantee to global property of the composition.

To define such a projection, we provide first a notion of projection of a transition of $K$ into a set of local transitions corresponding each one to a transition in $K_i$. This projection, as well as the condition provided in Definition 4.3.10, are defined on the syntax of components. Thus it takes into account how this transition is obtained and it is computed w.r.t. the interaction model used to build this composite component.

In the following, we suppose $\{K_1, \ldots, K_n\}$ a set of components such that $K_i = (TS_i, X_i, Inv_i, g_i, Fu_i, Prog_i)$ where $TS_i = (Q_i, q_1^i, P_i, \rightarrow_i)$. $I$ an interaction model defined on $\bigcup_{i=1}^n P_i$ and defining an interface $P_I$. $K = I(K_1, \ldots, K_n)$ a composite component obtained as described in Definition 4.2.6 $K = (TS, X, Inv, g, Fu, Prog)$ where $TS = (Q, q_0^i, P_I, \rightarrow)$.

Definition 4.3.9 (Projection of a transition) Let $t = (q_1, \ldots, q_n) \xrightarrow{P} (q_1', \ldots, q_n')$ be a transition of $K$ ($t \in \rightarrow$) where $P$ is the exported port of a connector $\gamma$ with a support set $P_\gamma$. Then the projection of $t$ on the component $K_i$ denoted $\pi_i(t) \in \rightarrow_i$ is defined as follows:

- if $P_i \cap P_\gamma = \{p_i\}$ then, $\pi_i(t) = q_i \xrightarrow{P_i} q_i'$
CHAPTER 4. A CONTRACT FRAMEWORK FOR COMPONENTS WITH DATA

• if \( \mathcal{P}_i \cap \mathcal{P}_\gamma = \emptyset \) then, \( \pi_i(t) = \emptyset \)

The projection of a set of transitions \( T \subseteq \rightarrow \) is \( \pi_i(T) = \{ \pi_i(t) | t \in T \} \).

Control transitions and control states of the composite component \( K \) are called global, and their corresponding projections are called local.

Note here, that we do not treat the case of \( \tau \) transitions as they are not relevant in a composition, however we will discuss progress by \( \tau \)-transitions when defining refinement in Section 4.4.

We now provide sufficient conditions allowing to deduce that a given progress condition is guaranteed by a composition of components using progress condition guaranteed by these components. For the sake of clarity and w.l.o.g, in the following definition, we treat the case of \( K \) obtained as a composition of two components \( K_1, K_2 \) as it can be naturally extended to a composition of \( n \) components.

**Theorem 4.3.10 (Progress condition of a composition)** Let \( pr = (T_c, T_p) \) be a pair of sets of transitions of \( K \). Let be the set \( T_{c_i} \) defined by \( T_{c_i} \subseteq \pi_i(T_c) \) if \( \pi_i(T_c) \neq \emptyset \) and \( T_{c_i} = \top \) otherwise. If one of the following conditions holds then we have \( K \models pr = (T_c, T_p) : \)

1. If \( T_p \) is a set of transitions labeled by ports representing observation connectors, then to guarantee \( pr \) it is sufficient that \( \exists i \in [1, 2] \), such that \( K_i \models (T_{c_i}, \pi_i(T_p)) \); This means that to guarantee \( pr \) it is sufficient to guarantee a looser progress implying \( pr \), that is the progress by any transitions which do not require synchronization.

2. If \( T_p \) is of the form \( T_p^{obs} \cup T_p^{rdv} \) where \( T_p^{obs} \) a set of transitions labeled by ports of observation connectors and \( T_p^{rdv} \) a set of transitions for rendez-vous connectors. Then to guarantee \( pr \), it is sufficient to guarantee either the progress condition \( (T_c, T_p^{obs}) \) as described above, or one of the following conditions:

   • It exists a set \( T \subseteq T_p^{rdv} \) of transitions labeled by the same port \( p \) and at least one of two components, let it be \( K_1 \), guarantees \( (T_{c_1}, \pi_1(T)) \) and the other, that is \( K_2 \), guarantees that in any situation compatible with \( T_{c_2} \) some transitions in \( \pi_2(T) \) will eventually be enabled forever unless no \( T_{c_i} \)-transitions are infinitely taken or one of the transitions in \( T \) is taken. A situation is compatible with \( T_{c_2} \) if it exists executions containing infinitely many \( T_{c_2} \)-transitions. Note that in the case of \( n \) components, the second condition has to be ensured by \( (n-1) \) components.

   • It exists a set \( T \subseteq T_p^{rdv} \) of transitions labeled by the same port \( p \) and in any state \( q = (q_1, q_2) \in Q \) a global state of \( K \) in which a infinite sequence of transitions containing \( T_{c_i} \)-transitions is possible and for at least one of the two components \( (K_1) \) we have \( q_1 \in \text{start}(\pi_1(T)) \) and for the other \( (K_2) \) we have \( q_2 \notin \text{start}(\pi_2(T)) \) then \( K_2 \) must also guarantee that a transition labeled by \( p \) will eventually be enabled in \( q_2 \). This guarantees that whenever a component guarantees a transition for synchronization in a given global state, the rest of components involved in the synchronization will offer this transition in the same global state.

   In both cases \( K \) satisfies the much stronger progress condition \( (T_c, T) \).
Note that we can weaken these conditions to guarantee \((T_c, T_p^{\text{obs}} \cup T)\) by requiring that \(K_1\) guarantees \((\pi_1(T_c), \pi_1(T_p^{\text{obs}}) \cup \pi_1(T))\) and that \(K_2\) guarantees \((\pi_2(T_c), \pi_2(T_p^{\text{obs}}))\) or the additional condition required to guarantee \(T\).

Note that the additional condition that \(K_2\) has to guarantee in case 2 is semantic and a syntactic approximation of this condition is possible using the syntactic description of components.

This definition distinguishes two cases related to the type of connectors used to compose components. When the promise of the progress condition involves transitions of observation connectors, then to guarantee this progress, it is sufficient that at least one component guarantees this progress locally. However, as shown in Definition 4.3.10, it is much complicated when the promise of the progress condition involves transitions corresponding to strong synchronizations between the \(\{K_i\}\).

Note that these conditions are sufficient means that if they are not satisfied by the set of components \(\{K_i\}\), they do not help us to decide whether \(pr\) is indeed a progress condition of \(K\) or not. New weaker conditions could also be defined, however in the application described in Chapter 5, the conditions provided here are sufficient to deduce progress of the built compositions.

Example 4.3.11 Let be the composite component \(I(K^1, K^2)\) described in the Example 4.2.7 built as a composition of components \(K^1\) and \(K^2\) (see Figure 4.3). We have already explained in Example 4.3.8 how \(K^1 \models (\{S^1_1 \xrightarrow{\text{out}^1} S^1_1\}, \{S^1_2 \xrightarrow{\text{in}^1} S^1_1\})\), as in addition \(K^2\) guarantees that the transition \(S^2_2 \xrightarrow{\text{out}^2} S^2_2\) is always enabled in \(S^2_2\), then we can deduce, according to the second condition of Definition 4.3.10, that \(I(K^1, K^2)\) guarantees the progress condition \((\{S^1_1 S^2_1 \xrightarrow{1\text{to}2} S^1_1 S^2_1\}, \{S^1_2 S^2_2 \xrightarrow{2\text{to}1} S^1_1 S^2_1\})\).

4.4 Relations of the contract framework

In this Section, we provide the remaining ingredients to the definition of the contract framework. For this purpose, let us first recall that a contract framework is defined by (1) a component framework, (2) a conformance relation \(\preceq\) that is transitive and (3) a refinement under context relation \(\sqsubseteq_{(E, I)}\) parameterized by a context and implying conformance.

In the previous sections we have provided a component framework enriched with variables and data transfer. We now provide a description of conformance and refinement under context relations.

Usually, conformance relation is given first and refinement is a preorder implying conformance, where very often it is chosen to be the weakest such relation that is compositional. In order to be able to use the property of circular reasoning, we may need stronger refinement relations. Here we inverse the presentation for presentational reasons only. In fact, it turned out easier to define conformance in terms of refinement under context rather than the other way round.

4.4.1 Refinement

We provide here, the relation of refinement under context we have chosen for the already described component framework. Remember that refinement relates two components
with respect to a given context. In this section, a context is denoted \((E, I)\), where \(E = (TS_E, X_E, Inv_E, g_E, Fu_E, Prog_E)\) with \(TS_E = (Q_E, q_0^E, P_E, \rightarrow_E)\) and \(I\) is an interaction model given by a set of connectors defined on the set of ports \(S_I\). \(I\) defines an interface \(P_I\) that is the exported ports of its connectors.

Let be two components \(K_{abs}\) and \(K_{conc}\) where \(K_{abs}\) is called the abstract component and \(K_{conc}\) the concrete component and we want to define a refinement of \(K_{abs}\) by \(K_{conc}\) in the context \((E, I)\).

\(K_{abs}\) and \(K_{conc}\) are given as follows: \(K_{abs} = (TS_{abs}, X_{abs}, Inv_{abs}, g_{abs}, Fu_{abs}, Prog_{abs})\) and \(K_{conc} = (TS_{conc}, X_{conc}, Inv_{conc}, g_{conc}, Fu_{conc}, Prog_{conc})\) where \(TS_{abs} = (Q_{abs}, q_0^0, P, \rightarrow_a)\) and \(TS_{conc} = (Q_{conc}, q_0^0, P \cup \{\tau\}, \rightarrow_c)\).

Note that the interaction model \(I\) connects \(K_{abs}\) and \(K_{conc}\) to their environment \(E\), thus it is defined on \(S_I = P \cup P_E\) and it contains structured connectors of the form \(\gamma = (p_\gamma, P_\gamma)\) where \(\text{card}(P_\gamma) \leq 2\). Indeed, as connectors do not connect more than one port of the same component, \(P_\gamma\) contains either two ports one of \(P_E\) and the other of \(P\) or only one port that is of \(P_E\) or of \(P\).

We denote also by \(\text{Sem}_{I(K_i, E)}\) the LTS defining the semantics of the ELTS given by \(I(K_i, E)\) according to the Definition \[4.1.3\] and by \(\text{Sem}_{I(K, E)}^\text{Tr}\) the set of its transitions.

For the sake of clarity of our definition of refinement, we suppose the following:

1. \(K_{abs}\) has no internal (\(\tau\)) transitions;
2. \(E\) has no no internal (\(\tau\)) transitions;

These assumptions imply no loss of generality as internal transitions do not involve any interaction with the environment.

The refinement we propose is defined by the means of two relations:

- \(\alpha\) is a boolean expression on variables of \(K_{conc}\) and \(K_{abs}\) representing a relation between abstract and concrete valuations.
- \(\mathcal{R}\) relates the control states of \(K_{conc}\) and its environment \(E\) to the control states of \(K_{abs}\) and thus defines which concrete states refine the abstract ones.

It is a well-known fact that any property expressed by a Buchi automaton can be written as an intersection of a safety and a progress property \[AS87\], and this holds also for our properties. Thus our definition of refinement consists of two parts. The safety part allows to describe that any transition that is offered in \(K_{conc}\) is also allowed by \(K_{abs}\). This guarantees that in the concrete component \(K_{conc}\), we do not have executions that are non permitted by the abstract component \(K_{abs}\). Note that to preserve safety, we have to take into account the context. Indeed, a given transition of \(K_{conc}\) that is structurally forbidden by the context, does not have to be allowed by \(K_{abs}\). Safety preservation is defined by the relation \(\mathcal{R}\) defined on control states of components.

The progress part of our refinement relation takes into account the set of progress properties of components \(K_{conc}\) and \(K_{abs}\). To preserve progress by refinement, we propose to define some particular relation of projection with respect to the relation \(\mathcal{R}\) of the refinement relation (see Definition \[4.4.1\]). This projection allows to infer from the set of progress properties of \(K_{abs}\), a new set of progress properties that \(K_{conc}\) has to provide. More precisely, we use the relation \(\mathcal{R}\) obtained by
the safety preservation of the refinement relation, to compute the inverse image, by $\mathcal{R}$, of transitions $K_{\text{abs}}$ that is a set of transitions of $K_{\text{conc}}$. Thus we provide now this definition of the projection of a transition with respect to a given relation between control states of components $K_{\text{conc}}, K_{\text{abs}}$, and $E$.

**Definition 4.4.1 (Projection of a relation)** Let $\mathcal{R}$ be a relation on $(Q_{\text{conc}} \times Q_{\text{abs}})$. We say $\overline{\mathcal{R}} \subseteq Q_{\text{conc}} \times Q_{\text{abs}}$ is the projection of $\mathcal{R}$ and $q_a \overline{\mathcal{R}} q_a$ if $\exists q_a \times q_a'$ such that $q_a \in Q_{\text{conc}}$ and $q_a' \in Q_{\text{abs}}$.

For $Q_a \subseteq Q_{\text{abs}}$, let be $\overline{\mathcal{R}}^{-1}(Q_a) = \{q_c \in Q_{\text{conc}} \mid \exists q_a \in Q_a \times q_c \overline{\mathcal{R}} q_a\}$ the inverse image of $Q_a$ under $\overline{\mathcal{R}}$. Thus we have $\overline{\mathcal{R}}^{-1}(Q_a) \subseteq Q_{\text{conc}}$.

Similarly, we denote by $\overline{\mathcal{R}}^{-1}$ the inverse image of a transition $t \in p$ defined as follows: $\overline{\mathcal{R}}^{-1}(\{q_a \xrightarrow{p} q_a'\}) = \{(q_c \xrightarrow{p} q_c') \in \overline{\mathcal{R}}^{-1}(\{q_a\}) \wedge q_c' \in \overline{\mathcal{R}}^{-1}(\{q_a'\})\}$.

**Definition 4.4.2 (Refinement under context)** $K_{\text{conc}}$ refines $K_{\text{abs}}$ in the context of $(E, I)$, denoted $K_{\text{conc}} \sqsubseteq_{E, I} K_{\text{abs}}$, iff:

- **Safety part:** $\exists \alpha$ a relation in $X_{\text{conc}} \cup X_{\text{abs}}$ and $\forall \mathcal{R} \subseteq (Q_{\text{conc}} \times Q_{\text{abs}})$ and $s.t. (q_c^0, q_E^0) \mathcal{R} q_a^0$ implies:
  1. $\text{Inv}_{q_c} \wedge \alpha(X_{\text{conc}}, X_{\text{abs}}) \Rightarrow \text{Inv}_{q_a}$.
  2. $\forall p \in P, \exists \gamma = (p_\gamma, p_E) \in I$ such that $p \in P_\gamma$, then the following holds:
     - $t_c = (q_c, q_E) \xrightarrow{p} (q_c', q_E') \in \text{Sem}_{(K_{\text{conc}}, (E, I))}^{Tr}$ implies $\exists q_a' such that $t_a = (q_a, q_E) \xrightarrow{p_\gamma} (q_a', q_E' \overline{\mathcal{R}} q_a') \in \text{Sem}_{(K_{\text{abs}}, (E, I))}^{Tr}$. If $t_c$ is independent of the context, i.e., $\gamma = \emptyset$, then $q'_E = q_E$.
  3. $q_c \xrightarrow{\tau} q_c' \overline{\mathcal{R}} q_a$: states related by $\tau$-transitions refine the same state.

- **Progress part:**

$$\forall (T_c, T_p) \in \text{Prog}_{\text{abs}}, K_{\text{conc}} \models (\overline{\mathcal{R}}^{-1}(T_c), \overline{\mathcal{R}}^{-1}(T_p)).$$

Note that if $(\overline{\mathcal{R}}^{-1}(T_c) \neq \emptyset)$ then the progress condition $(\overline{\mathcal{R}}^{-1}(T_c), \overline{\mathcal{R}}^{-1}(T_p))$ is trivially satisfied by $K_{\text{conc}}$. However, if $(\overline{\mathcal{R}}^{-1}(T_c) \neq \emptyset)$ then the progress condition $(\overline{\mathcal{R}}^{-1}(T_c), \overline{\mathcal{R}}^{-1}(T_p))$ is inconsistent with $K_{\text{conc}}$ and thus that $K_{\text{conc}}$ does not guarantee this progress.

Our relation of refinement preserves safety as it is based on the usual notion of simulation and it also preserves progress from the abstract component to the concrete one. Note that the progress part of our definition of refinement takes into account the context, as the progress conditions that $K_{\text{conc}}$ has to satisfy are computed using the inverse image of $\mathcal{R}$ that is a relation involving the context. Note that for a given transition $t$ in $T_p$, the projection of its abstract start state by $\overline{\mathcal{R}}^{-1}$ may correspond to a set of concrete states due to the presence of $\tau$ transitions in $K_{\text{conc}}$. Thus proving the progress by $t$ is then transformed into proving the progress between the states refining $\text{start}(t)$ to a state guaranteeing...
CHAPTER 4. A CONTRACT FRAMEWORK FOR COMPONENTS WITH DATA

t. This is detailed in the following rule where we suppose that \((T_c, T_p)\) a pair of sets of transitions of \(K_{abs}\), where \(T_p = \{ t_p = q \xrightarrow{p} q' \}\), \(K_{abs} \models (T_c, t_p)\) and \(K_{conc} \sqsubseteq_{E, I} K_{abs}\) according to the Definition 4.4.2 of refinement. We also suppose that \(\overline{R}_T^{-1}(T_c) \neq \emptyset\) and \(\overline{R}_T^{-1}(t_p) \neq \emptyset\).

Rule 4.4.3 (Progress with \(\tau\)-transitions) Let \(\overline{R}_T^{-1}(q) = Q_c\) be the set of states of \(K_{conc}\) refining \(q\) the start state of \(t_p\). Then to prove that \(K_{conc} \models (\overline{R}_T^{-1}(T_c), \overline{R}_T^{-1}(t_p))\) it is sufficient to prove the following:

\[ \forall q_c \in Q_c \text{ either } K_{conc} \models (\overline{R}_T^{-1}(T_c), q_c \xrightarrow{p} \text{ or all finite sequences of } \tau \text{ transitions } t_1 t_2, \ldots, t_n \text{ starting from } q_c, \text{ relating states in } Q_c, \text{ lead to a state } q_{conc} \in Q_c \text{ such that } K_{conc} \models (\overline{R}_T^{-1}(T_c), q_{conc} \xrightarrow{p}) \text{ and } \forall i \in [1, n], K_{conc} \models (\overline{R}_T^{-1}(T_c), t_i). \]

Proof Suppose that \(\forall q_c \in Q_c \text{ either } K_{conc} \models (\overline{R}_T^{-1}(T_c), q_c \xrightarrow{p}) \text{ or all finite sequences of } \tau\text{-transitions } t_1 t_2, \ldots, t_n \text{ starting from } q_c, \text{ relating states in } Q_c, \text{ lead to a state } q_{conc} \in Q_c \text{ such that } K_{conc} \models (\overline{R}_T^{-1}(T_c), q_{conc} \xrightarrow{p}) \text{ and } \forall i \in [1, n], K_{conc} \models (\overline{R}_T^{-1}(T_c), t_i). \) What we have to prove is that it does not exist an execution \(\sigma\) of \(\text{exec}(\text{Sem}_{K_{conc}})\) such that \(\sigma\) contains infinitely many transitions in \(\overline{R}_T^{-1}(T_c)\) and visiting infinitely often states in \(Q_c\) without guaranteeing a transition labeled by \(p\). Suppose that such execution exists, then visiting infinitely often states in \(Q'_c \subseteq Q_c\) without firing a transition \(p\) means that all states in \(Q'_c\) do not guarantee \(p\) and no finite sequence to such state is guaranteed from these states, which is in contradiction with our assumption. In addition, a state \(q_{conc} \in Q_c\) such that \(q_{conc} \xrightarrow{p}\) indeed exists as it is guaranteed by the fact that \(\overline{R}_T^{-1}(t_p) \neq \emptyset\) (see Definition 4.4.1).

The choice of refinement under context is one among other possible relations, however this choice is highly motivated by the fact that this relation allows the property of circular reasoning. A proof is given in Section 4.5.1.

Example 4.4.4 Figure 4.9 gives an example of an abstract component \(K^a\) and a concrete component \(K^c\) such that \(K^c \sqsubseteq_{K^E, I} K^a\). The relation between variables is given by \(\alpha\) denoting that the variable \(y\) of \(K^a\) is computed as the sum of two variables \(x_1\) and \(x_2\) of \(K^c\). The interaction model \(I\) specifies how each component interact with the environment \(K^E\). Note that labels which are not ports and thus are not connected by \(I\) represent internal transitions of components (\(\tau\)-transitions).

To check if the component \(K^c\) refines \(K^a\) in the context \((K^E, I)\) and according to Definition 4.4.2 we start by the initial states of these components and check if they are related by \(R\). For this purpose we have first to prove that, given the relation \(\alpha\) between concrete and abstract variables, we have an implication from the concrete invariant to the abstract one. This is verified for the initial states as \((x_1 = x_2 = 0) \land (y = x_1 + x_2) \implies (y = 0)\). Second we have to check that all (non internal) outgoing transitions of \(S^c_1\), which are allowed by the environment \(K^E\) and by the interaction model \(I\), are also outgoing transitions of \(S^a_1\), which is the case of the transition labeled by get. Similarly, we have to prove now that the target states of this transition (get) are also related by \(R\).
The different refinement conditions are successfully applied to the different states of these components and the obtained states in relations are: \((S^c_1, S^a_1) \mathcal{R} S^c_2\), \((S^c_2, S^a_2) \mathcal{R} S^c_3\) and \((S^c_3, S^a_2) \mathcal{R} S^c_3\).

\(\tau\)-Transitions are internal transitions, thus they may not be offered by \(K^a\). Note that the abstract state \(S^a_2\) is refined by two concrete states. This is in general the case when one wants to give more details about the concrete variables of a component or to explicit some (implicit) actions that have been hidden in the abstract description of some states. Figure 4.9 describes only the safety part of

\[ \begin{align*}
K^c \quad & X_1 \quad S^c_1 \quad (X_1=X_2=0) \quad \text{get} \quad \text{give} \quad \\
& \quad \tau \quad \text{wait} \quad \text{get} \quad \text{give} \quad \text{wait} \\
K^a \quad & Y = X_1 + X_2 \quad (Y > 0) \quad \text{give} \quad \text{get} \quad \text{wait} \quad \\
& \quad \tau \quad \text{wait} \quad \text{get} \quad \text{give} \quad \text{wait} \\
K^b \quad & X_2 \quad S^c_2 \quad \text{X}_2=0 \land X_1>0 \quad \text{get} \quad \text{give} \\
K^c \quad & X_1 \quad S^c_3 \quad \text{X}_1=0 \land X_2>0 \quad \text{get} \quad \text{give} \\
K^e \quad & \quad \tau \quad \text{wait} \quad \text{get} \quad \text{give} \\
\mathcal{I} \quad & \quad \tau \quad \text{wait} \quad \text{get} \quad \text{give} \\
\end{align*} \]

Figure 4.9: Refinement: \(K^c \sqsubseteq_{K^c, \mathcal{I}} K^a\).

\(K^b\) and \(K^c\).

To illustrate the progress refinement part, let suppose that \(Pr^a = (S^a_2 \xrightarrow{wait} *, S^a_2 \xrightarrow{give} *) \in \text{Prog}_{K^a}\). There are several ways to guarantee this progress by \(K^c\). We have \(\overline{\mathcal{R}}^{-1}_T(\{S^a_2\}) = \{S^c_2, S^c_3\}\). Then to guarantee \(Pr^a\) it is sufficient that \(K^c\) guarantees one of the following conditions:

1. \(\{S^c_2 \xrightarrow{wait} *, S^c_2 \xrightarrow{give} *\}\) and \(\{S^c_3 \xrightarrow{wait} *, S^c_3 \xrightarrow{give} *\}\).
2. \(\{S^c_2 \xrightarrow{wait} *, S^c_3 \xrightarrow{give} *\}\) and \(\{S^c_3 \xrightarrow{wait} *, S^c_3 \xrightarrow{\tau} S^c_3\}\).
3. \(\{S^c_3 \xrightarrow{wait} *, S^c_3 \xrightarrow{give} *\}\) and \(\{S^c_2 \xrightarrow{wait} *, S^c_2 \xrightarrow{\tau} S^c_3\}\).

4.4.2 Conformance

To define conformance we first introduce the notion of empty context for an interface of a component.

80
Definition 4.4.5 (Empty context) An empty context for an interface $\mathcal{P}$ is a pair $(E, \mathcal{I})$ where, $E$ defines any component and $\mathcal{I}$ is the interaction model defined by the set of observation connectors $\{\gamma_i = (p_i, \{p_i\}) \mid \forall p_i \in \mathcal{P}\}$ and thus $\mathcal{P}_{\mathcal{I}} = \mathcal{P}$.

Intuitively, an empty context $(E, \mathcal{I})$ for an interface $\mathcal{P}$ of a component $K$ means a context that does not connect $K$ to $E$ and an $\mathcal{I}$ that exports only ports of $K$ which means that the behavior of $E$ is irrelevant. We now define conformance as a refinement in an empty context.

Definition 4.4.6 (Conformance) Let be $K_1$ and $K_2$ two components defined on the same interface $\mathcal{P}$. Let $(E, \mathcal{I})$ be an empty context on $\mathcal{P}$.

$$K_1 \preceq K_2 \iff K_1 \sqsubseteq_{(E, \mathcal{I})} K_2$$

Intuitively, this relation of conformance is a simulation of the safety part of $K_1$ to $K_2$ and an inclusion of infinite executions of the progress description of $K_2$ into those of $K_1$.

Example 4.4.7 Figure[4.9] represents the component $K^c$ which refines the component $K^a$ in the context $(K^E, \mathcal{I})$. This allows to deduce that the closed system obtained as $\mathcal{I}(K^c, K^E)$ conforms to the closed system $\mathcal{I}(K^a, K^E)$.

We now have all the necessary ingredients that define a contract framework.

Theorem 4.4.8 We have defined a contract framework. Furthermore, if assumptions are deterministic, then circular reasoning is sound for the refinement under context of Definition 4.4.2.

We now have to prove that such a framework is well formed to support the methodology, described in Chapter[3] that is its a well-defined contract framework with a refinement under context ensuring the property of circular reasoning. See Proof[4.5.1].

4.5 Proofs

In this section we detail the proof of the Theorem 4.4.8 and the proofs of all related properties.

4.5.1 Well-definedness of the contract framework

We detail here the proof steps required to show that we have first a component framework as required in Definition 2.2.1. Second, we prove that this component framework together with the relations of refinement under context and conformance, described in Section 4.4, define indeed a contract framework in which circular reasoning is sound.
Well-defined Component framework. In the proposed component framework, we are using a variant of BIP component framework. $GL$ is the set of interactions models given as connectors described in Section 4.2. Their composition operator $\circ$ of $BIP$ component framework.

In the proposed component framework, we are using a variant of BIP component framework. Their composition operator $\circ$ defined in Section 4.2.3 allows flattening as the merge of two connectors, that is a hierarchical connector, allows to build a new flat connector. Composition of interaction models is thus defined as a union of a set of disjoint connectors, which allows to deduce that $(GL, \circ)$ is a commutative monoid, i.e. it ensures associativity, identity and of course commutativity (see [BS08a]):

1. $\forall I_1, I_2, I_3 \in GL. (I_1 \circ I_2) \circ I_3 = I_1 \circ (I_2 \circ I_3)$
2. $\exists I_e \in GL. \forall I \in GL. I_e \circ I = I \circ I_e = I$
3. $\forall I_1, I_2 \in GL. I_1 \circ I_2 = I_2 \circ I_1$

We have also to prove that $I_1 \{I_2\{K_1\}, K_2\} = (I_1 \circ I_2)\{K_1 \cup K_2\}$, where $K_1$ and $K_2$ are set of components. In other words, composition of components is consistent with composition of components. Here, the equivalence is defined as the syntactic equality.

Proof We write the proof for $K_1 = \{K_1, K_2\}$ and $K_2 = \{K_3\}$ and. The generalization should be clear. We also suppose w.l.o.g that each interaction model consists into only one connector. This equivalence is trivial if the two connectors of each interaction model are disjoint as in this case $\circ$ is defined by the union operator which guarantees this equivalence. Thus we can reduce the proof to a merge of two connectors. Let be $I_1 = \gamma_1$ and $I_2 = \gamma_2$, where the exported port of $\gamma_2$, $p_{\gamma_2} \in P_{\gamma_1}$, $P_{\gamma_1} = \{p_{\gamma_1}, p_3\}$ with $p_3 \in P_{K_3}$ and $P_{\gamma_2} = \{p_1, p_2\}$. For the sake of clarity and w.l.o.g we also suppose that each component $K_i$ has only $\{p_i\}$ as interface that is $P_{K_i} = \{p_i\}$.

$I_1 \{I_2\{K_1, K_2\}, K_3\} = (TS, X, Inv, g, Fu, Prog)$, where:

- $TS = (Q, q^0, \mathcal{P} \cup \{\tau\}, \rightarrow)$
- $Q = ((Q_1 \times Q_2) \times Q_3)$
- $q^0 = ((q_1^0, q_2^0), q_3^0)$
- $\mathcal{P} = p_{\gamma_1}$
- $X^{st} = (X_1^{st} \cup X_2^{st}) \cup X_3^{st}$, $X^{tr} = (X_1^{tr} \cup X_2^{tr} \cup X_{I_2}) \cup X_2^{tr} \cup X_3^{tr} \cup X_{I_1}$
- $Inv = ((Inv_{q_1} \land Inv_{q_2}) \land Inv_{q_3})$
- for $q = ((q_1, q_2), q_3) \in Q$ and $q' = ((q_1', q_2'), q_3') \in Q$, the transition $t = q \xrightarrow{p} q' \in \rightarrow$ iff $p = p_{\gamma_1}$ and it exists local transitions $t_{12} = (q_1, q_2) \xrightarrow{p_{12}, (q_1', q_2')} \in \rightarrow I_2\{K_1, K_2\}$ and $t_3 = q_3 \xrightarrow{p_3} q_3' \in \rightarrow K_3$ with $gt_1 = g_{t_{12}} \land g_{t_3} \land R_{t_{12}} \land R_3 \land G_{\gamma_1}$. Besides $t_{12} = (q_1, q_2) \xrightarrow{p_{12}, (q_1', q_2')} \in \rightarrow I_2\{K_1, K_2\}$ means that it exists $t_1 = q_1 \xrightarrow{p_1} q_1' \in \rightarrow K_1$ and $t_2 = q_2 \xrightarrow{p_2} q_2' \in \rightarrow K_2$ with $gt_{12} = g_{t_1} \land g_{t_2} \land R_1 \land R_2 \land G_{\gamma_2}$. 

$(I_1 \circ I_2)\{K_1, K_2, K_3\} = (TS, X, Inv, g, Fu, Prog)$, where:
CHAPTER 4. A CONTRACT FRAMEWORK FOR COMPONENTS WITH DATA

- $TS = (Q, q^0, \mathcal{P}_E \cup \{\tau\}, \rightarrow)$
- $Q = (Q_1 \times Q_2 \times Q_3)$
- $q^0 = (q^0_1, q^0_2, q^0_3)$
- $\mathcal{P} = p_{\gamma_1}$ (according to the definition of merge of connectors).
- $X^{st} = (X_1^{st} \cup X_2^{st}) \cup X_3^{st}$, $X^{tr} = (X_1^{tr} \cup X_2^{tr} \cup X_3^{tr}) \cup (X_{I_2} \cup X_{I_1}$ according to the Definition 4.2.8 of merge of connectors. Note that if the two connectors are observation connectors then, not variables are associated to them.
- $Inv_q = Inv_{q_1} \land Inv_{q_2} \land Inv_{q_3}$
- $Inv_0 = Inv_{q_1} \land Inv_{q_2} \land Inv_{q_3}$

Thus after renaming of states of $\mathcal{I}_1 \{I_2 \{K_1, K_2\}, K_3\}$ which are $((q_1, q_2), q_3)$ become of the form $(q_1, q_2, q_3)$ and if we extend as below to deal with observation connectors, we can obtain easily the syntactic equality between both of the composite components.

Well-defined Contract framework. Here, we prove that conformance is transitive, then that refinement implies conformance and finally that our relation of refinement is preserved by composition and it allows circular reasoning.

- The transitivity of conformance is trivially deduced from the transitivity of the relation of refinement.
- Preservation of refinement by composition: $K_1 \subseteq_{I_E \{E_1, E_2\}} K_2 \implies \mathcal{I}_1 \{K_1, E_1\} \subseteq_{E_2, I_2} \mathcal{I}_1 \{K_2, E_1\}$ where $\mathcal{I}_1$ and $\mathcal{I}_2$ are the composition operators obtained from $\mathcal{I}$ and $\mathcal{I}_E$ using flattening and decomposition.

Proof Let suppose that $K_1 \subseteq_{I_E \{E_1, E_2\}} K_2$ and that $K_i = (TS_i, X_i, Inv_i, g_i, Fu_i, Prog_i)$, $E_i = (TS_i, X_i, Inv_{E_i}, g_{E_i}, Fu_{E_i}, Prog_{E_i})$ where $TS_i = (Q_i, q^0_i, \mathcal{P}_i, \rightarrow_i)$. This means according to Definition 4.4.2 of refinement that it exists a relation $R \subseteq (Q_1 \times (Q_{E_1} \times Q_{E_2})) \times (Q_2 \times Q_{E_1})$ with the stated properties. We define the relation $R' \subseteq ((Q_1 \times Q_{E_1}) \times Q_{E_2}) \times (Q_2 \times Q_{E_1})$ by

$((q_1, q_{E_1}), q_{E_2}) R' (q_2, q_{E_1}) \iff (q_1, (q_{E_1}, q_{E_2})) R q_2$

Let us prove that $R'$ has the expected properties:
- $(q^0_1, q^0_{E_1}), R' (q^0_2, q^0_{E_1})$ holds by definition.
- We have that $Inv_0 \land \alpha'(X_1, X_2) \implies Inv_{q_2}$, we have to prove that $\exists \alpha'(X_1 \cup X_{E_1}, X_2 \cup X_{E_1}) (Inv_{q_1} \land Inv_{q_{E_1}}) \land \alpha'(X_1 \cup X_{E_1}, X_2 \cup X_{E_1}) \implies (Inv_{q_2} \land Inv_{q_{E_1}})$. For it is sufficient to
choose $\alpha'$ such that $\alpha' \implies \alpha$.

- Suppose now that $((q_1, q_{E_1}), q_{E_2}) R'(q_2, q_{E_2})$ and that $t_1 = ((q_1, q_{E_1}), q_{E_2})_\text{pI}_1$
  $\implies ((q_1, q_{E_1}), q_{E_2}) \in \Sem_{\text{pI}_1}(I_1(K_1, E_1), E_2)$ we have to prove that $\exists (q', q'_1)$ such that $t_2 = ((q_2, q_{E_1}), q_{E_2})_\text{pI}_2\implies ((q', q'_1), q_{E_2}) \in \Sem_{\text{pI}_2}(I_2(K_2, E_1), E_2)$ and $((q_1, q_{E_1}), q_{E_2}) R'(q_2, q_{E_2})$.
  
  $t_1 = ((q_1, q_{E_1}), q_{E_2})_\text{pI}_1\implies ((q_1, q_{E_1}), q_{E_2}) \in \Sem_{\text{pI}_1}(I_1(K_1, E_1), E_2)$ implies w.l.o.g that $p_{E_1}, p_{E_2} \in \mathcal{P}_{I_2}$ such that $t_2$ corresponds to the synchronization of the two transitions labeled by these ports. Note that here we suppose w.l.o.g that $I_2$ and $I_2$ have only rendezvous connectors as the observation connector can be then easily deduced. We suppose that $I_2$ has only the connector $\gamma_1$ connecting $p_{E_1}$ and $p_{E_2}$, similarly $I_1$ has only the connector $\gamma_1$ connecting $p$ and $p_{E_1}$ where $p$ is a port of $K_1$. This implies that $(q_1, q_{E_1})_\text{pI}_1\implies (q_1', q_{E_1}) \in \Sem_{\text{pI}_1}(I_1(K_1, E_1)$ and $q_{E_2} = q_{E_2}' \in \Sem_{\text{pI}_2}(I_2(K_2, E_1), E_2)$ implies that it exists a valuations of the variables $X_1, X_{E_1}, X_{E_2}$ for which the following predicate holds: $(\Inv_{q_1} \land \Inv_{q_{E_1}} \land \Inv_{q_{E_2}}) \land (G_{\gamma_1} \land G_{\gamma_2} \land (\Inv_{q'_1} \land \Inv_{q_{E_1}} \land \Inv_{q_{E_2}})$ which implies that $(\Inv_{q_1} \land \Inv_{q_{E_1}}) \land (G_{\gamma_1}) \land (\Inv_{q'_1} \land \Inv_{q_{E_1}})$.
  
  Which means that $(q_1, q_{E_1})_\text{pI}_1\implies (q_1', q_{E_1}) \in \Sem_{\text{pI}_1}(I_1(K_1, E_1)$ and as $(q_1, (q_{E_1}, q_{E_2})) R q_2$ we obtain that it exists $q'_2$ such that $(q_2, (q_{E_1}, q_{E_2}))_\text{pI}_2\implies (q'_2, (q'_{E_1}, q'_{E_2})) \in \Sem_{\text{pI}_2}(I_2(K_2, E_1), E_2)$ and $(q_1', (q'_{E_1}, q'_{E_2})) R q'_2$ which implies by definition that $((q_1, q_{E_1}), q_{E_2}) R'(q_2, q_{E_2})$.
  
- $\tau$ transitions of $I_1(K_1, E_1)$ are $\tau$ transitions of $K_1$ as we suppose that the environment does not have $\tau$ transitions. Then we use $R$ to establish the needed relation.

  - for the progress part, we have that $K_1 \equiv_{R \setminus E} K_2$ this means that progress of $K_2$ can guarantee in $I_2 \setminus E \equiv_{R \setminus E} K_2$ this means that progress of $K_2$ can guarantee in $I_2 \setminus E \equiv_{R \setminus E} K_2$. As refinement implies conformance we obtain that $I \{K_1, I \setminus E \{E_1, E_2\}\} \implies I \{K_1, I \setminus E \{E_1, E_2\}\}$, besides we have that $I \{K_1, I \setminus E \{E_1, E_2\}\} \equiv I \{I \setminus E \{E_1, E_2\}\}$ this allows to deduce that $I \{I \setminus E \{E_1, E_2\}\} \equiv I \{I \setminus E \{E_1, E_2\}\}$ which means that preservation of progress conditions.

\[\square\]

- Conformance is consistent with refinement under context:
  $K_1 \equiv_{R \setminus E} K_2 \implies I \{K_1, E\} \implies I \{K_2, E\}$. \textbf{Proof} Suppose that $K_1 \equiv_{R \setminus E} K_2$ to prove conformance we have to prove that $I \{K_1, E\} \equiv_{R \setminus E} I \{K_2, E\}$ which is deduced from the preservation of refinement by composition.

\[\square\]

\textbf{Circular reasoning.}

- Circular reasoning is sound: $K \equiv_{A,G} G \land E \equiv_{G,J} A \implies K \equiv_{E,J} G$.  

84
CHAPTER 4. A CONTRACT FRAMEWORK FOR COMPONENTS WITH DATA

Proof Let \( K \) be a component on an interface \( P, (E, \mathcal{I}) \) a context for \( P \) and \( \mathcal{C} = (A, \mathcal{I}, G) \) a contract for \( P \). We suppose that \( K \subseteq_A \mathcal{I} \subseteq E \subseteq G \subseteq \mathcal{I} \subseteq A \). We have to prove that \( K \subseteq E \). 

Given two relations \( \alpha_1(X_K, X_G) \) and \( \alpha_2(X_E, X_A) \) and as \( K \subseteq A \subseteq E \subseteq G \subseteq A \), there exist two relations \( R_1 \) and \( R_2 \) on respectively \( (Q_K \times Q_A) \times Q_G \) and \( (Q_E \times Q_G) \times Q_A \), verifying the conditions of Definition 4.4.2.

We define \( R \subseteq (Q_K \times Q_E) \times Q_G \) as follows: for \( q_K \in Q_K, q_E \in Q_E, q_G \in Q_G \), we define \( (q_K, q_E) \) \( \mathcal{R} \) \( q_G \) iff there exists \( q_A \) such that \( (q_K, q_A) \) \( \mathcal{R}_1 \) \( q_G \) and \( (q_E, q_G) \) \( \mathcal{R}_2 \) \( q_A \). \( (V_i \) denotes a valuation of \( X_i \).

Safety part. We have \( (q_K^0, q_A^0) \) \( \mathcal{R}_1 \) \( q_G^0 \) and for all \( q_K \in Q_K, q_G \in Q_G, q_A \in Q_A, (q_K, q_A) \) \( \mathcal{R}_1 \) \( q_G \) implies:

1. \( I_{q_K}(X_K) \land \alpha^1(X_K, X_G) \implies I_{q_G}(X_G) \)
2. \( \forall p[x] \in P, \) the following holds:
   a. for any value \( v \) of \( x \) : \( \exists t_K = q_K \xrightarrow{p} q'_K \) and \( \forall V_K, \forall q_G \) satisfying \( \text{Sem}_{t_K} \) implies \( \exists q_G, t_G = q_G \xrightarrow{p} q_G' \) and \( \forall t_G, \forall q_G \) consistent with \( \alpha^1 \) and satisfying \( \text{Sem}_{t_G} \).
   b. \( \exists \gamma. P_\gamma = \{p, p_A\} \land (q_K, q_A) \xrightarrow{p} (q_K', q_A') \implies (q_K', q_A') \) \( \mathcal{R}_1 \) \( q_G' \) with \( q_G' \) as above.
3. \( q_K \xrightarrow{\tau} q_K' \implies (q_K', q_A) \) \( \mathcal{R}_1 \) \( q_G \)

Besides, we have \( (q_E^0, q_A^0) \) \( \mathcal{R}_2 \) \( q_A^0 \) and for all \( q_E \in Q_E, q_A \in Q_A, q_G \in Q_G, (q_E, q_G) \) \( \mathcal{R}_2 \) \( q_A \) implies:

1. \( I_{q_E}(X_E) \land \alpha^2(X_E, X_A) \implies I_{q_A}(X_A) \)
2. \( \forall p[x] \in P, \) the following holds:
   a. for any value \( v \) of \( x \) : \( \exists t_E = q_E \xrightarrow{p} q_E' \) and \( \forall V_E, \forall q_G \) satisfying \( \text{Sem}_{t_E} \) implies \( \exists q_A, t_A = q_A \xrightarrow{p} q_A' \) and \( \forall t_A, \forall q_A \) consistent with \( \alpha^2 \) and satisfying \( \text{Sem}_{t_A} \).
   b. \( \exists \gamma. P_\gamma = \{p, p_G\} \land (q_E, q_G) \xrightarrow{p} (q_E', q_G') \implies (q_E', q_G') \) \( \mathcal{R}_2 \) \( q_A' \) with \( q_A' \) as above.
3. \( q_E \xrightarrow{\tau} q_E' \implies (q_E', q_G) \) \( \mathcal{R}_2 \) \( q_A \)

We prove now that \( R \) is the relation we are looking for. Let \( q_K \in Q_K, q_E \in Q_E, q_G \in Q_G \) be such that \( (q_K, q_E) \) \( \mathcal{R} \) \( q_G \). Let \( q_A \) be such that \( (q_K, q_A) \) \( \mathcal{R}_1 \) \( q_G \) and \( (q_E, q_G) \) \( \mathcal{R}_2 \) \( q_A \). We have to prove that:

1. \( I_{q_K}(X_K) \land \alpha'(X_K, X_G) \implies I_{q_G}(X_G) \)
2. \( \forall p[x] \in P, \) the following holds:
   a. for any value \( v \) of \( x \) : \( \exists t_K = q_K \xrightarrow{p} q_K' \) and \( \forall V_K, \forall q_G \) satisfying \( \text{Sem}_{t_K} \) implies \( \exists q_G, t_G = q_G \xrightarrow{p} q_G' \) and \( \forall t_G, \forall q_G \) consistent with \( \alpha \) and satisfying \( \text{Sem}_{t_G} \).
(b) \( \exists \gamma, P_\gamma = \{p, p_E\} \land (q_K, q_E) \overset{p_E}{\rightarrow} (q'_K, q'_E) \implies (q'_K, q'_E) \mathcal{R} q'_G \) with \( q'_G \) as above.

3. \( q_K \overset{\tau}{\rightarrow} q'_K \implies (q'_K, q'_E) \mathcal{R} q'_G \)

- **Condition 1.** Given \( \alpha' = \alpha_1 \), condition 1 is the same as condition 1 of \( \mathcal{R}_1 \)

- **Condition 2.**
  
  Part (a): Deducted from condition 2, part (a) for \( \mathcal{R}_1 \).
  
  Part (b): Let us suppose that \( q_K \overset{p}{\rightarrow} q'_K \land q_E \overset{p_E}{\rightarrow} q'_E \land (\exists \gamma \ s.t. \ P_\gamma = \{p, p_E\}) \). Condition 2 for \( \mathcal{R}_2 \) implies that \( \exists q'_A \in \mathbb{Q}_A \ s.t. q_A \overset{p_E}{\rightarrow} q'_A \).

  Hence: \( q_K \overset{p}{\rightarrow} q'_K \land q_A \overset{p_E}{\rightarrow} q'_A \land (\exists \gamma \ s.t. \ P_\gamma = \{p, p_E\}) \).

  Thus Condition 2 for \( \mathcal{R}_1 \) implies that there exists a \( q''_G \) s.t. \( (q_G \overset{p}{\rightarrow} q''_G) \land (\forall q'_A \ s.t. q_A \overset{p_E}{\rightarrow} q'_A) \)

  \( q'_A \implies (q'_K, q'_E) \mathcal{R}_1 q''_G \). We now want to prove that \( \forall q'_E, q_E \overset{p_E}{\rightarrow} q'_E \implies (q'_K, q'_E) \mathcal{R} q''_G \).

  Let us fix \( q'_E \) such that \( q_E \overset{p_E}{\rightarrow} q'_E \). So we have:

  \(- q_E \overset{p_E}{\rightarrow} q'_E \)

  \(- (\exists \gamma \ s.t. \ P_\gamma = \{p, p_E\}) \land q_K \overset{p}{\rightarrow} q'_K \).

  Thus condition 2 part (2) for \( \mathcal{R}_1 \) implies \( \exists q''_G \in \mathbb{Q}_G \ s.t. q_G \overset{p}{\rightarrow} q''_G \). So condition 2 for \( \mathcal{R}_2 \) implies that it exists a \( q''_A \) such that \( q_A \overset{p}{\rightarrow} q''_A \) and \( \forall q'_G, q_G \overset{p}{\rightarrow} q'_G \implies (q'_E, q'_G) \mathcal{R}_2 q''_A \).

  We apply this relation to \( q''_G \) and Similarly we apply \( \forall q'_A, q_A \overset{p_E}{\rightarrow} q'_A \implies (q'_K, q'_A) \mathcal{R}_1 q''_G \) to \( q''_A \). Thus we get:

  \(- (q'_E, q''_G) \mathcal{R}_2 q''_A \)

  \(- (q'_K, q''_A) \mathcal{R}_1 q''_G \)

  Hence \( (q'_K, q'_E) \mathcal{R} q''_G \).

- **Condition 3.** Let us suppose that \( q_K \overset{\tau}{\rightarrow} q'_K \). Thus condition 3 of \( \mathcal{R}_1 \) implies that \( (q'_K, q_A) \mathcal{R}_1 q_G \). Besides we have \( (q_E, q_G) \mathcal{R}_2 q_A \). Hence \( (q'_K, q_E) \mathcal{R} q_G \)

**Progress part.** As the progress part does not involve the context, the progress part for \( \mathcal{R} \) is deduced from the progress properties for \( \mathcal{R}_1 \). \( \square \)
Chapter 5

An Application: Resource Sharing in a Networked System

In this Chapter we apply our approach to a case-study “Sharing resource system” to verify some global requirement. We choose this system because it can be defined recursively, thus we can apply the extension of our methodology proposed to verify recursively defined systems. We also apply our methodology to verify a top-level progress requirement of this system using the rich contract framework proposed in the previous chapter. In addition, we report experimentations of the different verification steps using a tool we have developed for this purpose.

5.1 Sharing resource system

We apply the proposed methodology to a variant of an algorithm for sharing resources in a network presented in [DDHL08]. The overall structure of the application is given in Figure 5.1 where the system structured as binary trees of nodes defining a token ring. We may also consider more realistically, trees with arbitrary branching, but we restrict ourselves to binary branching for simplifying the presentation.

Resources shared between nodes are represented by tokens circulating in packets containing one or more tokens along the token ring. Red arrows in Figure 5.1 gives how these resources circulate in the network. The value of a packet is the number of tokens it contains. Nodes are trying to collect tokens so that they can perform some computation. There is a particular unique token in the network called the privilege and denoted $P$. This privilege allows nodes to accumulate tokens. A node without this (unique) privilege may either use the tokens of the just received packet or, if it needs no or more resources, it must keep the tokens circulating and wait until it receives a larger packet or the privilege. According this resource sharing algorithm, a node in the network behaves as follow (see Figure 5.2):

- A node may request tokens and a variable $Req$ indicates the numbers of tokens requested;
- When it has enough tokens for satisfying its request, it is expected to use them, and relax the privilege $P$ if it has it;
Figure 5.1: The overall structure of the application.

- When it has resources (tokens) in use, it cannot request additional ones; it may later free them or keep them forever;
- A node can raise a request only when it has no resource in use and no pending request means when \( Req = 0 \).

The number of tokens circulating in the network (and not in use) are denoted by the variable \( Tk \).

We assume that connectivity of the network is guaranteed and tokens are never lost. Here, this assumption is encoded in the composition operator. This allows separating completely design and correctness proofs from the resource sharing algorithm and the algorithm guaranteeing connectivity, which is typically implemented in a lower layer of the overall network protocol.

We represent networks of arbitrary size by a grammar and associating a contract with each node, such that the correctness proof boils down to a set of small verification steps as proposed by the methodology steps detailed in Section 3.1.

As described in Figure 5.1, a network can be built as just one node, or as a node connected to two networks. Thus the system can be defined by the following grammar \( \mathcal{G} \), where \( \{E, Node\} \) are terminals and \( \{Sys, Net\} \) nonterminals with axiom \( Sys \). The rules are:

1. \( Sys \rightarrow I_{Net}(E, Net) \)
2. \( Net \rightarrow Node \)
3. \( Net \rightarrow I(Node, Net, Net) \)
4. \( Node \rightarrow I_{Node} \)

Note here that \( Sys \) defines a closed system built as a composition of the network and its environment. The goal is to use our methodology to verify that the described system ensures some top-level requirement. The starting point is both a high-level requirement and an abstract description of the behavior of an individual node.

The described algorithm ensures, under weak assumptions, a safety, a progress and a fairness property. The safety property expresses that the global number of tokens in the system is constant.
CHAPTER 5. AN APPLICATION: RESOURCE SHARING IN A NETWORKED SYSTEM

The progress property is that, as long as there are tokens available, some node will be served. Fairness ensures that if there are tokens available for long enough, every node making a request will get into a privileged position where it is served with some priority. In this chapter we focus on the progress property.

We use components given by the Definition 4.1.1 to model the behaviors and the interfaces of nodes and networks. The node interface is given in Figure 5.2 where tokens (and the privilege) are exchanged through ports get_T (get_P) and give_T (give_P). The port req models the action of requesting some resources by the node. The port use represents the action of using some resources by the node. When the resources are no more in use by the node, it liberates them by an action labeled by the port free.

The Node has state variables indicating whether it has the privilege (P), the number of tokens it has (Tk), the number of token it requests (Req). It has also some port variables used during interactions. In the initial state of the node, see Figure 5.2 (b), the node has no request which means that Req = 0 (State S_1). Once it makes a request, then 3 cases are possible. The first case (State S_2) corresponds to a request that is smaller than the number of token in the node (0 < Req < Tk), in this state, the node has enough token to perform the action use. The second case corresponds to state S_3, where the node does not have enough tokens (Req > Tk) and the node does not have the privilege (¬P). Thus the node has to keep circulating packets of tokens until it gets a packet with a sufficient number of tokens or it gets the privilege token and thus goes to state S_4 which corresponds to the third case. This means that the node does not have enough tokens but it has the privilege (Req > Tk ∧ P) which allows it to accumulate tokens until it reaches the requested number.

Note that in the interface of the node, depicted in Figure 5.2 (a), ports get_T (get_P) and give_T (give_P) are synchron ports, as a node has to synchronize with the rest of the network when it has to give or to get a token (similarly for the privilege). However, ports use, req and free are trigger ports as the node can perform these actions locally with no synchronization.

In this application what we have is the given description of a node and some top-level requirements that a network built as a composition of these nodes has to ensure. Here we choose to verify a progress top-level requirement.
5.1.1 The top-level requirement $\phi$.

We consider here one of the top-level requirements of the algorithm, a progress requirement $\phi$. This requirement has to be ensured by the closed system $Sys$ defined by a network, built by a set of nodes according to the structure depicted in Figure 5.1 and an environment (see the first rule of the grammar $G$ given previously in Section 5.1).

To specify the network and its requirement, we define a variable $R_x$ denoting the maximal request among the set of requests of the nodes of the closed system and we denote by $Tk$ the number of tokens circulating in this system.

The top-level requirement $\phi$ states that “as long as the requests of nodes are reasonable, some of the nodes will be served.” Let be Reasonable requests means that there is no request made by a node that is bigger than the number of tokens in the system $0 < R_x \leq Tk$. According to the description given in Figure 5.2 a node is served means that it performs the transition labeled by the port use. Thus $\phi$ means that whenever $0 < R_x \leq Tk$, some use action will occur.

$$
pr_1 = (T, S_2 \xrightarrow{\text{use}} *)
$$

$$
pr_2 = (\{ S_1 \xleftarrow{\text{free}} S_2, S_2 \xrightarrow{\text{req}} S_1 \}, S_2 \xleftarrow{\text{use}} *)
$$

Figure 5.3: Top-level requirement $\phi$

Figure 5.3 gives a description of $\phi$ in our formalism. This progress requirement is particularly specified by the two progress properties $pr_1$ and $pr_2$. The first property $pr_1$ says that “it is not possible to stay forever in state $S_2$, without performing a use-transition”. The progress property $pr_2$ says that “it is not possible to switch infinitely often between states $S_1$ and $S_2$ (that is, free occurs infinitely often) without that a use occurs infinitely often as well” (progress transitions use are represented by red arrows). When the system has no request pending or when there are not enough tokens to serve the maximal request (state $S_1$), the system has not to guarantee any progress.

5.1.2 Methodology

Now we apply the previously described verification methodology to the system described by the given grammar $G$ so as to verify if it ensures the top-level progress requirement $\phi$. For this purpose,
CHAPTER 5. AN APPLICATION: RESOURCE SHARING IN A NETWORKED SYSTEM

we instantiate the different methodology steps detailed in Section 3.1.2 to a network built according to grammar \( \mathcal{G} \), together with an actual environment \( E \):

1. We formulate a contract \( C_{\text{Net}} = (A_{\text{Net}}, I_{\text{Net}}, G_{\text{Net}}) \) associated with \( \text{Net} \) (which plays here the role of \( K_0 \)) and we show that \( I_{\text{Net}}(A_{\text{Net}}, G_{\text{Net}}) \cong \varphi \).

2. We define \( C_{\text{Node}} = (A_{\text{Node}}, I_{\text{Node}}, G_{\text{Node}}) \) a contract for \( \text{Node} \) (note that we already have a contract for \( \text{Net} \)).

3. We show that \( C_{\text{Node}} \) dominates \( C_{\text{Net}} \) and that \( \{C_{\text{Node}}, C_{\text{Net}}\} \) dominates \( C_{\text{Net}} \) w.r.t. \( I \).

4. We prove that \( I_{\text{Node}} \) satisfies \( C_{\text{Node}} \) and that \( E \) satisfies \( C_{\text{Net}}^{-1} = (G_{\text{Net}}, I_{\text{Net}}, A_{\text{Net}}) \).

Note that this proof implies that all networks that can be built according to this grammar satisfy \( \varphi \). The first step consists in defining a top-level contract of of the closed system \( \text{Sys} \) and check conformance between the description of the system given by this contract and the already given specification of the top-level requirement \( \varphi \). Here we rely on the already given description of \( \varphi \) (see Figure 5.3). To define the top-level contract, we have to give a description of the network \( G_{\text{Net}} \), of a property of its environment \( A_{\text{Net}} \) and how they are connected \( I_{\text{Net}} \) (glues). Then we have to define a contract for each subsystem consisting the network \( \text{Net} \) that is contracts: \( C_{\text{Net}} \) for a network and \( C_{\text{Node}} \) for a node. Note that the different rules of the grammar \( \mathcal{G} \) are transformed into a dominance check between contracts.

If we want to further refine the \( \text{Node} \) component, we may start by a contract \( C_{\text{Node}} = (A_{\text{Net}}, I_{\text{Net}}, G_{\text{Node}}). \) Indeed, we can always refine terminal symbols by iteratively applying the same approach to this symbol. This means that we can either give an implementation of the node or any abstract description that can be refined later.

5.1.3 Interaction models and contracts

Now we detail the steps 1 and 2 of the proposed methodology, where we have to define contracts for each subsystem. Our system can be either a node or a composition of a node with two networks, this is detailed by the rules 2 and 3 of the grammar \( \mathcal{G} \). Thus the contracts that we have to define are \( \text{Node} \) and \( \text{Network} \) contracts. To define these contracts we have to give a description of the properties that the \( \text{Node} \) and the \( \text{Network} \) have to ensure, a description of their environments and how they are connected to these environments. We recall that these subsystems \( \text{Node} \) and \( \text{Network} \) are modeled as components according to the definition of component given in the previous section.

Network and node contracts

To define the contracts \( C_{\text{Net}} = (A_{\text{Net}}, I_{\text{Net}}, G_{\text{Net}}) \) and \( C_{\text{Node}} = (A_{\text{Node}}, I_{\text{Node}}, G_{\text{Node}}) \), we first give a description of their assumptions namely \( A_{\text{Net}} \) and \( A_{\text{Node}} \).

Here we choose to give the same assumption to both contracts \( A_{\text{Net}} = A_{\text{Node}} \), as the environments of both the node and the network are supposed to ensure the same properties. This assumption is described in Figure 5.4. \( A_{\text{Net}} \) represents the environment of an arbitrary network component, and it
5.1. SHARING RESOURCE SYSTEM

has just one state, from which it can perform all its actions. The progress properties given in the right hand side of Figure 5.4 describes the following:

- $pr^A_1$: $A_{Net}$ cannot perform infinitely many use, without giving back tokens or $P$, this ensures that the assumption cannot keep always the tokens and $P$;

- $pr^A_2$: the assumption has to always offer the interaction use, and also the interactions get$_P$ and get$_T$ so that we guarantee that it cannot refuse tokens given by the network or the node.

- $pr^A_3$: this property ensures that the assumption has to give back tokens if it does not have the privilege $P$.

The guarantee $G_{Net}$ of the contract $C_{Net}$ is described in Figure 5.6 and the guarantee $G_{Node}$ of the contract $C_{Node}$ is given in Figure 5.5. Their transitions give$_T$ and get$_T$ decrease, respectively increase, the local state variable $Tk$ of the network and the node as they allow exchange tokens with their environment. Similarly, transitions give$_P$ and get$_P$ determine when they release and take

Figure 5.4: Assumption of the network and the node contract.

Figure 5.5: Guarantee of the Node Contract.
CHAPTER 5. AN APPLICATION: RESOURCE SHARING IN A NETWORKED SYSTEM

the privilege from their environment. Some Transitions of the guarantee of the node perform local functions on variables: $f_{\text{give}_P} : (Tk := Tk - tk; tk := 0)$, $f_{\text{get}_T} : (Tk := Tk + tk)$, $f_{\text{give}_P} : (\neg P)$, $f_{\text{get}_P} : (P)$, $f_{\text{use}} : \text{Req} = 0$.

The guarantee of the node is similar to the behavior of node already described in Figure 5.2, however to reason about progress, some progress properties are added in the guarantee of the node. Colored transitions represents the transitions of the promise set of the progress properties. The first 4 progress properties of the $G_{\text{Node}} (pr_1^N, pr_2^N, pr_3^N, pr_4^N)$, which represent unconditional progress, guarantee that the node has to always circulate tokens and $P$ and it cannot keep them forever. The property $pr_5^N$, ensures that the node cannot stay forever in state $S_2$ without performing use. $pr_6^N$ provides that the node cannot do infinitely many get$_T$ transitions in state $S_4$, without performing a get$_T$ allowing the node to go to state $S_2$.

Note that transition give$_T$ has a guard to make sure that $Tk$ is never negative, and transition give$_P$ has also a guard to make sure that they have $P$.

The guarantee of the network contract is slightly more nondeterministic and more complex than the node guarantee. In particular, additional variables are added comparing to the node guarantee. Variable $R^x$, as defined for the description of $\varphi$ denotes the maximal request among the set of requests of the nodes of the network. Moreover, it is not enough that a network has the privilege for being able to collect tokens. Indeed, the network may have the privilege, but it may be located in a node with no request. Thus this node cannot collect token and so does the network.

This ability of a network to collect tokens is indicated by the boolean state variable $G\text{Req}$ (see Figure 5.6). For example, in states with $R^x = 0$, all that the network has to guarantee is that it circulates the tokens and the privilege. When $R^x \neq 0$, the request could be a good or a bad request denoted

\[
\begin{align*}
pr_1^N & = (\top, S_1 \xrightarrow{\text{give}_T} \{1,3\}) \\
pr_2^N & = (\top, S_i \xrightarrow{\text{get}_T} \{i \in \{1,3\}\}) \\
pr_3^N & = (\top, S_i \xrightarrow{\text{get}_P} \{i \in \{1,3\}\}) \\
pr_4^N & = (\top, S_2 \xrightarrow{\text{use}} \top)
\end{align*}
\]
5.1. SHARING RESOURCE SYSTEM

also by the boolean variable \( G_{Req} \). A request is good if the network has \( P \) and \( P \) will be caught by a node with a pending request or when there is a packet with enough tokens for some node of the network. With a good request and enough tokens, the network ensures progress with the transition use. The guarantee \( G_{Net} \) describes also a set of progress properties (see Figure 5.6) which are similar for the first 6 properties of the node. The additional progress properties of the network are due to the fact that its behavior is less deterministic than the node behavior. For example, in state \( S_2 \) of the node, it cannot perform infinitely often use without going to state \( S_1 \) where it will give back \( P \). However, in state \( S_2 \) of the network a loop labeled by use is possible, as this use may be performed by any node of the network, thus \( pr_{8_{Net}} \) guarantees that the network cannot keep always \( P \) when staying in \( S_2 \) or \( S_4 \).

Interaction models To define contracts, we need to provide the interaction model used to compose guarantees and assumptions. For both contracts \( C_{Net} \) and \( C_{Node} \) the interaction models are the same that is \( I_{Net} = I_{Node} \) depicted in Figure 5.7. Interaction models are defined by a set of connectors, as already described in the framework presented in Chapter 4. In the description of our application we do not use the proposed an abstract description of connectors presented in the Definitions of Section 4.2, where functions on data are represented as predicates. Here, concrete upward and downward functions on data are used in connectors. We use a concrete description of connectors, because we do not propose to refine them.

Figure 5.7 describes the interaction model \( I_{Net} \) relating a network — and therefore also a node — to the rest of the system. We represent by get and give respectively port sets \( \{get_T, get_P\} \) and \( \{give_T, give_P\} \) for token and privilege exchange. For example, connector \( \{give_T[tk] \mid get_T[tk_A], \delta_G : [tk > 0], tk_A := tk\} \) pushes a positive number of tokens from the Network to the environment. Connectors relating ports use, req and free allows to observe when a network or its environment fires a transition labeled by one these ports by exporting the corresponding ports of terminals.

The dominance problem

Once the different contracts are defined, the third step of the methodology is to prove dominance for each rule defined by the grammar \( G \). That is \( C_{Node} \) dominates \( C_{Net} \) and that \( \{C_{Node, C_{Net, C_{Net}}} \} \)
dominates $C_{\text{Net}}$ w.r.t. $\mathcal{I}$.

Figure 5.8 shows the inner structure of a network component Net. The interaction model $\mathcal{I}$ builds a tree from a (root) node and two networks $Net_1$, $Net_2$. Interactions performed by the connectors depicted here are similar to those of Figure 5.7, except that they also ensure that tokens circulate in the correct order. Tokens are exchanged by the node and its connected networks through the following connectors:

- $\{\text{get}_{\text{Net.Net}}\{tk_i\}, \text{give}_{\text{Net.Net}}[tkn]\}$:
  
  $\delta:[\text{tkn}>0]$
  
  $T_{k\text{Net.Node}} := T_{k\text{Net.Node}} + tkn$
  
  $tk_{i+1}\text{MOD}3 := tk$
  
  $T_{k\text{Net.Net}_i} := T_{k\text{Net.Net}_i} - tkn; tkn := 0$

- $\{\text{get}_{\text{Net.Net}_i}[tkn], \text{give}_{\text{Net.Node}}[tk, tk_i]\}$:
  
  $\delta:[tk_i>0]$
  
  $T_{k\text{Net.Node}} := T_{k\text{Net.Node}} - tk_i$
  
  $tk_i := 0$

The set of n-ary connectors $\gamma_{\text{use}}$, $\gamma_{\text{req}}$ and $\gamma_{\text{free}}$ are used to verification goals, Indeed they help updating the values of variables and allow to detect at a higher level some actions performed locally. To prove dominance, we use the sufficient condition of circular reasoning. Thus to prove that $C_{\text{Node}}$ dominates $C_{\text{Net}}$, we need that $\mathcal{I}_{\text{Node}} = \mathcal{I}_{\text{Net}}$, which is already true and to prove that:

- $G_{\text{Node}} \sqsubseteq A_{\text{Net}}, \mathcal{I}_{\text{Net}} G_{\text{Net}}$
- $A_{\text{Net}} \sqsubseteq G_{\text{Node}}, \mathcal{I}_{\text{Node}} A_{\text{Net}}$

Then, to prove that $\{C_{\text{Node}}, C_{\text{Net}}, C_{\text{Net}}\}$ dominates $C_{\text{Net}}$ w.r.t. $\mathcal{I}$, we have to prove the following:

1. $\Pi_{\mathcal{P}_{\text{Net}}}(\mathcal{I}(G_{\text{Node}}, G_{\text{Net}}, G_{\text{Net}})) \models C_{\text{Net}}$
2. $\Pi_{\mathcal{P}_{\text{Node}}}(\mathcal{I}_1(A_{\text{Net}}, G_{\text{Net}}, G_{\text{Net}})) \models C_{\text{Node}}^{-1}$
5.2 Implementation and experimental results

As described previously, dominance, conformance and satisfaction problems are reduced to refinement under context. We have implemented a prototype tool, so that refinement under context is checked and discharged automatically by a Java tool. In this Section, we present a description of this tool with its main features. We also give some results about the application described in Section 5.1. Our tool consists basically in two modules:

- Refinement Checker Module: given two components \( K_{abs} \), \( K_{conc} \) and a context \((E,I)\), this module checks if \( K_{conc} \sqsubseteq_{E,I} K_{abs} \), according to our defined refinement relation (see Definition 4.4.2).

- Composition Module: to prove dominance, we have to perform composition of components w.r.t. a given interaction model (see the sufficient condition for dominance given in Theorem 3.3.4). Thus, this module allows to compute the composition of a set of components according to Definition 4.2.6.

The front-end of our implementation takes as input a description of a component given in Definition 4.1.3. This description written in a simple markup language (xml) and allows to describe the different parameters of a component (ports, variables, states ...). Each input file is then parsed to build an instance of a Java class called Behavior.

5.2.1 Refinement checker module.

Figure 5.9 shows the overall structure of this module. xml input files of this module describe three components representing \( K_{conc} \), \((E,I)\) and \( K_{abs} \). A parser allows to build a model of a component from each xml file. Then, the checker performs the refinement test by testing the safety part then the progress part of the refinement relation. Indeed, both parts (safety and progress) are independent so they can be checked separately.

1. Safety Test: this part of our implementation has to ensure the conditions given in the safety part of the refinement relation (Definition 4.4.2). To check the first condition, we have to manipulate invariants of states and prove implication \( (I_{qc} \wedge \alpha(X_{conc}, X_{abs}) \implies I_{qa}) \). For this purpose, we use the Sat-Solver tool Yices [DdM06] to prove implication of invariants of states variables. If the safety part is not satisfied, our tool precise that it is due to safety conditions.

2. Progress Test: progress properties of each component are specified in the xml file by a dedicated element denoted progressP. We have implemented a computation of the inverse image of the projection given in Definition 4.4.1 to check the progress condition. If progress conditions

\[ 5.2. \text{ IMPLEMENTATION AND EXPERIMENTAL RESULTS} \]

3. \( \Pi_{PNet}(\mathcal{I}_2(A_{Net}, G_{Node}, G_{Net})) \models c_{Net}^{-1} \)

These different relations are checked automatically using our prototype tools described in the next section.
CHAPTER 5. AN APPLICATION: RESOURCE SHARING IN A NETWORKED SYSTEM

Progress Test

Safety Test

Parser

Model

Refinement Module

Test Results

$K_{\text{abs}}$  $(E, I)$  $K_{\text{conc}}$

Components (xml files)

Model Generation

Safety Test

Progress Test

Figure 5.9: Refinement Checker Structure.
are not satisfied, our implementation report that the refinement is not sound due to progress conditions.

When the refinement relation is verified between \( K_{\text{conc}} \) and \( K_{\text{abs}} \) in the context \((E, \mathcal{I})\), our implementation provides the different states in relation.

Figure 5.10: Composition Module Structure.

5.2.2 Composition module.

Figure 5.10 describes the overall structure of the composition module of our implementation. This module allows to build the composition of a set of input components w.r.t. a given interaction model. The output of this module is an instance of the Java class “Behavior” denoting the composition. The interaction model \( \mathcal{I} \) of a composition, which corresponds to the set of connectors connecting the input components, is given within the components input files. Thus each component gives the set its connected ports and how they are connected.

To obtain the composition of the set of \( n \) components we are performing two main operations:

1. Syntactic composition: in this step we only perform the composition described in Definition 4.2.6. We build all possible transitions, states and we compute invariants as given in the composition rules.

2. Semantics: Once the syntactic composition is built, a semantic computation is needed as some
states and transitions may not be reachable. The semantics computation is performed according to the abstract semantics given in Definition 4.1.3.

5.2.3 Results.

The implementation of our prototype tool has been performed using 20 Java class and approximately Java functions of about 4820 l.o.c.

We have discharged the different dominance and conformance problems using our tool and we have proved the needed conditions given by the methodology steps detailed in Section 5.1.2. Table 5.2 gives the different states related by the refinement relation given by the fact that \( C_{\text{Node}} \) dominates \( C_{\text{Net}} \) (generated by our tool). Note that here we do not have to precise with which state of the assumption \( \text{Node} \) and \( \text{Network} \) states are related because the given assumption has just one state (see Figure 5.4). The next relation to prove is that \( \{C_{\text{Node}}, C_{\text{Net}}, C_{\text{Net}}\} \) dominates \( C_{\text{Net}} \) w.r.t. \( \mathcal{I} \) and which boils down to the following conditions:

- **Relation 1:** \( \Pi_{\mathcal{P}_{\text{Net}}}(\mathcal{I}(G_{\text{Node}}, G_{\text{Net}}, G_{\text{Net}})) \models C_{\text{Net}} \)
- **Relation 2:** \( \Pi_{\mathcal{P}_{\text{Node}}}(\mathcal{I}_1(A_{\text{Net}}, G_{\text{Net}}, G_{\text{Net}})) \models C^{-1}_{\text{Node}} \)
- **Relation 3:** \( \Pi_{\mathcal{P}_{\text{Net}}}(\mathcal{I}_2(A_{\text{Net}}, G_{\text{Node}}, G_{\text{Net}})) \models C^{-1}_{\text{Net}} \)

To compute the product of **Relation 1**, we use the composition module, which generates a component with 64 states and 1276 transitions. The composition needed to check **Relation 2** generates a component with 16 states and 394 transitions. **Relation 3** needs the composition \( \mathcal{I}_2(A_{\text{Net}}, G_{\text{Node}}, G_{\text{Net}}) \) which leads to a component with 16 states and 262 transitions.

<table>
<thead>
<tr>
<th>States</th>
<th>( S^\text{Node}_1 )</th>
<th>( S^\text{Node}_2 )</th>
<th>( S^\text{Node}_3 )</th>
<th>( S^\text{Node}_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S^\text{Net}_1 )</td>
<td>( R )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S^\text{Net}_2 )</td>
<td>( R )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S^\text{Net}_3 )</td>
<td></td>
<td>( R )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S^\text{Net}_4 )</td>
<td></td>
<td></td>
<td>( R )</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: \( C_{\text{Node}} \) dominates \( C_{\text{Net}} \): Related States.
### 5.2. IMPLEMENTATION AND EXPERIMENTAL RESULTS

<table>
<thead>
<tr>
<th>States ((\text{Node}, \text{Net1}, \text{Net1}))</th>
<th>(S^\text{Net}_1)</th>
<th>(S^\text{Net}_2)</th>
<th>(S^\text{Net}_3)</th>
<th>(S^\text{Net}_4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((s1,s1,s1))</td>
<td>(\mathcal{R})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>((s2,s1,s1))</td>
<td></td>
<td>(\mathcal{R})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((s2,s2,s1))</td>
<td></td>
<td></td>
<td>(\mathcal{R})</td>
<td></td>
</tr>
<tr>
<td>((s2,s1,s1))</td>
<td></td>
<td></td>
<td></td>
<td>(\mathcal{R})</td>
</tr>
<tr>
<td>((s2,s2,s1))</td>
<td></td>
<td></td>
<td>(\mathcal{R})</td>
<td></td>
</tr>
<tr>
<td>((s4,s2,s1))</td>
<td></td>
<td>(\mathcal{R})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((s1,s4,s1))</td>
<td></td>
<td>(\mathcal{R})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((s2,s1,s1))</td>
<td>(\mathcal{R})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>((s2,s1,s3))</td>
<td>(\mathcal{R})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>((s1,s2,s2))</td>
<td></td>
<td>(\mathcal{R})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((s4,s4,s2))</td>
<td></td>
<td>(\mathcal{R})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((s1,s2,s1))</td>
<td></td>
<td>(\mathcal{R})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((s4,s2,s1))</td>
<td>(\mathcal{R})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\vdots)</td>
<td>(\vdots)</td>
<td>(\vdots)</td>
<td>(\vdots)</td>
<td>(\vdots)</td>
</tr>
</tbody>
</table>

Table 5.2: Relation1: Related States.
Part III

Building Distributed Controllers for Systems with Priorities
Chapter 6

Controllers for Systems with Priorities

We have proposed in Part II a compositional design methodology for systems of components so as to verify some global property. We have shown that component frameworks in particular their interaction models provide a powerful mechanism for design and verification of large scale systems because they allow enforcing several properties structurally. In this part we study these interaction models and the question of how they can be implemented in a distributed setting. In particular, we focus here on interaction models defined by interactions and priorities on these interactions. We propose to model them as properties which have to be enforced in the system and which thus define controllers of this system.

Interactions are here simple synchronizations between components where we abstract away their guards and functions. Indeed, in Part II, interactions are structured using connectors. This structural aspect is relevant in the previous chapters, because our purpose have been to define a compositional design methodology based on the structure of the system. However, to study interactions and to implement them we abstract away their structural aspect.

In this chapter, we also abstract away the details about how these interactions are actually performed and which will be described at the implementation issue in Chapter VII.

We use in the following chapters, a formalism that is similar but more abstract than the one proposed in Part II as components are described by simple labeled transition systems rather than extended labeled transition systems.

In this chapter, we first present, in Section 6.1 a notion of component that is simpler than those already defined in the previous chapters. Indeed the components we consider here are just labeled transition system as the data transfer issue is not interesting for our purpose here. We also define a notion system corresponding to the notion of closed systems already defined. Then, we discuss how interaction models can be described as properties defining controllers of these systems. In Section 6.2, we provide a lightweight method for imposing the property of deadlock freedom, using a controller defined by priorities.
6.1 Systems and controlled systems

We introduce here an abstract representation of components and systems which is simpler than the description already used in Part II (see Definition 2.1.2). Indeed, here components are identified with an abstraction of their behavior and represented by transition systems labeled by ports without variables, guards or functions.

An interaction between these components is simply defined as sets of ports from different components where the transition corresponding to such an interaction is the synchronization of the corresponding local transitions labeled by the ports of this interaction. We consider here two types of interaction models: a first type that is interaction models defined by a set of interactions synchronizing transitions of different components and a second type that is interaction models defined by a set of interactions and a set of priorities between these interactions.

6.1.1 Components, interaction models and systems

We suppose given a set of ports \( \text{Ports} \).

**Definition 6.1.1 (Component)** A component \( K \) on a set of ports \( \mathcal{P} \subseteq \text{Ports} \) is a labeled transition system (LTS) where transitions are labeled by \( \mathcal{P} \).

We also define for a given component the notion of executions, deadlock and reachable states as previously described in Definition 2.1.3.

**Interaction models and parallel composition**

A set of components \( \{K_i\}_{i=1}^n \) for some \( n > 0 \) may be composed by means of an interaction model \( \mathcal{I} \) which defines the set of allowed global interactions corresponding to a joint execution of actions of a (non empty) subset of \( \{K_i\}_{i=1}^n \).

Here, we do not use connectors to structure interactions as previously defined in the previous chapters. Note, however, that an interaction can always be described as a rendez-vous connector without data exchange. In the sequel of this thesis, we only focus on the notion of interactions as simple synchronizations and on the semantics of interaction models as a set of interactions. Thus we provide the following simple definition.

**Definition 6.1.2 (Interaction, Interaction model)** Consider a set \( \{\mathcal{P}_i\} \) of disjoint interfaces \( \mathcal{P}_i \subseteq \text{Ports} \). An interaction \( a \) on \( \{\mathcal{P}_i\} \) is a pair \( (p, \alpha) \) where \( p \in \text{Ports} \) is called the exported port of \( a \) and \( \alpha \) is a non empty subset of \( \bigcup_i \mathcal{P}_i \) which contains at most one action in each interface \( \mathcal{P}_i \).

An interaction model \( \mathcal{I} \) on \( \{\mathcal{P}_i\} \) is a non empty set of interactions with distinct exported ports denoted \( I(\mathcal{I}) \). The exported ports of these interactions define the interface \( \mathcal{P}_I \) of \( \mathcal{I} \).

In the sequel, we adopt the convention that exported ports have the same name as their corresponding interaction, which we both denote by \( a, b, \ldots \). Also, for an interaction \( (a, \alpha) \in I(\mathcal{I}) \), \( \alpha \), which we denote \( \mathcal{a}_a \), is given in the form \( \{a_i\}_{i \in I_a} \) where \( I_a \subseteq [1, n] \) is a subset of indices such that \( a_i \in \mathcal{P}_i \) for \( i \in I_a \).
We recall here the semantics of a composite component \( I \{ K_i \} \) obtained as a composition of a set of components \( \{ K_i \} \), using the interaction model \( I \) defined as a set of interactions \( I(\mathcal{I}) \). That is, the interactions in \( I \) define the possible transitions of the composition both as they appear at the interface of the composite component and at the level of the constituting components, which may be required to synchronize.

**Definition 6.1.3 (Composition)** Let be \( I \) an interaction model on \( \{ P_i \} \) where \( \forall i, P_i \subseteq \text{Ports} \) and \( \{ K_i \} \) a set of components on \( \{ P_i \} \) as above with \( K_i = (Q_i, q_i^0, P_i, \rightarrow_i) \). We denote by \( I \{ K_i \} \) the component \((Q, q^0, P, \rightarrow)\) on \( P_I \), where:

- \( Q = \prod_{i=1}^n Q_i \) with \( q^0 = (q_1^0, \ldots, q_n^0) \)
- \( P = P_I \)
- \( \rightarrow \) is the least set of transitions satisfying the rule:

\[
\forall a \in I(\mathcal{I}) \quad \forall i \in I_a, q_i^1 \xrightarrow{a_i} q_i^2 \quad \forall i \in [1, n] \setminus I_a, q_i^1 = q_i^2
\]

As usually, this means that the component \( I \{ K_i \} \) has a transition with label \( a \) in state \( q = (q_1^1, \ldots, q_n^1) \) iff \( a \) is in interactions of \( I \) and for each \( i \in I_a \), the corresponding \( K_i \) has a transition with label \( a_i \in \alpha_a \) in \( q_i \). Firing this transition in \( I \{ K_i \} \) consists in synchronously firing the corresponding transitions in the \( K_i \) such that \( i \in I_a \) while letting unchanged the states of the components not involved in \( a \) (i.e., the \( K_i \) with \( i \notin I_a \)).

Note that an interaction model \( I \) may “compose” a single component by allowing only a subset of its ports to be executed. In this case, \( I \) defines a restriction operator as in CCS [Mil80].

**Notation 6.1.4** Given an interaction \((a, \alpha_a)\). Without loss of generality, if a port can be part of at most one interaction, we simplify further the notation of interaction by renaming also the \( \alpha_i \in \alpha_a \) into \( a \). Thus given a set of disjoint interfaces \( \{ P_i \} \), we can deduce by a simple name matching of their ports, the set of interactions defined on these interfaces. Such an interaction model is denoted \( || \). || is defined by the maximal interaction model which allows arbitrary interactions while still containing at most one port of each component to be composed.

**Priorities**

Components are called non-deterministic if they have a state in which there is a choice of several outgoing transitions. Those transitions may have different labels, in which case non-determinism is called controllable because the environment is able to choose a label and thus decide which interaction should take place, or they may have the same label (uncontrollable non-determinism).

We allow reducing controllable non-determinism by means of priorities which allow specifying that an interaction should be preferred over another whenever both are possible in a state.
A detailed description of how priorities are defined and how they can define a composite component is given in Definitions 2.1.15 and 2.1.16 of Section 2.1.6 where priorities on an interaction model \( I \) define a new interaction model denoted \( I \prec \).

The semantics of the composition of a set of components using \( I \prec \) is obtained by first applying \( I \) according to Definition 6.1.3, then by applying \(<\) according to the semantics detailed in 2.1.17. Definition 6.1.3, where in any state of the composite component \( K \), the transitions that can be executed according to \( I \prec \) are those executable according to \( K \) that are not inhibited by an interaction with higher priority. Thus \( I \prec (K) \) has generally fewer executions and fewer reachable states than \( K \), and thus defines a restriction of \( K \) which nevertheless preserves deadlock freedom, as expressed later in Lemma 6.1.8. The interest of priorities over a restriction operator as in CCS is that priorities do not introduce new deadlocks.

**Systems**

Now, we introduce the notion of *system* which corresponds to the previously notion of composite component but which cannot be further composed, that is a closed system. Indeed, in Part II, composite components can be further composed to built hierarchical components and which is a key notion on which rely the previously defined methodology. However, in this part, we only reason about closed systems defined as a composition of components. This structured representation is needed to distinguish between local and global information below, and also to define a distributed implementation in Chapter 7.

**Definition 6.1.5 (System)** A system \( Sys \) is a pair \( (\{K_i\}, I \prec) \), where the \( K_i \) are components on disjoint interfaces \( \{P_i\} \), \( I \) is an interaction model on \( \{P_i\} \) and \(<\) is a priority order on \( P_I \).

We distinguish between the system \( Sys = (\{K_i\}, I \prec) \) and the component \( K_{Sys} \) representing the behavior of \( Sys \) by an explicit global transition system, defined by \( I \prec \{K_i\} \). That is, \( K_{Sys} \) is obtained by applying first \( I \) to \( \{K_i\} \), then \(<\) on the resulting component. All notations defined for components extend to systems by identifying \( Sys = (\{K_i\}, I \prec) \) with \( K_{Sys} \).

As the main motivation of the work presented in this part is to provide a distributed implementation of systems with respect to the properties defined by the interaction models used to build these systems, we provide now some definitions allowing to distinguish between global and local notions.

**Definition 6.1.6 (Global and local states, Global and local priorities)** Consider a system \( Sys \) of the form \( (\{K_i\}, I \prec) \), where \( K_i = (Q_i, q_0^i, P_i, \rightarrow_{i}) \). Then:

- a tuple \( (q_1, \ldots, q_n) \) with \( q_i \in Q_i \) is called a global state of \( Sys \) (note that it is a state of \( K_{Sys} \)) and \( q_i \in Q_i \) a local state of \( K_i \). A local state \( q_i \) is compatible with a global state \( q \) if the \( i \)th element of the tuple \( q \) is \( q_i \).

- a priority \( a < b \) is called local if the interactions \( a \) and \( b \) have a common component, i.e., \( \alpha_a \cap \alpha_b \neq \emptyset \). Otherwise, we call this priority global.
Definition 6.1.7 (Locally ready interaction, Globally ready interaction, Enabled interaction)
Let be $Sys$ as above. Consider a global state $q = (q_1, \ldots, q_n) \in Q$, where $Q$ is the set of states of $K_{Sys}$, and an interaction $a \in \mathcal{I}$.

- $a$ is locally ready in $q_i$ iff $q_i \xrightarrow{a} i$
- $a$ is globally ready in $q$ iff $\forall i \in I_a, q_i \xrightarrow{a_i} i$
- $a$ is enabled in $q$ iff $a$ is globally ready in $q$ and no interaction with higher priority is also globally ready in $q$.

If $a$ is globally ready in $q$, we denote $\text{ind}_a^q$ the set of indexes of the components which must participate in $a$, that is, $\text{ind}_a^q = \{i_1, \ldots, i_k\}$ such that $\{K_j \mid j \in [1, k]\}$ is exactly the set of components involved in $a$ (and therefore in which $a$ is locally ready in $q$).

The distinction between an interaction that is locally ready, globally ready or enabled can be used to characterize the phases of the algorithm presented in Chapter 7 and which describes how a given interaction could be fired with respect to a given interaction model.

Lemma 6.1.8 Consider a system $Sys = (\{K_i\}, \mathcal{I}_\succ)$. If there exists in some (global) state $q$ an interaction $a$ that is globally ready, then there exists in $q$ some interaction $b$ (possibly equal to $a$) that is enabled.

This means that the application of priority rules cannot introduce any new deadlock (for a proof, see e.g. [GS04]). This is a motivation for using priorities to control a system.

6.1.2 Controllers defined by properties
The practical motivation for the work presented in this part of the thesis is to propose a distributed implementation of systems with respect to their interaction models. For this purpose, we propose, first to define the interaction models described in the previous section as properties and second to enforce these properties. In other words, our aim is controlling systems using properties defined by their interaction models. We now describe in this section different notions related to the control problem. In particular, we define what it means to control a system by some property, and to implement or refine a controller. We show that our systems can be seen as particular instances of a control problem.

We are interested in controlling systems so as to enforce some given properties. Intuitively, given an interface $\mathcal{P} \subseteq \mathcal{Ports}$ and a property $\varphi$ on this interface, a controller for $\varphi$ transforms a system $K_{Sys}$ on $\mathcal{P}$ into a component $K'_{Sys}$ on the same interface such as:

- $K'_{Sys}$ ensures the property $\varphi$
- $K'_{Sys}$ has only executions that are also executions of $K_{Sys}$.
To achieve this, a controller may forbid some transitions while allowing the rest of them to be fired.

Generally, in control theory, one distinguishes between the property that must be enforced by restricting the behavior of a given component, and the controller realizing such a restriction, where the property may be specified in a declarative manner as it is the case in Part II, but the controller must be a (deterministic) operational specification, for example an algorithm or a program. In this chapter, we are not interested in the technical means of forbidding transitions in a system. Thus we do not formalize this distinction here. However, a concrete controller described by an algorithm will be provided in the next chapter. Rather, we focus on how to define that (1) a system satisfies a property, (2) a system is controlled to enforce a property, and later (3) a property is distributed so that controlling locally its components is sufficient to control globally a system. Thus, we consider properties as (possibly non-deterministic) controllers, and we define a refinement or implementation relation between properties/controllers.

Typical properties \( \varphi \) to be enforced are safety properties such as invariants (allowing executions to visit some subset of global states), transition invariants (allowing in a given global state only a subset of its outgoing transitions), or more general temporal logic properties. However, the controller forbidding all transitions trivially refines any controller defined by an arbitrary safety property. Therefore, controllers also make progress requirements.

As a result, a property \( \varphi \) representing a controller is generally given in the form \( \varphi_S \cap \varphi_{pr} \) where \( \varphi_S \) is a safety property and \( \varphi_{pr} \) defines a progress property.

Typical progress properties \( \varphi_{pr} \) are for example absence of new deadlocks. New deadlocks are deadlock states that are not already specified in the system as final states. Fairness is also a progress property as it defines progress of all individual components of a system. Maximal progress with respect to a property \( \varphi \), is an example of progress property which states that the controller may forbid a transition only if it leads to an unavoidable violation of \( \varphi \).

Providing an executable controller refining a controller defined by both progress and an arbitrary safety property is of high complexity, and it does not always exist. This is due to the presence of so-called uncontrollable transitions or choices in the behavior of the system \( K_{Sys} \) to be controlled. It is possible that allowing in state \( q \) some transitions, may lead to a violation of \( \varphi \) many steps later if beyond that transition, the transitions or choices in \( K_{Sys} \) cannot be controlled. Thus, in this work, we are interested in controllers represented by a particular class of properties:

- Among safety properties, we consider only those which are not sensitive to the fact that some transitions cannot be controlled. Such properties only use their current state to forbid some transitions (based on their label). In other words, they are memoryless. Note that interaction models and priority orders as presented in the previous section define such properties.

- We consider deadlock freedom. This progress property is meaningful for example if non-determinism in the specification represents possible design choices which can be exploited to optimize an actual implementation. In Section 6.2, we propose a step towards “correctness by construction” and exploit this freedom in an algorithm generating (whenever possible) a priority order avoiding existing deadlocks in a given specification.

- We also consider maximal progress. This progress property is typical for the case where non-
determinism represents choices of the environment, in which case one would like to constrain the environment only as much as necessary to guarantee safety. In Chapter 7, we propose a message-based protocol for controlling a system by an interaction model and a priority order in a distributed setting. Here also, the idea is to achieve “correctness by construction”, this time by proposing an algorithm generating a distributed implementation from a set of local components and a global safety and progress constraint.

We now formalize the notions needed for discussing controllers and controlled systems which we have used informally so far. As already mentioned, we do not formally distinguish a controller and a property to be enforced. Properties are represented semantically, by sets of extended prefixes with certain closure properties, an extended prefix being a pair consisting of a finite history (state-action sequence) and the set of actions possible after that history. Satisfaction (of a property by a component) and refinement (between properties) are defined by comparing sets of extended prefixes.

**Definition 6.1.9 (Property)** Let $P$ be an interface and $Q$ a set of states.

A property $\varphi$ on $P$ and $Q$ is a set of extended prefixes that is prefix- and suffix-closed, i.e.:

- if $(\sigma, A) \in \varphi$ then $\forall a \in A, \exists q, \exists A' : (\sigma \cdot a \cdot q, A') \in \varphi$

- if $\exists a \in A, \exists q, \exists A' : (\sigma \cdot a \cdot q, A') \in \varphi$ then $(\sigma, A) \in \varphi$

**Definition 6.1.10 (Systems as properties, Conformance, Refinement)**

- A component $K$ on $P$ with state set $Q$ defines a property $\text{Acc}_K$ as the set of extended prefixes $(\sigma, A)$ such that $\sigma$ is a prefix of an execution of $K$, and $A$ is a subset of the set of outgoing labels from $q_{\sigma}$ with the constraint that $A$ may only be $\emptyset$ if $q_{\sigma}$ is a final state (in $F$) or a deadlock.

- A system $\text{Sys}$ defined a component $K_{\text{Sys}}$ on $P$ with state set $Q$ defines a property $\text{Acc}_{\text{Sys}}$ as the set of extended prefixes $(\sigma, A)$ such that $\sigma$ is a prefix of an execution of $K_{\text{Sys}}$, and $A$ is the set of outgoing labels from $q_{\sigma}$ with the constraint that $A$ may only be $\emptyset$ if $q_{\sigma}$ is a deadlock.

- $K_{\text{Sys}}$ conforms $\varphi$ (denoted $K_{\text{Sys}} \preceq \varphi$) iff $\text{Acc}_K \subseteq \varphi$. Moreover, $\varphi$ refines (also called implements) $\varphi'$ iff $\varphi \subseteq \varphi'$.

Note that the notion of conformance defined here is consistent with the notion of conformance defined in the previous chapters, as it defines a relation between closed systems.

Note the difference between $\text{Acc}_K$ of a component $K$ and $\text{Acc}_{\text{Sys}}$ of a system $\text{Sys}$ is due to the fact that the first represents an open system and the second a closed one. Thus $\text{Acc}_K$ represents what is usually called an acceptance semantics of a component $K$ in any context. $\text{Acc}_{\text{Sys}}$ represents an acceptance semantics of $\text{Sys}$ in an empty context, that is the context used to define conformance in Section 4.4. Note $\text{Acc}_K$ is indeed a property, as it is prefix- and suffix-closed.

**Definition 6.1.11 (Memoryless property, Safety property, Progress property)**

- $\varphi$ is memoryless (history independent) iff $(\sigma, A) \in \varphi$ implies that $(\sigma', A) \in \varphi$ for any $\sigma'$ such that $q_{\sigma'} = q_{\sigma}$, that is, $A$ does only depend on $q_{\sigma}$, not on the history expressed by $\sigma$. 

109
• \( \varphi \) is a safety property iff for a given prefix \( \sigma \), the corresponding set of acceptance sets is subset closed, that is, \( (\sigma, \Lambda) \in \varphi \) implies that for all \( \Lambda' \) included in \( \Lambda \), \( (\sigma, \Lambda') \in \varphi \).

• Symmetrically, \( \varphi \) is a progress property iff it is superset closed, that is, it defines which executions a component satisfying it should have at least.

Note that the property \( \text{Acc}_K \) defined by a component \( K \) is a memoryless property on \( \mathcal{P} \) and \( Q_K \). It is almost a safety property, as it specifies an overapproximation of the behavior of \( K \). It is not fully a safety property though, because only final states and deadlocks may have an empty acceptance set. Thus, for a given prefix \( \sigma \), the set of acceptance sets is not subset closed. Note that this definition is consistent with the set of executions of \( K \) defined in Section 6.1. As a result, \( K \) represents a very weak progress property, and we need an additional controller if we want to enforce stronger ones. Hence the following properties.

• Deadlock freedom is defined by a property \( \varphi \) defined by: iff \( (\sigma, \Lambda) \in \varphi \) implies that \( \Lambda \neq \emptyset \).

• Maximal progress, denoted \( \varphi_{mp} \), is defined with respect to a given component \( K \) and safety constraint \( \varphi_S \). It consists in determinizing \( \text{Acc}_K \cap \varphi_S \) by choosing for each prefix \( \sigma \) only the acceptance sets \( \Lambda \) which are maximal with respect to set inclusion.

Now, interestingly, the notions of interaction model and priority order that have already been introduced also define properties.

**Definition 6.1.12 (Property of an interaction model)** An interaction model \( \mathcal{I} \), with the set of interactions \( \mathcal{I}(\mathcal{I}) \), defines the memoryless property \( \varphi_{\mathcal{I}} \) consisting of all pairs \( (\sigma, \Lambda) \) where \( \Lambda \subseteq \mathcal{I}(\mathcal{I}) \): in any state, the corresponding acceptance sets contain only interactions allowed by \( \mathcal{I} \).

**Definition 6.1.13 (Property of a priority order)** A priority order \( \prec \) defines the memoryless property \( \varphi_{\prec} \) consisting of all pairs \( (\sigma, \Lambda) \) such that for all \( a, b \in \Lambda \), \( a \) and \( b \) are not related by \( \prec \). That is, in any state, any enabled transition disables all those with lower priority.

Now, we still have to define what it means to control a system by a property.

**Definition 6.1.14 (Controlled system)** Let \( K_{\text{Sys}} \) be a component on \( \mathcal{P} \). Then:

- Any property \( \varphi \) on \( \mathcal{P} \) and the set of states \( Q_{\text{Sys}} \) may be used as a controller for \( K_{\text{Sys}} \).

- The controlled system defined by \( K_{\text{Sys}} \) and \( \varphi \), denoted \( (K_{\text{Sys}}, \varphi) \), is the property \( \text{Acc}_{\text{Sys}} \cap \varphi \).

- An implementation of \( (K_{\text{Sys}}, \varphi) \) is a system \( K'_{\text{Sys}} \) conforming to \( \text{Acc}_{\text{Sys}} \cap \varphi \).

Similarly, we can define what it means to control a component by a property, using the acceptance semantics of a component that is \( \text{Acc}_K \) as define above.

Note that if \( \varphi \) is memoryless, then \( \text{Acc}_K \cap \varphi \) is also memoryless. Note that we are interested here only in memoryless controllers. And we can now establish very easily the fact that an interaction model \( \mathcal{I}_{\prec} \) as defined in Section 2.1.6 defines a memoryless controller for a system of the form \( ||\{K_i\}|| \).
Lemma 6.1.15 (Interaction models and priority orders as memoryless controllers) Consider a system $Sys$ of the form $(\{K_i\}, I)$ with $I$ an interaction model and $< \subseteq$ a priority order defining, according to Definition 2.1.16, a component $K_{Sys} = I < \{K_i\}$. Then $I$ and $<$ define a memoryless controller for the component $\|\{K_i\}$ in the sense that $I < \{K_i\}$ is an implementation of the controlled component $(\|\{K_i\}, \varphi_I \cap \varphi_<)$.\\

**Proof** Interaction models and priority orders define memoryless properties as they define the set of allowed interactions independently of the history, in fact, they do even not depend on the current state. The combined effect of an interaction model and a priority order is represented by the intersection of these properties, which is memoryless as well. The executions of $I < \{K_i\}$ are also executions of $\|\{K_i\}$. Moreover, after a prefix $\sigma$, the corresponding acceptance sets in $Acc_I < \{K_i\}$ are exactly the same as those in $Acc_\|\{K_i\} \cap \varphi_I \cap \varphi_<$. Hence: $Acc_I < \{K_i\} \subseteq Acc_\|\{K_i\} \cap \varphi_I \cap \varphi_<$. That is, $I < \{K_i\} \models Acc_\|\{K_i\} \cap \varphi_I \cap \varphi_<$, which means by definition that $I < \{K_i\}$ implements $(\|\{K_i\}, \varphi_I \cap \varphi_<)$. \hfill $\square$\\

### 6.2 Synthesis of priorities for avoiding deadlocks\\

In this section, we propose a small algorithm allowing to enforce the property of deadlock freedom using the property defined by priorities. More precisely, we propose an algorithm that generates for a system which has some deadlocks a controller in the form of a priority order $<$ in order to avoid these deadlocks. Later, in Chapter 7, we propose a generic algorithm that generates a distributed implementation for a distributed system controlled by an interaction model and a priority order.\\

In terms of the notations introduced earlier, the problem we want to solve is the following: given a system of the form $Sys = (\{K_i\}, I_0)$ which has deadlocks, determine a priority order $<$ such that $(\{K_i\}, I_<$) is deadlock-free if such a $<$ exists. Otherwise, report that no appropriate priority order $<$ exists.\\

Given a system $K_{Sys}$ and a set $Tr$ of transitions to be avoided, the algorithm computes, if possible, a set of priority rules which make these deadlocks unreachable by either inhibiting the transitions in $Tr$ or making them unreachable by inhibiting earlier transitions.\\

Before presenting the full algorithm, we illustrate how it works — or fails — on simple examples.
6.2. SYNTHESIS OF PRIORITIES FOR AVOIDING DEADLOCKS

A simple system controllable with priorities Consider two components $K_1$ and $K_2$ defined on \{a, b, c\} and depicted in Figure 6.1 and which are composed by the already mentioned interaction model $I$ requiring that transitions with the same name must interact. The obtained system has a deadlock which is reached if initially the two components choose $b$, as after that, $K_1$ expects an interaction on $c$ while $K_2$ has already terminated.

This deadlock is avoided if in the initial state 00, $a$ is chosen instead. We can achieve this by choosing the priority order defined by \{b < a\}. It forbids the $b$-interaction in the initial state where also an $a$-interaction is enabled, and it allows a $b$-interaction after the execution of the $a$-interaction, (in state 11), where it is the only enabled interaction.

A simple system not controllable with priorities Figure 6.2 shows a slightly different system defined by two components $K_1'$ and $K_2'$ composed using the same convention as before. For this system, there exists no priority order that avoids deadlocks. We sketch below how this is detected by our algorithm:

1. The composed component $K' = I\{K_1', K_2'\}$ is (partly) computed and transitions leading to deadlocks are marked as error transitions, as shown in Figure 6.2.

2. In the initial state (1), $b$ must be preferred to $a$ in order to prevent a deadlock, and we conclude that any priority order making $K'$ deadlock free includes the rule $a < b$.

3. In state 2, there are two possibilities to forbid the $b$-transition leading to a deadlock: either $b$ has lower priority than $a$ or state 2 is made unreachable. The first option is impossible as it would mean that $a < b$ and $b < a$ which violates the requirement that a priority is a strict partial order. The second option implies that the transition from 1 to 2 labeled by $b$ should be inhibited by a transition with higher priority enabled in the initial state which leads to exactly the same contradiction.

We conclude that no priority order can control the given example to guarantee deadlock freedom. Note that dynamic priorities can deal with this example, as it is sufficient to define as priorities, $a < b$ in state 1 and then $b < a$ in state 2.

Figure 6.1: An example where reducing non-determinism eliminates a deadlock.
Note that we only consider static priorities which do not depend on the state of the system. Obviously, dynamic priorities defined in [BBBS08, GS04] — which are allowed to be different in different states of the system — are more powerful in the sense that they allow to always eliminate all deadlocks. But this can be generally done at the price of drastically reducing — or even eliminating — concurrency. Indeed, dealing with dynamic priorities requires generally a precise knowledge about the current global state in order to decide whether a given transition has highest priority or not and “precise knowledge about the global state” can generally be only achieved by adding more communications, that is, reducing the degree of concurrency.

The general algorithm is given below, and it is quite easy to see how it could be extended for generating dynamic priorities — if needed.

The dining philosophers We consider a variant of the dining philosophers problem inspired from [Pad]. Philosophers are components providing thoughts if they have got two forks. These forks represent shared resources given in form of a unique component providing forks and expecting to get thoughts in return. Figure 8.6 shows a configuration with two philosophers and a resource with two forks. A deadlock arises if both philosophers have each one a fork and wait forever for a second one.

This deadlock can be avoided by always giving the highest priority to the request closest to completion. This is a classical method for managing resources. The priority order that is needed here is \( \{fork_1^\alpha < fork_2^\beta, \ fork_1^\beta < fork_2^\alpha\} \). For readability reasons, we simplify in Figure 8.6 the names of the interactions of the composed behavior in a straightforward manner. \( fork_{1,2}^{\alpha,\beta} \) in the behavior of the component Forks corresponds to the interactions \( \{fork_1^\alpha, \ fork_2^\alpha, \ fork_1^\beta, \ fork_2^\beta\} \) of the two philosophers.

Note that in this example priorities are local as there exists a unique pool of forks involved in all interactions. If instead of a resource pool, we define a set of resource components, each one administrating one fork that is shared by two neighbors, then the corresponding priorities, which are still useful, are global.
6.2. SYNTHESIS OF PRIORITIES FOR AVOIDING DEADLOCKS

Algorithm 1 Priority4Tr(K = (Q, q0, Σ, δ, F), Tr, π): priorityOrder or ⊥

if Tr = ∅ then
    return π // there are no error transitions, so π is a solution
else
    Pot ←− ∅
    Initialize(K, Tr, π, Pot) // initialize sets π, Pot; simplify K, Tr accordingly
    if Tr = ∅ then
        return π
    else if Pot = ∅ then
        return ⊥ // error transitions cannot be avoided, so there is no solution
    end if
    O ←− PotentialOrders(Tr, π, Pot) // calculate the set of potential priority orders
    return FindOrRefine(K, Tr, Pot, O) // find O of O being a solution or refine it
end if

Figure 6.3: A solution to the dining philosophers problem.

Algorithm for inferring priorities For simplicity, we suppose that the following are precalculated before calling the main algorithm which is Priority4Tr:

1. the global behavior of the component or system of interest, that is K if it is a component and K_{Sys} if it is a system Sys.

2. the set Tr of error transitions, namely those leading into a deadlock state.

The set π is initialized as the empty set. Then, at any step of the algorithm, Tr holds the set of error transitions which are not yet forbidden by some priority in π. π contains at any time a valid priority order < forbidding the error transitions not inhibited by forbidding transitions in Tr, and at successful termination it contains a desired solution.
Algorithm 2 Initialize($K, Tr, \pi, Pot$)

for all $t = (s, a, s') \in Tr$ do
  if $q^0 \rightarrow s$ and $s \rightarrow b \in \delta$ implies $(b < a \in \pi \lor b = a)$ then
    Pot $\leftarrow \emptyset$ break
  else if $q^0 \rightarrow s$ and $\exists b \in \Sigma$ s.t. \{\begin{align*} b \rightarrow & \delta \implies (b < a \in \pi \lor b = a) \end{align*}\} then
    if $b < a \in \pi$ then
      Pot $\leftarrow \emptyset$ break
    else if $b < a \in \pi$ then
      add($\pi, a < b$) // add $a < b$ to Prio and normalize by adding induced priorities
      $Tr' \leftarrow \{t\} \cup \{(q, a, q') \in \delta \mid q \rightarrow b\}$ // remove from $\delta$ transitions inhibited by $a < b$
      simplify($Q, \delta$) // simplify $\delta$ and $Q$ by removing unreachable transitions and states
      $Tr \leftarrow Tr \cap \delta$ // simplify $Tr$ in accordance with the new $\delta$
    end if
  end if
add($\pi, a < b$) // add $a < b$ to Prio and normalize by adding induced priorities
$Tr' \leftarrow \{t\} \cup \{(q, a, q') \in \delta \mid q \rightarrow b\}$ // remove from $\delta$ transitions inhibited by $a < b$

simplify($Q, \delta$) // simplify $\delta$ and $Q$
$Tr \leftarrow (Tr \cup \{(q, l, s) \in \delta\}) \cap \delta$ // simplify $Tr$ according to $\delta$

end for

The Priority4Tr algorithm successively calls the algorithms Initialize, PotentialOrders and FindOrRefine, which play the following roles:

1. Initialize computes the priority rules Prio that are necessary to avoid all deadlocks from any reachable state of $K$. If Prio contains a contradiction the overall algorithm terminates with failure.

2. PotentialOrders computes a set of alternative priority rules Pot which may be used to control the execution,

3. FindOrRefine picks one such priority order and explores it. If it fails, then another alternative in Pot is explored until success or failure — if none of them works.

Note that a priority order is represented explicitly by a set of rules $a < b$ where for simplicity, also those rules which can be deduced by transitivity are explicitly represented. We also do not detail here the basic straightforward algorithms for manipulating priorities (adding or deleting rules, building the union of priority order, . . . ) in this normal form and maintaining them in this form.

Moreover, we use the following notations:

- For any transition $t \in Tr$, $Pot_t$ represents the set of potential priority rules that can inhibit $t$. $Pot\setminus\{r\}$ denotes that rule $r$ is removed from $pot$ and thus from all sets $Pot_t$. 

115
6.2. SYNTHESIS OF PRIORITIES FOR AVOIDING DEADLOCKS

Algorithm 3 PotentialOrders($Tr$, $\pi$, $Pot$): Set of priorityOrders

choose $t \in Tr$

$O \leftarrow \{r \in Pot_t \mid \text{add(Prio, } r) \text{ is defined}\}$ \quad // some priorities in $Pot_t$ may contradict $\pi$

for all $r \in O$ do

$Tr^r_{ok} \leftarrow \{t' \mid r \in Pot_t\}$ \quad // transitions in $Tr$ inhibited by $r$

end for

Unreachable$_t \leftarrow$ PotentialOrders($Tr\{t\}$, $\pi$, $Pot$) \quad // suppose $t$ needs not be inhibited

Inhibited$_t \leftarrow \bigcup_{r \in O} \text{PotentialOrders}($$Tr\{t\} \cup Tr^r_{ok}$, $\text{add(} \pi, r, Pot\{r\}$$)$) \quad // $t$ is inhibited by $r$

return Unreachable$_t \cup$ Inhibited$_t$

- $q^0 \rightarrow^* s$ denotes that $s$ is reachable from $q^0$ by a (possibly empty) sequence of transitions for which there exists no alternative, that is, transitions of the form $(s_1, \alpha, s_2)$ such that $\delta$ has no other transition with $s_1$ as origin.

Theorem 6.2.1 Given a component $K$, the algorithm terminates, and at termination, $\pi$ defines a priority order $<$ such that $\langle < \rangle (K)$ is deadlock free or the algorithm terminates with failure if no such priority order exists.

Proof The correctness of the algorithm is guaranteed by the following facts:

1. if $Prio$ does not define strict partial order, the algorithm terminates unsuccessfully, and $Prio$ contains only rules which must be used to inhibit some error transitions.

2. at any point of time, $Prio$ together with avoiding $Tr$ guarantees avoidance of deadlocks because initially avoiding $Tr$ obviously guarantees deadlock freedom, and a transition $t$ is only eliminated from $Tr$ if there is a rule in $Prio$ forbidding it or if $t$ is replaced by some transition (set) leading to the start state of $t$. On termination, $Tr$ is empty, thus $Prio$ — which is a priority order — is able to prevent all deadlocks.

3. The fact that the algorithm terminates unsuccessfully implies that indeed there is no appropriate priority order is guaranteed by the fact that the algorithm systematically explores transition sets allowing to block the access to a deadlock state without introducing a new deadlock, and such a set is rejected only if avoiding requires contradictory priorities.

4. the algorithm does indeed terminate as (1) the main algorithm is called each time with a set $Tr$ of error transitions that decreases or contains transitions closer to the initial state and (2) the other algorithms explore a finite set of alternative orders, pick one of them or abandon if none of them works.

For readability, the parameters of Initialize are call-by-reference, while those of Priority4Tr, PotentialOrders and FindOrRefine are call-by-value.
CHAPTER 6. CONTROLLERS FOR SYSTEMS WITH PRIORITIES

Algorithm 4 FindOrRefine($K, Tr, Pot, O$)

if $\exists O \in O$ s.t. $\forall t, Pot_t \cap O \neq \emptyset$ then
    return $O$  // $O$ inhibits all error transitions, thus $O$ is a solution
else
    while $O \neq \emptyset$ do
        choose $O \in O$
        $O \leftarrow O \setminus \{O\}$
        $Tr_{bad} \leftarrow \{t | Pot_t \cap O = \emptyset\}$  // transitions not inhibited by $O$
        $\delta \leftarrow \delta \setminus (Tr \setminus Tr_{bad})$  // remove inhibited transitions from $\delta$
        simplify($Q, \delta$)  // simplify $\delta$ and $Q$
        $Tr \leftarrow (Tr \cup pre(Tr_{bad}))$  // add predecessors of transitions not inhibited to $Tr$
        $Tr \leftarrow Tr \cap \delta$  // simplify $Tr$ according to $\delta$
        $Result \leftarrow$ Priority4Tr($B, Tr, O$)
        if $Result \neq \bot$ then
            return $Result$
        end if
    end while
    return $\bot$
end if
6.2. SYNTHESIS OF PRIORITIES FOR AVOIDING DEADLOCKS
Chapter 7

Distributed Controllers for Systems with Priorities

The practical motivation of the work presented in this part of the thesis, is to provide a distributed implementation of systems controlled by properties of their interaction models. In particular, our purpose is enforcing the properties of the interaction models with priorities which, as already described, define a memoryless controller. The obtained controlled system correspond actually to a simple BIP system with priorities. In this chapter, we propose such a distributed implementation by synthesizing a distributed memoryless controller.

Distributed characterization of a system consists of a set of components communicating through message passing where we suppose that the underlying communication platform ensures reliable and order-preserving transmission of messages.

We have adopted previously a very abstract view of a controller as a property. Thus, in this chapter, we first define, in Section 7.1 what it means to distribute a controller, and what a correct implementation should be. Then, in Section 7.2 we propose a protocol allowing to implement such a distributed controller based on message-passing.

7.1 Distributing systems and controllers

We suppose from now on that we work with systems \( Sys = (\{K_i\}, \langle \rangle) \) in which a port is part of at most one interaction. Thus, local properties (defined on \( P_i \) and \( Q_i \)) and global properties (on the interface \( P \) and the set \( Q \) of states of \( K_{Sys} \)) share their interfaces. This can be done without loss of generality, and does drastically simplify the presentation.

For a system \( Sys \) of the form \( (\{K_i\}, \langle \rangle) \) on which a global property \( \varphi \), typically \( \varphi_I \cap \varphi_< \cap \varphi_{mp} \), must be enforced, we are now interested in defining a distributed controller. Such a controller consists of a set of local controllers i.e., local properties \( \varphi^i \), taking decisions about the next transitions that can be executed. Those decisions are based on the local information of the corresponding component \( K_i \) in such a way that the union of local decisions is a decision allowed by the global property \( \varphi \), which means that we are yielding a disjunctive controller.
In Section 7.2, we implement such a distributed controller for the specific case where the global property to be enforced is of the form $\phi_I \cap \phi_\prec \cap \phi_{mp}$, that is, for systems in which a set of components are constrained by an interaction model and a priority order, and for which we want a controller achieving maximal progress. Note that:

- $\phi_I \cap \phi_\prec$ is a safety property that it is union closed: indeed, for every state $q$ the set of acceptance sets has a maximal element $(\pi, A)$ where $A$ is the set of interactions that are enabled with maximal priority in $q$. Remember that $\phi_I$ and $\phi_\prec$ are memoryless, thus $\phi_I \cap \phi_\prec$ is memoryless too.

- Maximal progress, i.e. property $\phi_{mp}$, consists in allowing for each prefix only the set of acceptance sets which are maximal with respect to set inclusion. Thus, in our special case, there exists in every state exactly one element of the form $(\pi, A)$, namely the maximal element mentioned in the previous item meaning that every allowed interaction is also required.

More generally, the properties we want to distribute are characterized for each prefix by (1) a set of allowed interactions and (2) a set of required interactions, the two sets are identical for deterministic specifications when maximal progress is required. This means that if local properties are correct in the sense that they allow only globally enabled interactions, and complete in the sense that for every interaction allowed by $\varphi$ at least one component allows it, then the union of the locally allowed interactions is exactly the set of globally allowed interactions. On the other hand, if each of the local properties ensures maximal progress, then the union of the locally required interactions is exactly the set of globally required interactions. Note that other progress requirements, in particular fairness and deadlock freedom can be also achieved in that way.

Moreover, note that the properties that we consider are memoryless. We will see in Section 7.2 that this greatly simplifies the construction of the local controllers.

We now formally define the notion of distributed controller as a composition of local controllers.

**Definition 7.1.1 (Composition of local properties)** Consider a set of properties $\{\varphi^i\}$ on $\{Q_i\}$ and $\{P_i\}$ and the interaction model $I_\parallel$ defining the set of all possible interactions in $P = \bigcup_i P_i$. We define the composition of the $\varphi^i$, denoted $\oplus_i \{\varphi^i\}$, as the set of extended prefixes $\{(\pi, \bigcup_i A_i) \mid \forall i, (\pi_i, A_i) \in \varphi^i\}$ where $\pi$ is an alternating sequence of states in $Q = \prod_i Q_i$ and actions in $I_\parallel$, and $\pi_i$ can be obtained from $\pi$ by replacing states in $\pi$ by their $i$-th component.

The property $\oplus_i \{\varphi^i\}$ defined by the set of properties $\{\varphi^i\}$ can be seen as a controller and it implements a controller $\varphi$ when $\oplus_i \{\varphi^i\} \subseteq \varphi$. Before we give a formal definition of distributed controller, we first extend the composition of properties to properties on extended local states.

This means that we must either extend local properties also to the global interface $P$ or we must restrict the set of properties which we want to distribute. We make the second choice as we are mainly interested in controllers which are memoryless and where $\varphi$ is of the form $\varphi_S \cap \varphi_{pr}$ for $\varphi_S$ a safety property and $\varphi_{pr}$ a progress property. We have seen that safety properties are subset and union closed which means that indeed in a given situation, the union of locally identified next interaction is a valid global set of next interactions.
CHAPTER 7. DISTRIBUTED CONTROLLERS FOR SYSTEMS WITH PRIORITIES

In general, none of the individual components is able to decide locally whether in a state an interaction $a$ is enabled. Some minimal knowledge about the current global state is then necessary to take a decision locally. In [BBPS, GPQ10], it is proposed to use statically computed knowledge derived from the set of reachable states. A more classical solution — which we adopt here — consists in letting local controllers compute dynamically the knowledge required to take a correct decision locally. On a platform where communication is by message passing, this is achieved via messages as well, and this requires a minimal degree of stability of the information transmitted by messages. Here, we also follow this approach.

As a consequence, we consider properties $\varphi^i$ which are not defined on the set of local states $Q_i$ but on a set of extended local states $Q_i^{\text{ext}}$ of the form $Q_i \times Q_i^{\text{kn}}$ where in a state $(q_i, q_i^{\text{kn}})$, $q_i^{\text{kn}}$ represents a set of global states which are compatible with $q_i$, the intention being that in $(q_i, q_i^{\text{kn}})$, $\varphi^i$ knows that the current global state is in $q_i^{\text{kn}}$, which helps deciding for a sufficient subset of interactions whether they are enabled. Concretely, we do not formalize here how these $\varphi^i$ are built, only how we check that they implement indeed a correct distributed controller. We can now adapt composition of local properties to these properties defined on extended local states. This only requires, in addition to the previous definition, to check that each global state $q$ of an execution is compatible with the extended local state $(q_i, q_i^{\text{kn}})$ of the corresponding executions, i.e., $q_i$ is the $i$-th component of $q$ and $q \in q_i^{\text{kn}}$.

**Definition 7.1.2 (Composition of local properties defined on extended local states)** $\oplus_i \{\varphi^i\}$ for a set of properties on extended local states is the set of extended prefixes $\{ (\pi, \bigcup_i A_i) | \forall i, (\pi_i, A_i) \in \varphi^i \}$ where $\pi$ is an alternating sequence of states in $Q = \prod_i Q_i$ and actions in $I_\parallel$, and $\pi_i$ is obtained from $\pi$ by replacing each state $q$ in $\pi$ by a pair consisting of its $i$-th component and a set of global states including $q$.

**Definition 7.1.3 (Distributed controller)** Let be $Sys = (\{K_i\}, \parallel)$ and $\varphi$ as above. A distributed controller for $(Sys, \varphi)$ is a set of properties $\{\varphi^i\}$ such that $\oplus_i \{\text{Acc}_{K_i} \cap \varphi^i\} \subseteq \text{Acc}_{K_{\text{Sys}}} \cap \varphi$, where $K_{\text{Sys}} = \parallel \{K_i\}$.

In the case that we consider the property of maximal progress, $\text{Acc}_{K_{\text{Sys}}} \cap \varphi$ contains for every prefix exactly one set $A$, the set of globally enabled interactions, and therefore, inclusion means indeed equality. Note also that such a distributed controller always exists for memoryless properties as those considered. Indeed, a set of local controllers defined on the global state space can clearly decide for every interaction whether it is enabled or not.

### 7.1.1 Concurrency and confusion

We aim at distributed executions, thus interactions which are independent, denoted *concurrent* in Definition 7.1.4 below, can be executed concurrently, however, interactions which are not concurrent, that is in conflict, cannot. Throughout this section, consider a system $Sys = (\{K_i\}, I_\subseteq)$. We can now define the usual notions of *concurrency* and *conflict* of interactions, where in a distributed setting we want to allow the independent execution of concurrent interactions, so as to avoid global sequencing. We distinguish explicitly between the usual notion of conflict which we call structural conflict, and a conflict due to priorities.
Definition 7.1.4 (Concurrent interactions, Conflicting interactions) Let $a, b$ be interactions of $P$ and $q \in Q$ a global state in which $a$ and $b$ are globally ready.

- $a$ and $b$ are called concurrent in $q$ iff $\text{ind}_a^q \cap \text{ind}_b^q = \emptyset$ (see Definition 6.1.7). That is, when $a$ is executed then $b$ is still globally ready afterwards, and vice versa, and if executed, both interleavings lead to the same global state.

- $a$ and $b$ are called in structural conflict in $q$ iff they are not concurrent in $q$, that is $a$ and $b$ are alternatives disabling each other.

- $a$ and $b$ are in prioritized conflict in $q$ iff $a$ and $b$ are concurrent in $q$ but $a < b$ or $b < a$ holds.

Note that in case of prioritized conflict, it is known which interaction cannot be executed, whereas in case of structural conflict, the situation is symmetric. We use the notations $\text{Concurrent}_q(a)$, $\text{Conflict}_q(a)$, $\text{PrioConflict}_q(a)$ to denote the set of interactions that in state $q$ are concurrent to $a$, respectively in structural or prioritized conflict to $a$.

Our local controllers are defined on extended local states, and an implementation of the distributed controller has to collect the required knowledge for being able to take a decision. However, collecting this information may take time and some concurrent transition $t$ may be executed concurrently. Thus, the notion of concurrency chosen must provide sufficient stability of the collected information for it to be useful: indeed, after receiving this information, the local controller does not know whether $t$ has taken place, and thus its extended local state contains both source states and target states of $t$. Thus, the usual notion of concurrency is not sufficient to detect such situations which are called Confusion.

Confusion is a situation where concurrency and conflict are mixed. More precisely, confusion arises in a state where two interactions $a_1$ and $a_2$ may fire concurrently, but firing one modifies the set of interactions in conflict with the other. It is a situation occurring in distributed systems [Bol07, Bol05]. Typically, detecting those situations is important for designing correct algorithms for partial order reduction. In presence of priorities, confusion situations may compromise correctness of a distributed implementation of a specification. We first define some preliminary notions which allow us to characterize different situations of confusion.

Figure 7.1 illustrates a situation of structural conflict: interactions $a_1$ and $a_3$ are in structural conflict as they both involve component $K_1$ (respectively $K_2$). Figure 7.2 illustrates a prioritized conflict of $a_1$ with $a_3$ as these interactions are concurrent but $a_1 < a_3$ holds.

A symmetric (left) and an asymmetric (right) situation of confusion are shown in Figure 7.1: in the symmetric case, interactions $a_1$ and $a_2$ of $K_1$ and $K'_1$ are concurrent but are both in conflict with $a_3$ and the execution of $a_1$ (resp. $a_2$) changes the set of interactions in conflict with $a_2$ (resp. $a_1$). In the asymmetric case, the interactions $a_1$ and $a_2$ of $K_2$ and $K'_2$ are concurrent but $a_1$ will enter in conflict with $a_3$ if $a_2$ fires before $a_1$.

Definition 7.1.5 (Confusion) Let $a_1$ and $a_2$ be interactions, and $q$ a global state of $Sys$. We suppose that $a_1$ and $a_2$ are concurrent — and thus globally ready — in $q$.

- $a_1$ is in structural confusion with $a_2$ iff $\exists q' \in Q$, $q \xrightarrow{a_2} q'$ implies $\text{Conflict}_{q'}(a_1) \neq \text{Conflict}_{q'}(a_1)$
• $a_1$ is in prioritized confusion with $a_2$ iff $\exists q' \in Q, q \xrightarrow{a_2} q'$ implies $\text{PrioConflict}_{q'}(a_1) \neq \text{PrioConflict}_q(a_1)$

Figure 7.1: Symmetric and asymmetric confusion.

Figure 7.2 (left) illustrates a situation of prioritized confusion: $a_1$ and $a_2$ of components $K_1$ and $K'_1$ are concurrent, however firing $a_2$ enables $a_3$ which has higher priority than $a_1$ which means that $a_1$ is no more enabled after the execution of $a_2$.

The classical notion of confusion is what we call structural confusion. Note that all situations of confusion are important for designing partial order reductions which are very important for making the verification of global properties of $\text{Sys} = (\{K_i\}, I_\prec)$ feasible. The reason is that eliminating arbitrarily one of the two interleavings of $a_1$ and $a_2$ may change the set of reachable states, and thus lead to different verification results.

For designing a distributed implementation of $\text{Sys}$, only the situation of Figure 7.2 where executing $a_2$ disables $a_1$ due to a new priority conflict, is problematic. The reason is that in this case $a_1$ and $a_2$ are not really “concurrent”, whereas in all other cases, it does still hold that $a_1$ and $a_2$ can be executed in any order and both orders lead to the same global state.

Figure 7.2: Prioritized confusion.

In Section 7.2 we propose a distributed implementations of systems $\text{Sys} = (\{K_i\}, I_\prec)$ in which concurrent interactions are executed independently, based on the notion of concurrency of definition 7.1.4. This means that our algorithm does not support systems $\text{Sys}$ with such prioritized conflict situations.
In order to deal with this kind of confusion, we could use a more appropriate notion of concurrency which however would lead to inefficient implementations. We rather propose to eliminate such confusions statically by adding a priority $a_1 < a_3$ or $a_2 < a_1$ (see Figure 7.2(right)) such that either $a_1$ and $a_2$ are not anymore considered concurrent or at least it is guaranteed that executing concurrent interactions does not introduce priority conflicts which destroy this concurrency. Note that adding priorities between concurrent interactions in a given system does not add new deadlocks.

### 7.2 Implementation of a distributed controller as a protocol

Given a system $Sys = (\{K_i\}, I_>)$ defined by a set of components $\{K_i\}$, an interaction model $I$ and a priority order $<$ to be enforced, our goal is to define a distributed implementation for $Sys$. In this section, we define an algorithm which constructs such a distributed implementation by defining for each component a local controller $C_i$ ensuring a property $\varphi^i$ such that the joint execution of all components $K_i$ and their corresponding controllers guarantees the following:

1. all executions are executions of $Sys$, that is executions of $K_{Sys} = I_\prec \{K_i\}$
2. if $Sys$ is deadlock free, then no deadlock will ever occur

This means according to Definition 7.1.3 that the properties $\varphi^i$ of $C_i$ have to ensure the following:

$$\oplus_i \{Acc_{K_i} \cap \varphi^i\} \subseteq Acc_{\{K_i\}} \land \varphi_I \land \varphi_\prec \land \varphi_{mp}$$

As presented in Section 7.1, local controllers use messages to accumulate the knowledge required to extend their local state. For simplification, we do not formally represent knowledge as sets of global states. Instead, we use, as an abstract representation of this sets, a set of properties with respect to interactions, e.g. that interaction $a$ is globally possible etc. Absence of confusion ensures that this knowledge is sufficient. Based on this knowledge, for every enabled interaction, at least one local controller in any global state can determine that it is enabled. A local controller in a local state in which a transition labeled by $a$ is locally possible exchanges messages with the local controllers of other components involved in $a$ (to determine whether $a$ is globally possible) and with the local controllers of other components involved in an interaction $b$ with higher priority than $a$ to determine whether $a$ is enabled. In the latter case, it is sufficient to communicate with one component involved in $b$.

Here we represent local controllers $\phi_i$ as protocols interacting with the controlled component $K_i$ and with other controllers by exchanging messages.

Besides, we rely on $Sys$ to guarantee deadlock-freedom and fairness. That is, any distributed implementation of $Sys$ that does not introduce new deadlocks, not defined by $Sys$, is considered correct. Indeed, we suppose that, if needed, $Sys$ has been obtained using the algorithm of Section 6.2 that eliminates deadlocks. For this reason, we suppose in the following that $Sys$ has no deadlock.
CHAPTER 7. DISTRIBUTED CONTROLLERS FOR SYSTEMS WITH PRIORITIES

7.2.1 Description of the protocol

The system is supposed to have a fixed number of components, although it may be arbitrarily large. In order to simplify the presentation of the algorithm, we suppose here only binary interactions; an extension to arbitrary multi-party interactions is discussed at the end of this chapter. We also assume that the internal activities of components are terminating and that there exists no prioritized confusion (see Section 7.1.1), that is, the notion of concurrency used by the algorithm is correct. Without this last condition, we may observe global executions which are witnesses of a priority violation. As quite usually in distributed protocols [Bag89b, Bag89a, PCT04], we assume that the message passing mechanism ensures the following basic properties:

1. any message is received at its destination within a finite delay;

2. messages sent from location $L_1$ to $L_2$ are received in the order in which they have been sent;

3. there is no duplication nor spontaneous creation of messages.

For each interaction $a$ involved in at least one priority rule, one of the involved components $K_i$ place the role of the negotiator for $a$. If there exists at least one interaction with higher priority, the role of the negotiator is to check for the enabledness of $a$, and if there exists at least one interaction with lower priority, its role is to answer readiness requests. This notion of negotiator, is introduced to deal with priority, thus no similar notion exist in related algorithms. The choice of negotiators is discussed later.

We now describe the controllers of individual components which enforce correct executions, and in particular adherence to the global priority order. It is understood that what is called component $K_i$ is in fact controlled by a local controller $C_i$.

The controller associated with each component, maintains a set of data structures shared and maintained by the different subtasks of the controller: readySet (resp. enabledSet) contains the set of interactions which are known to be globally ready (resp. enabled) in the current local state $q$, and involved and possibleSet maintains the set of interactions that are locally ready. Note that possibleSet contains purely local information which can be calculated immediately when entering a new local state. The other two sets are calculated by a series of message exchanges, and the complete information is generally not calculated but as soon as an interaction is known to be enabled, its triggering will be initiated.

The general structure of the controller for each individual component $C_i$ is shown in Figure 7.3. The overall controller — and the component to be controlled — are represented as a set of concurrent activities (which we call threads, and which in our implementation are realized as Java threads) with a shared memory and shared message buffers.
Indeed, incoming messages are stored until one of the activities is ready to handle them. We use several FIFO buffers which are chosen such that the order amongst messages stored in different buffers does not influence the algorithm; in particular, they are used by concurrent threads. A buffer, which is read only by the thread Main, stores messages of the form \texttt{POSSIBLE}(a), \texttt{NOTPOSSIBLE}(a), \texttt{READY}(a), \texttt{NOTREADY}(a), and \texttt{REFUSE}(a). A second buffer stores messages of the form \texttt{COMMIT}(a), this buffer is read first by thread \texttt{WaitingForCommit}, then by \texttt{TryToCommit}. The role of each message is described in Table 7.1. Given that we are handling binary interactions, we do not explicit the recipient or the sender.

Note that the message \texttt{READY} denotes a question and a response at the same time. More precisely, when a negotiator of a sends \texttt{READY}(a), it informs another negotiator about the readiness of a. However, when a negotiator which is not the negotiator of a sends \texttt{READY}(a), then, it asks about its readiness.

\(C_i\) is either in state \texttt{READY} or in state \texttt{BUSY}. In state \texttt{BUSY}, \(C_i\) waits for \(K_i\) to execute the local action of the interaction that has been chosen. Incoming messages are stored and will not be handled until the controller moves to state \texttt{READY}. In state \texttt{READY}, the controller \(C_i\) looks for a next interaction to fire, proceeding as follows:
CHAPTER 7. DISTRIBUTED CONTROLLERS FOR SYSTEMS WITH PRIORITIES

Algorithm 5

Main

Require: toNegotiate = \{a ∈ possibleSet | negotiator(a) = K\}  

// The set of interactions for which K is a negotiator 

Input: set of interactions possibleSet ≠ ∅ 

Output: interaction i 

\n
\text{prioFree} = \{a ∈ \text{possibleSet} | \not\exists b, .b < a\} 

\text{waitingSet} ← − ∅ 

\text{checking global readiness:} 

\text{notReadySet ← − ∅} 

\text{readySet} ← − ∅ 

\text{lessPrio(a)} = \{b ∈ \text{readySet} | b < a\} 

\text{for all}\ a ∈ \text{possibleSet}\ \text{do} 

\quad \text{send}\ \text{POSSIBLE}(a) 

\text{end for} 

\text{create WaitingForCommit(possibleSet)} 

\text{if receive}\ \text{POSSIBLE}(a)\ \text{and}\ a ∈ \text{toNegotiate}\ \text{then} 

\text{create}\ \text{Negotiate(a)}\ \text{and}\ \text{readySet ← − readySet} \cup \{a\}\ \text{and} 

\text{for all}\ b ∈ \text{lessPrio(a)}\ \text{do} 

\quad \text{kill}\ \text{Negotiate(b)} 

\text{end for} 

\text{end if} 

\text{WHEN}\ \exists a, \text{ s.t. } \text{Negotiate(a)}= \text{OK}\ \text{or (receive}\ \text{POSSIBLE}(a)\ \text{and}\ a ∈ \text{prioFree})\ \text{call}\ \text{TryToCommit(a)}\ \text{and}\ \text{kill}\ \text{WaitingForCommit(possibleSet)}\ \text{and} \ \forall b ∈ \text{readySet}\ \text{kill}\ \text{Negotiate(b)} \ \text{if}\ \text{TryToCommit(a)}= \text{OK}\ \text{then} 

\text{return}\ a 

\text{else} 

\quad \text{goto checking global readiness} 

\text{end if} 

\text{if} \ \forall a ∈ \text{readySet}\ \text{Negotiate(a)}= \text{NOK}\ \text{then} 

\quad \text{goto checking global readiness} 

\text{end if} 

\text{if receive}\ \text{REFUSE}(b)\ \text{and}\ b ∈ \text{readySet}\ \text{then} 

\text{kill}\ \text{Negotiate(b)}\ \text{and}\ \text{readySet ← − readySet} \backslash \{b\} 

\text{end if} 

\text{if receive}\ \text{POSSIBLE}(b)\ \text{and}\ b ∈ \text{possibleSet} \backslash \{\text{toNegotiate} \cup \text{prioFree}\}\ \text{then} 

\text{send}\ \text{POSSIBLE}(b)\ \text{and}\ \text{readySet ← − readySet} \cup \{b\} 

\text{end if} 

\text{if receive}\ \text{NOTPOSSIBLE}(b)\ \text{and}\ b ∈ \text{possibleSet} \backslash \text{prioFree}\ \text{then} 

\text{notReadySet ← − notReadySet} \cup \{b\} 

\text{end if} 

\text{if receive}\ \text{POSSIBLE}(b)\ \text{and}\ b \not\in \text{possibleSet}\ \text{then} 

\quad \text{send}\ \text{NOTPOSSIBLE}(b) 

\text{end if} 

127
7.2. IMPLEMENTATION OF A DISTRIBUTED CONTROLLER AS A PROTOCOL

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSSIBLE</td>
<td>Offer an interaction (which is locally ready)</td>
</tr>
<tr>
<td>NOTPOSSIBLE</td>
<td>respond that an interaction is not locally ready</td>
</tr>
<tr>
<td>READY</td>
<td>Ask about the global readiness of an interaction</td>
</tr>
<tr>
<td>NOTREADY</td>
<td>Respond that an interaction is not globally ready</td>
</tr>
<tr>
<td>COMMIT</td>
<td>Commit to an interaction (cannot be undone by ( K_i ))</td>
</tr>
<tr>
<td>REFUSE</td>
<td>Inform that a component cannot commit to an interaction</td>
</tr>
</tbody>
</table>

Table 7.1: Messages used by the algorithm.

- The **Main** thread starts by checking its locally ready interactions (*possibleSet*) for interactions that are globally ready (see Algorithm 5). To check the global readiness of an interaction \( a \), messages of the form \( \text{POSSIBLE}(a) \) are exchanged, and peers in which \( a \) is currently not locally enabled respond with \( \text{NOTPOSSIBLE}(a) \) after which the requesting component “abandons” \( a \) until it changes state or the peer enters a state in which \( a \) is locally enabled and sends a \( \text{POSSIBLE}(a) \).

Whenever it is detected that an interaction \( a \) for which it plays the role of a negotiator is globally ready, a thread **Negotiate\( (a) \)** is created which checks whether \( a \) is enabled (which corresponds to transition 1 of Figure 7.7 and Figure 7.3). If an interaction with maximal priority is globally ready, it is immediately known to be enabled.

- **Negotiate\( (a) \)** checks the enabledness of an interaction \( a \) (see Algorithm 6). It asks all negotiators of interactions with higher priority than \( a \), in the set \( \text{higherPrio}(a) \), if their interactions are globally ready by sending a \( \text{READY}(b) \) message to all negotiators of these interactions. In turn the negotiators of these interactions, if not \( \text{BUSY} \), respond positively or negatively as soon as they have the information available.

- **Main** handles local priorities locally. Whenever an interaction \( b \) is known to be globally ready, \( \text{Main} \) kills all threads **Negotiate\( (a) \)** with \( a < b \) because the readiness of \( b \) inhibits \( a \).

- Concurrently to **Main**, **WaitingForCommit** handles incoming **COMMIT** messages (see Algorithm 7). Whenever a **COMMIT\( (a) \)** is received — which implies that \( a \) is enabled. Indeed, if \( a \) is involved in a priority rule, then the message **COMMIT\( (a) \)** is first sent by its negotiator. Otherwise, the first controller finding \( a \) globally ready will commit to it (transition 5 in Figure 7.7). As our goal is firing an interaction as fast as possible, the **WaitingForCommit** activity is added concurrently to allow detecting such **COMMIT** messages and thus to terminate all other negotiation activities and to response back by a **COMMIT** (which corresponds to transition 11 in Figure 7.7).

- **Main** tries to commit to the first interaction found enabled (as a way to handle local conflicts) by activating **TryToCommit** (transitions 4, 5 and 6 in Figure 7.7). **WaitingForCommit**
Algorithm 6 Negotiate

Require: \( \text{higherPrio}(a) = \{ c \mid a < c \} \)

Input: interaction \( a \) \hspace{1cm} Output: \( OK \) or \( NOK \)

toCheck ← higherPrio(\( a \))

for all \( b \in \text{toCheck} \) do
    send \( \text{READY}(b) \)
end for

while toCheck \( \neq \emptyset \) do
    if receive \( \text{READY}(b) \) then
        return \( NOK \)
    else if receive \( \text{NOTREADY}(b) \) then
        toCheck ← toCheck \( \setminus \{ b \} \)
    end if
end while

return \( OK \)

Algorithm 7 WaitingForCommit

Require: set of interactions \( \text{waitingSet} \)

Input: set of interactions \( \text{possibleSet} \) \hspace{1cm} Output: interaction \( a \)

if \( \text{waitingSet} \neq \emptyset \) then
    choose \( a \in \text{waitingSet} \) and kill main and send \( \text{COMMIT}(a) \) and send \( \text{REFUSE}(b) \) for all \( b \in \text{possibleSet} \) and goto BUSY(\( a \))
else if \( \text{waitingSet} = \emptyset \) and receive \( \text{COMMIT}(a) \) and \( a \in \text{possibleSet} \setminus \text{toNegotiate} \) then
    kill main and send \( \text{COMMIT}(a) \) and send \( \text{REFUSE}(b) \) for all \( b \in \text{possibleSet} \) and goto BUSY(\( a \))
end if

if receive \( \text{COMMIT}(a) \) and \( a \notin \text{possibleSet} \) then
    send \( \text{REFUSE}(a) \)
end if
is terminated once \textit{TryToCommit} is activated, in order to avoid multiple commits at the same time. Note that the incoming \textit{COMMIT} messages that are used to be handled by \textit{WaitingForCommit}, will be stored in the variable \textit{waitingSet} and will be treated if \textit{TryToCommit} returns \textit{NOK}.

- \textit{TryToCommit}(a) sends a \textit{COMMIT}(a) message to the corresponding peer and waits for a response (see Algorithm 8). Note that if \textit{TryToCommit} fails committing to \textit{a} because it receives a \textit{REFUSE} message — in that case the peer has committed to a conflicting interaction — the controller starts again by checking the global readiness of its locally ready interactions. Indeed, as the peer has committed to another action its state may have changed. For the interactions \textit{a} for which there exists at least one interaction with higher priority, the commit procedure is always initiated by the negotiator of \textit{a} who is the first one to know about \textit{a}’s enabledness.

\begin{algorithm}
\begin{algorithmic}
\Require: \textbf{Input:} interaction \textit{a} \hspace{2cm} \textbf{Output:} \textit{OK} or \textit{NOK}
\State \textbf{send} \textit{COMMIT}(a)
\If {receive \textit{COMMIT}(a)}
\State \textbf{return} \textit{OK} and \textbf{send} \(\forall b \in \text{readySet}\backslash\{a\} \} \textit{REFUSE}(b)
\EndIf
\If {receive \textit{COMMIT}(b), \textit{b} \neq \textit{a} and ((a, b) \notin \text{cyclesof}(K) \text{ or } (a, b) \in \text{notRefuse}(K))}
\State \textit{waitingSet} \leftarrow \textit{waitingSet} \cup \{b\}
\EndIf
\If {receive \textit{COMMIT}(b), \textit{b} \neq \textit{a} and ((a, b) \in \text{cyclesof}(K) \text{ and } (a, b) \notin \text{notRefuse}(K))}
\State \textbf{send} \textit{REFUSE}(b) and \textbf{readySet} \leftarrow \textbf{readySet}\backslash\{b\}
\EndIf
\If {receive \textit{REFUSE}(a)}
\State \textbf{return} \textit{NOK}
\EndIf
\end{algorithmic}
\end{algorithm}

- Finally, \textit{AnswerNegotiators} is a simple thread which is always active if the component \textit{K\textsubscript{i}} is the negotiator for at least one interaction that dominates some other interaction. It receives messages of the form \textit{READY}(a) for interactions \textit{a} for which \textit{K\textsubscript{i}} is the negotiator. It returns \textit{READY}(a) if \textit{a} is currently in the \textit{readySet} of \textit{K\textsubscript{i}}, \textit{NOTREADY}(a) if it is in the \textit{notReadySet} or if it is not in its \textit{possibleSet}, and otherwise defers the answer until the status of \textit{a} is known.

\textbf{Example 7.2.1} Now we propose to illustrate how the proposed algorithm works in a small example, where global priorities are defined. In Figure 7.4 we give a system consisting in a set of 4 components. As previously, interaction between components is represented synchronization on common labels. The system modeled in Figure 7.4 represents 4 components \textit{K\textsubscript{1}}, \textit{K\textsubscript{2}}, \textit{K\textsubscript{3}} and \textit{K\textsubscript{4}}. Components \textit{K\textsubscript{1}} and \textit{K\textsubscript{3}} synchronize on \textit{a}. \textit{K\textsubscript{1}} synchronizes with \textit{K\textsubscript{2}} on \textit{b} and \textit{K\textsubscript{3}} with \textit{K\textsubscript{4}} on \textit{c}. This system represents a priority rule stating that \textit{c} < \textit{b}. Thus we have to assign a negotiator for each of these
two interactions. Note that this priority is global as b and c do not have any component in common, so we assign negotiators arbitrary. Here we choose K1 as the negotiator of b and K3 as the negotiator of c. Figure 7.5 gives a possible scenario of leading to the execution of the interactions in the system. In the initial state, each component proposes its possible interactions by sending a POSSIBLE message to the corresponding peer. As the interaction a is possible for both components K1 and K3, they will both send and receive POSSIBLE(a) which means that a is globally ready. a is not involved in any priority rule, thus once it is globally read, it is also enabled and both components exchange COMMIT(a) message to execute a. Components K2 and K4 cannot execute any interaction because they are waiting for their peers to synchronize.

When interaction a is performed by K1 and K3 (they enter in BUSY(a) in Figure 7.5), both components change their State. However components K2 and K4 are still asking about the readiness of their interactions (resp. b and c) by sending POSSIBLE messages. When K3 changes its state the interaction c becomes possible, and then globally ready once K3 receives POSSIBLE(c) from K4. K3 is the negotiator of c and c < b, thus K3 has to ask about the enabledness of c, means that it has to ask the negotiator of b (K1) about the readiness of b. Therefore K3 sends READY(b) to K1 and waits for response. In the scenario depicted in Figure 7.5 when K1 receives this message, it does not yet know if b is ready, so, it waits for a message from K2 so that it can answer. When POSSIBLE(b) is received, K1 answers K3 with READY(b) which inhibits the enabledness of c. c has maximal priority, thus no negotiation is needed. K1 and K2 can commit for it once it becomes ready. As K3 has only the interaction c in his possibleSet, it still negotiating c by sending READY(b) until it receives NOTREADY(b).

7.2.2 Avoiding deadlocks due to potential decision cycles

In order to avoid deadlocks due to decision cycles amongst interactions in conflict, we introduce the notion of cycle.

**Definition 7.2.2:** A cycle is a set of interactions \( A = \{a_i\}_{i=1}^n \) involving a set of components \( \{K_i\}_{i=1}^n \) for which the following holds: For all \( i \in [1, n] \), \( a_i \) is an interaction involving the two components \( K_i \) and \( K_{(i+1)\mod n} \) and there exists at least one global state in which all these interactions are enabled. We denote the fact \( A \) is a cycle by \( \text{Cycle}(A) \) Note that, in such global state, each \( K_i \) has at least two enabled interactions, in the corresponding local state, one interaction with its right neighbor and the other with its left neighbor.
Figure 7.5: Scenario of possible executions of interactions a, b and c.
A cycle $A$ bears a risk of deadlock or livelock in a state in which all interactions of $A$ are enabled. Indeed, it represents a symmetric situation for all involved components, where a component could wait forever for all others (deadlock) or propose a different choice than all others, reject it and start all over forever. In [Bag89], for example, to deal with this problem, a total order over the system interactions is defined, which allows to avoid deadlock by executing the interaction with higher order. In [PCT04], a similar solution is proposed by imposing a total order over all components, which breaks the cycle by executing the interaction proposed by the component with higher order.

The solution we propose is to detect statically the set of (minimal) cycles of the system. Then, in a second step, we define for each cycle statically a **Cyclebreaker**, which is one of the components of the cycle. This particular component will arbitrate when a blocking situation actually occurs. This means that whenever a potential deadlock may occur, the interaction that is committed by the **Cyclebreaker**, will be fired. This approach avoids to define a total order of all interactions or components which is useless if there is no cycle.

Thus to avoid deadlocks due to cycles and as given in Algorithm 8, if a given controller sends a **COMMIT** message, then it receives another **COMMIT** message to a different interaction, then either there is no cycle involving these two interactions or it exists at least one. In the second case, if the received **COMMIT** concerns the interaction committed by a **Cyclebreaker**, then the controller cannot send back a **REFUSE**. However, if this interaction is not the one committed by the **Cyclebreaker**, in this case the controller can send back a **REFUSE** which breaks the cycle (more details are given in the following illustrative example).

**Notation 7.2.3** We denote by $\text{cyclesof}(K)$, the set of pairs of interactions of $K$ involved in some cycles. $(a, b) \in \text{cyclesof}(K)$ implies that $a$ and $b$ are interactions of $K$ and $\exists A$ such that $\text{Cycle}(A) \wedge \{a, b\} \subseteq A$.

We denote by $\text{Cyclebreaker}(A, K)$ the predicate which holds if the component $K$ is the **Cyclebreaker** of $\text{Cycle}(A)$.

We denote also by $\text{notRefuse}(K)$ the set of pairs of interactions of the form $(a, b)$ such that $(a, b) \in \text{notRefuse}(K)$ implies:

1. $(a, b) \in \text{cyclesof}(K)$

2. $\forall \text{Cycle}(A)$ such that $\{a, b\} \subseteq A$, $\text{Cyclebreaker}(A, K_a)$ holds, where $K_a$ is the peer of $K$ in the interaction $a$. This means that whenever $K$ sends **COMMIT**($b$) message, and then it receives **COMMIT**(a), it cannot send back **REFUSE**(a).

Note that the order of interactions of a pair in $\text{notRefuse}(K)$ is relevant as the first interaction is the one that cannot be refused by $K$. Note that a pair of interactions $(a, b) \notin \text{notRefuse}(K)$ means that either there is no cycles involving these two interactions $((a, b) \notin \text{cyclesof}(K))$ or that there exist such cycles $((a, b) \in \text{cyclesof}(K))$ but if $K$ commit for $b$ and receives a **COMMIT** message for $a$ then it can send back a **REFUSE**(a) to its peer $K_a$, because the latter is not the **Cyclebreaker** of these cycles. Theorem 7.3.5 proves that this way to deal with cycles allows indeed to avoid deadlocks.

**Example 7.2.4** Figure 7.6 depicts an example representing a feasible cycle. The system consists of 4 components: 3 components $\{K_1, K_2, K_3\}$ forming with their set of interactions $A = \{a, b, c\}$ a
7.3. CORRECTNESS OF THE PROTOCOL

Figure 7.6: An example with cycle and independence.

cycle. The component \( K_4 \) represents a completely independent component. The existence of a cycle can be concluded from the structure and the behaviors of the components (the interactions \( a, b, c \) are always enabled). If no priority rules are defined on the set of interactions \( A \), then the cycle \( A \) may lead to a deadlock. A possible deadlock scenario is depicted on the right side of Figure 7.6. This occurs when \( K_1 \) sends a COMMIT message to \( K_{i+1} \) and waits for it. Which means that each component is waiting its peer who has made another choice. According to the proposed solution, let suppose that \( K_2 \) is chosen as the Cyclebreaker of \( A \).

According to Algorithm 8 (as described in Figure 7.6), whenever component \( K_i \) which is already engaged in committing an interaction and which receives a COMMIT for a different interaction, will send back a REFUSE message if this COMMIT is about an interaction which does not form with the already committed interaction a pair in notRefuse(\( K_i \)). However, if it is the case no REFUSE message will be sent back. Thus only the interaction committed by the Cyclebreaker of \( A \) will not be refused and will be indeed fired. Note that cyclesof(\( K_1 \)) = \{ (a, b) \}, notRefuse(\( K_1 \)) = \{ (b, a) \}, cyclesof(\( K_2 \)) = \{ (b, c) \}, notRefuse(\( K_2 \)) = \emptyset, cyclesof(\( K_3 \)) = \{ (c, a) \} and notRefuse(\( K_3 \)) = \{ (c, a) \}. Independently, the component \( K_4 \) can perform whenever it is possible the interaction \( d \).

7.3 Correctness of the protocol

In this section we prove that our protocol defined by the proposed algorithms guarantees the following properties:

1. Exclusion, i.e., interactions in conflict cannot be committed simultaneously.

2. Safety, i.e., if an interaction is fired then it is enabled.

3. Liveness (progress), i.e., if an interaction is enabled, it will eventually become disabled either because it is executed or because a component offering it commits to another interaction.
CHAPTER 7. DISTRIBUTED CONTROLLERS FOR SYSTEMS WITH PRIORITIES

To provide a proof check, we use the state transition diagram of Figure 7.7 where for a local controller, transitions represent steps of the algorithm and states represent the modes of the algorithm. Transitions may have a guard and an action and are depicted in Table 7.2.

Definition 7.3.1 We denote by \( \text{waits}(K_1, a, K_2) \), the global predicate which holds when the component \( K_1 \) has sent a \( \text{COMMIT}(a) \) message to its peer \( K_2 \) involved in the interaction \( a \) but has not yet received an answer in a global state in which \( a \) is enabled.

Lemma 7.3.2 If \( \text{waits}(K_1, a, K_2) \), then \( K_1 \) will receive a \( \text{REFUSE}(a) \) or a \( \text{COMMIT}(a) \) message within a finite delay.

Proof As we assume that the actual execution of an action \( a \) as well as all the basic functions used in our algorithm terminate and every message reaches its recipient within a finite delay. If \( K_1 \) waits for an answer, after sending a \( \text{COMMIT}(a) \) message to \( K_2 \), this means that it exists a global state of the system in which \( a \) is enabled and that \( K_2 \) is in \( \text{Committing}(a_1) \) in the diagram of Figure 7.7. Indeed in the rest of states of this diagram \( \text{Waiting, Active and Negotiating} \), the activity \( \text{waitingForCommit} \) depicted in Algorithm 7 catches this \( \text{COMMIT}(a) \) message and sends back a \( \text{COMMIT}(a) \) to \( K_1 \). Similarly, \( K_2 \) is in \( \text{Committing}(a_1) \) means that it exists a global state of the system where \( a_1 \) is enabled and one of the following cases holds:

1- \( (a, a_1) \in \text{cyclesof}(K_2) \) and \( (a, a_1) \notin \text{notRefuse}(K_1) \) (according to the guard of transition 10 of Table 7.2), in this case \( K_2 \) sends back a \( \text{REFUSE}(a) \) to \( K_1 \) within a finite delay.

2- \( (a, a_1) \notin \text{cyclesof}(K_2) \lor (a, a_1) \in \text{notRefuse}(K_2) \), in this case \( K_2 \) is also waiting for an answer from \( K_3 \) about \( a_1 \). Similarly, if \( K_3 \) does not answer with a \( \text{REFUSE}(a_1) \) then it exists an interaction \( a_2 \) such that \( (a_1, a_2) \notin \text{cyclesof}(K_3) \lor (a_1, a_2) \in \text{notRefuse}(K_3) \). As there exists a finite number \( n \) of components in the system, this means that there exists some cycle of size \( k \) of the form: \( \text{waits}(K_1, a, K_2) \land \text{waits}(K_2, a_1, K_3) \land \ldots \land \text{waits}(K_k, a_{k-1}, K_1) \) and where the following holds:

\[
(a, a_1) \notin \text{cyclesof}(K_2) \lor (a, a_1) \in \text{notRefuse}(K_2) \\
(a_1, a_2) \notin \text{cyclesof}(K_3) \lor (a_1, a_2) \in \text{notRefuse}(K_3) \\
\ldots \\
(a_{k-2}, a_{k-1}) \notin \text{cyclesof}(K_k) \lor (a_{k-2}, a_{k-1}) \in \text{notRefuse}(K_k)
\]

This is a contradiction. Indeed, the first part of each property means that there is no cycle containing these interactions, which is not true as we have a circular sequence which means a cycle. The second part does not hold as we assume that each cycle has just one Cyclebreaker which can try to commit to only one interaction. \( \square \)
7.3. CORRECTNESS OF THE PROTOCOL

![State diagram of the algorithm.](image)

**Theorem 7.3.3 (Safety property)** Let be \( q \) a state, then an interaction \( a \) is fired in \( q \) implies that \( a \) is enabled in \( q \).

**Proof** Let suppose given a global state \( q \), that \( a \) is fired and that \( a \) is not enabled in \( q \). An interaction \( a \) is fired if each component involved in \( a \) enters state \( BUSY(a) \) and thus it sent and received a \( COMMIT(a) \). Let suppose that \( K_1 \) and \( K_2 \) are the components involved in \( a \) in respectively the local states \( q_1 \) and \( q_2 \) of \( q \). \( a \) is not enabled in \( q \) means two cases: either \( a \) is not globally ready in \( q \), or that \( a \) is globally ready in \( q \) but not enabled.

1- Let suppose the first case that is \( a \) is not globally ready in \( q \), this means that \( a \) is not locally ready for at least one of the involved components, that we suppose w.l.o.g \( K_1 \). This means that \( a \not\in possibleSet \) of \( K_1 \) in \( q_1 \) and thus according to transition 11 of Table 7.2, \( K_1 \) will never send a \( COMMIT(a) \) to \( K_2 \) even if the latter sends a \( COMMIT(a) \) to \( K_1 \). Indeed, \( K_2 \) may send a \( COMMIT \) to \( K_1 \), if it receives for example an old \( POSSIBLE \) message from \( K_1 \) before the latter sends the \( NOTPOSSIBLE \) according to transition 13 of the Table 7.2. Thus if \( a \) is not globally ready, \( a \) cannot be fired.

2- The second case is that \( a \) is globally ready but not enabled. First \( a \) is globally ready means that both components have exchanged a \( POSSIBLE \) message as described in transition 1 of Table 7.2. Once \( a \) is found globally ready, than \( a \) is not enabled means that \( a \not\in prioFree \) which means that the transitions 4 and 5 of Table 7.2 cannot be fired and thus that the state \( committing(a) \) in the state diagram of Figure 7.7 is not reached. This means that to fire \( a \) each component has to fire either the transition 11 or the sequence of two transitions 4 then 7.

\( a \not\in prioFree \) means that \( a \) is involved in at least one priority rule. Let suppose w.l.o.g that \( K_1 \) is the negotiator of \( a \), which means that to fire \( a \), \( K_2 \) cannot commit for \( a \) before receiving the \( COMMIT \) from \( K_1 \). To send a commit \( K_1 \) has to fire the transition 4 which has as guard that Negotiate(\( a \))=ok which means that \( a \) is enabled and which is in contradiction with our assumption that \( a \) is not enabled. Thus we conclude that if \( a \) is fired than it is enabled. Note that this is guaranteed by the fact that we suppose that there is no confusion. Indeed, if a situation of confusion occurs there is no guarantee that the algorithm negotiate will get a consistent information about the global
readiness of interactions with higher priority.

**Theorem 7.3.4 (Exclusion property)** Let be \( q \) a state, \( a \) an interaction and denote \( A = \{ a_i \}_{i=1}^n \) the set \( \text{Conflict}_q(a) \cup \text{PrioConflict}_q(a) \) of interactions that are in conflict with \( a \) in state \( q \). Our algorithm guarantees that if \( a \) is fired in state \( q \), no interaction in \( A \) is fired in \( q \).

**Proof** What we have to prove is that if in state \( q \) a component commits to interaction \( a \), by executing one of the transitions 4, 5 or 6 of Figure 7.7, no interaction in \( A \) can be committed before the execution of \( a \) is terminated.

Suppose that \( a \in \mathcal{P}_i \). For all \( b \in A \) we have either \( b \in \mathcal{P}_i \), which means in structural conflict with \( a \), or \( b \in \text{prioConflict}_q(a) \) that is in prioritized conflict, this holds because two interactions can only be in structural conflict if they share a common component. We prove the theorem separately for these two cases:

1- First case: \( b \in \mathcal{P}_i \), that is \( a \) and \( b \) share the same component \( K_i \). First of all, only interactions committed by both peers are executed. Then, if \( K_i \) has sent a \( \text{COMMIT}(a) \) message executing one of the transitions 4, 5 or 6 of Figure 7.7 then according to the same table it is impossible to send a \( \text{COMMIT}(b) \) message before either a \( \text{REFUSE}(a) \) is received or the BUSY state is entered, then exited and the next state reached.

2- Second case \( b \in \text{prioConflict}_q(a) \) holds, that is \( a \) and \( b \) are concurrent (and thus belong to different components) and either \( a < b \) or \( b < a \). Suppose that \( K_j \) is the negotiator for \( b \).

If \( b < a \), then \( b \) should not be executed before the execution of \( a \) — which has started — has been completed and \( K_i \) enters READY for the successor state of \( q \). We have now to prove that from that moment on \( K_j \) cannot “believe that \( a \) is not ready” which is the condition for committing to \( b \).

Indeed, if \( K_j \) does not yet know about the readiness of \( a \), before committing \( b \), it will send a \( \text{READY}(a) \) message to \( K_i \), but as \( a \) is already engaged for execution, \( K_i \) will not send any response before the execution of \( a \) is terminated the next state reached, and the readiness of \( a \) evaluated in the new state; and \( K_j \) remains blocked for \( b \) during this time.

Now, we must prove that \( K_j \) cannot have old, depreciated knowledge that \( a \) is not ready. This can only be the case, if at some point \( a \) was not ready and \( K_i \) has sent \( \text{NOTREADY}(a) \) to \( K_j \), and then transitions concurrent to \( b \) have been executed leading to the current state \( q \) in which \( a \) is ready and executed, and \( K_j \) may use incorrect knowledge and execute \( b \). This corresponds exactly to a situation of confusion, which we have excluded.

If \( a < b \), the situation is almost symmetric. We must prove that in this case \( b \) is not ready. If \( K_i \) is the negotiator for \( a \), asks the negotiator of \( b \) whether \( b \) is ready, and only if the answer is negative, it will consider \( a \) to be enabled and may initiate the commitment of \( a \). Again, only if confusions exist \( K_j \) may use old knowledge. If the negotiator of \( a \) is the peer, then \( P_i \) will only commit to \( a \) on reception of a \( \text{COMMIT}(a) \) from its peer which uses the same procedure for deciding to commit to \( a \).
7.3. CORRECTNESS OF THE PROTOCOL

**Theorem 7.3.5 (Liveness property)** Let $a$ be an enabled interaction. Our algorithm guarantees that $a$ will eventually become disabled either because it is executed or because a component offering it commits to another interaction.

**Proof** An enabled interaction $a$ may become disabled because it is executed or because a component offering it commits to another interaction. When $a$ is enabled for a component $K_i$, a COMMIT($a$) message is sent to the corresponding peer and $K_i$ goes to state $committing(a)$. $a$ becomes disabled when $K_i$ leaves this state. In other words what we have to prove is that $K_i$ cannot stay in this state eternally. When $K_i$ is in state $committing(a)$, there must exist a component $K_j$ such that waits($K_i$, $a$, $K_j$). Thus the proof follows directly from Lemma 7.3.2. In fact, when receiving message COMMIT($a$) or REFUSE($a$), $K_i$ will leave state $committing(a)$ through transition 7 or 8.

**Choosing the negotiators** For each interaction $a$ involved in at least one priority rule, we choose one of the components involved in $a$ as its negotiator. This negotiator will send requests to negotiators of interactions with higher priority and answer requests from negotiators for lower priority interactions. Thid means that whenever $a$ is globally ready, the negotiator communicates with the negotiators for interactions with higher priority to find out whether $a$ is enabled or not. A component may be the negotiator for several interactions. Various strategies may be proposed to allocate negotiators to components. The criterion we use is to minimize for each interaction the maximal number of distinct components to which its negotiator has to send requests. This is meant to minimize the number of communications added due to priorities.

As already explained, local priorities — that means when $a < b$ and $a$ and $b$ have negotiators hosted by the same component $K_i$ — are decided locally. In the dining philosophers example (see figure 8.6), all the priority rules involve the component Forks, which will thus be designated as negotiator and it will enforce priorities locally. In some systems, priorities may include interactions which do not necessarily have a common component. In this case, for each interaction we choose as negotiator the component that is involved in the largest set of interactions, as it may have more knowledge about readiness of more interactions.

**Extension to multiparty interactions** Given a system $Sys = (\{K_i\}_{i=1}^n, T, <)$ defined by a set of components $\{K_i\}_{i=1}^n$ and a memoryless controller defined by $T$ and $<$, we have proposed in the previous section a distributed implementation of $Sys$ by defining to each component a local controller allowing to guarantee $\varphi_T$ and $\varphi_\prec$. The interaction model $T$ of the algorithm previously proposed allows to define only binary interactions. Extending this algorithm to an interaction model with $n$-ary interactions does not affect the way priorities are checked. This extension to multiparty interactions can be done as in $\alpha$-core algorithm [PCT04], where a particular component called coordinator is associated to each interaction. Similarly, we define for each interaction a negotiator. This negotiator has previously the task of checking only the enabledness of the interaction, now it will have also to check its readiness. The criterion to assign negotiators could still be the same as proposed previously.
Algorithm *Negotiate* is unchanged as each negotiator has to ask other negotiators about the readiness of a given interaction, this does not depend on the number of components involved in the interaction. The rest of algorithms proposed have to be slightly modified to deal with multiparty interactions. For this purpose we propose that, for each interaction $a$, the corresponding negotiator collects the responses of all the components involved in $a$ and checks that all of them are ready to execute the interaction. This is done using the exchange of messages *POSSIBLE*. We propose to add two new messages:

- **START**($a$) message sent by the negotiator of $a$ to inform all other components involved that $a$ could be fired;
- **CANCEL**($a$) message sent by the negotiator of $a$ to the participants to inform them that the interaction cannot be fired.

In the Algorithm 7, *WaitingForCommit*, whenever a component $K$ receives a **COMMIT**, it kills thread *Main*, sends back a **COMMIT** and waits for **START**. If it receives the **START** message, it executes the interaction. If it receives a **CANCEL** message, it restarts the Main thread again. Note that the operations performed in this algorithm concerns only interactions for which $P$ is not the negotiator.

In the Algorithm 8, *TryToCommit*, the component sends a **COMMIT** message to all participants involved in the interaction and waits for a **COMMIT** answer from all of them. If it receives at least one **REFUSE** message, it sends back **CANCEL** message to all participants. If it receives **COMMIT** from all participants, it sends back **START** to all of them and goes to state *Busy* to execute the corresponding interaction. Note that all the operations performed in this algorithm concerns only interactions for which $P$ is the negotiator.

In this chapter, we have proposed a protocol allowing to implement a controller enforcing some priority order in a distributed way. In the next chapter, we present a concrete implementation of this protocol and we present some performance analysis results.
### Table 7.2: Transitions of the protocol state diagram.

<table>
<thead>
<tr>
<th>Transition</th>
<th>Guard</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>possibleSet≠ ∅</td>
<td>∀ a ∈ possibleSet, send(POSSIBLE(a))</td>
</tr>
<tr>
<td>2</td>
<td>receive(POSSIBLE(a))∧ a ∈ possibleSet∩toNegotiate</td>
<td>call(Negotiate(a)∧readySet:=readySet∪{a})</td>
</tr>
<tr>
<td>3</td>
<td>receive(POSSIBLE(a))∧ a ∈ possibleSet∩toNegotiate</td>
<td>call(Negotiate(a)∧readySet:=readySet∪{a}∧∀ b ∈ lessPrio(a), kill(Negotiate(b)))</td>
</tr>
<tr>
<td>4</td>
<td>Negotiate(a)=ok</td>
<td>send(COMMIT(a))∧ kill(WaitingForCommit)∧ ∀ b kill(Negotiate(b))</td>
</tr>
<tr>
<td>5</td>
<td>receive(POSSIBLE(a))∧ a ∈ prioFree</td>
<td>send(COMMIT(a))∧ kill(WaitingForCommit)∧ ∀ b kill(Negotiate(b))</td>
</tr>
<tr>
<td>6</td>
<td>receive(POSSIBLE(a))∧ a ∈ prioFree</td>
<td>send(COMMIT(a))∧ kill(WaitingForCommit)</td>
</tr>
<tr>
<td>7</td>
<td>receive(COMMIT(a))∧ Committing(a)</td>
<td>goto(BUSY(a))∧ ∀ b ∈ readySet, send(REFUSE(b))</td>
</tr>
<tr>
<td>8</td>
<td>receive(REFUSE(a))∧ Committing(a)</td>
<td>goto(Active)∧ reset(readySet)∧ keep(possibleSet)</td>
</tr>
<tr>
<td>9</td>
<td>receive(COMMIT(b))∧ Committing(a)∧(a ≠ b)∧ (a, b) ∉ cyclesof(K) or (a, b) ∉ notRefuse(K)</td>
<td>waitingSet:=waitingSet∪{b}</td>
</tr>
<tr>
<td>10</td>
<td>receive(COMMIT(b))∧ Committing(a)∧(a ≠ b)∧ ((a, b) ∈ cyclesof(K) and (a, b) ∉ notRefuse(K))</td>
<td>send(REFUSE(b))∧ readySet:=readySet{b}</td>
</tr>
<tr>
<td>11</td>
<td>receive(COMMIT(a))∧ a ∈ possibleSet\toNegotiate</td>
<td>send(COMMIT(a))∧ ∀ b ∈ possibleSet and b ≠ a, send(REFUSE(b))</td>
</tr>
<tr>
<td>12</td>
<td>true</td>
<td>set(possibleSet)</td>
</tr>
<tr>
<td>13</td>
<td>receive(POSSIBLE(a))∧ a ∉ possibleSet</td>
<td>send(NOTPOSSIBLE(a))</td>
</tr>
</tbody>
</table>
Chapter 8

Implementation and Experimental Results

This chapter presents an implementation of the protocol described as algorithms in the previous chapter. In particular, we describe how we evaluate the performance of our protocol on hand of the implemented prototype. We have implemented the proposed protocol, using Java 1.6 and Message Passing Interfaces (MPI) and we experiment its efficiency on different examples. We have used the MPI library [SOHL+96] to perform the communication layer of our algorithm because of its good performance, usage facility and its portability [GLS99]. In this chapter, we analyze the performance of the algorithm on hand of a number of experiments and we measure three different metrics, namely message count, synchronization time and selection time.

8.1 Sensitivity of the prototype

In our prototype, the exchange of messages between components is performed at the MPI layer and all computations of our protocol are performed at the Java program level (see Figure 8.1). Tests have been run on a set of 2.2 GHz Intel machines with 2 GB RAM, in a configuration where each physical machine hosts only one component.

Our experiments evaluated essentially two metrics which are comparable to those used also in [PCT04]: the first is a metric called message count which measures the (average) number of messages required to schedule an interaction for execution, starting from the moment on that it is ready in one of the involved components. The second one is called response time and is defined as the sum of two other metrics sync time and selection time:

sync time (synchronization time) measures the (mean) time taken by the algorithm to ensure that a given interaction is globally ready, again starting from the moment where it is locally ready in at least one of the peers. An alternative option would be to measure only from the moment on where the interaction is already enabled, that is only the time required to "detect" this enabledness; this is however quite difficult to evaluate in a distributed setting.

selection time measures the (mean) time taken by the algorithm to select an interaction for execution
8.1. SENSITIVITY OF THE PROTOTYPE

once it has been found globally ready.

All metrics are measured for a given system by experimenting with different choices of parameters. We then analyze how variations of parameters affect the considered metrics and compare them to theoretical analysis on the algorithm.

We also compare for an example without priorities the message count metric obtained for our algorithm and for an implementation of the α-core algorithm. We could not compare execution times because the implementation of α-core we have at hand cannot be run in the same setting and the data provided in \[\text{[PCT04]}\] are obtained in a incomparable setting as well.

\[\begin{array}{c}
K_0 \\
\text{Java-Instance}_0 \\
\text{MPI-rank}_0 \\
\end{array}
\begin{array}{c}
K_1 \\
\text{Java-Instance}_1 \\
\text{MPI-rank}_1 \\
\end{array}
\begin{array}{c}
K_2 \\
\text{Java-Instance}_2 \\
\text{MPI-rank}_2 \\
\end{array}\]

Figure 8.1: Implementation layers

In this Section, we study the sensitivity of our algorithm to the degree of conflict in a given system. The degree of conflict \((d)\) is measured by the number of interactions that may be in actual conflict with any (or a particular) interaction. Remember that we distinguish between structural and prioritized conflict (see Definition \[\text{[7.1.4]}\]). Thus in this section, we study first the sensitivity of our prototype to the prioritized conflicts, then to the structural one.

8.1.1 Sensitivity to prioritized conflicts

The purpose of the algorithm that we implement is to ensure correct synchronization between components by respecting global priorities. We first show some results concerning prioritized conflicts.
CHAPTER 8. IMPLEMENTATION AND EXPERIMENTAL RESULTS

![System pattern for experiments](image)

Figure 8.2: System pattern for experiments

An evaluation has been undertaken using the example depicted in Figure 8.2 with a single global state. For each considered configuration, the system has been executed several times, and each execution has been terminated at the execution of the first interaction. The system consists of a set of $n$ components, and a set of $n$ interactions building a circular chain. This pattern is flexible and it allows as to observe how our algorithm performs in different situations. In fact, we can easily add both local and global priorities.

Considering a given system, that is a composition $\{K_1, ..., K_n\}$, the degree of conflict $d$ can be increased by adding priority constraints. Here, we simply count the maximal number of priorities in which a single component is involved, in order to obtain the degree $d$, but as the discussion will reveal, finer measures could also be considered. If there are no priorities in the system, then $d = 0$.

The degree of conflict of $(S, <_1)$ is greater than the one of $(S, <_2)$, if $<_1$ involves more interactions of different components of $S$ than $<_2$.

As already explained, our experiments are performed on a system as depicted in Figure 8.2, for $n = 4$ and using the following priorities to achieve different degrees of conflict, where component $K_2$ — which is chosen as the negotiator of $a_1$ — is the component which in all cases is involved in all the priorities, whereas other components are involved in at most two of them: $d = 0$: no priorities, $d = 1$: $a_2 < a_1$, $d = 2$: $a_2 < a_1 \land a_3 < a_1$, $d = 3$: $a_2 < a_1 \land a_3 < a_1 \land a_4 < a_1$, $d = 4$: $a_2 < a_1 \land a_3 < a_1 \land a_4 < a_1 \land a_3 < a_2$. We have measured the average message count, response time, sync time and the selection time for all cases.

Variation of metric message count Figure 8.3 shows that, as expected, the number of messages exchanged in order to execute the first (and unique) interaction increases with the degree $d$ of the system. Increasing $d$ means that more interactions are involved in priority rules, and thus more messages of type READY are exchanged, and globally less interactions can be executed. In the case chosen for $d = 1$, the priority is defined by the rule $a_2 < a_1$ which is a local priority involving only component $K_2$. Thus, no negotiations and no READY messages are needed which makes in principle, the same message count as for $d = 0$. This is confirmed by Figure 8.3 showing a non significant difference between $d = 0$ and $d = 1$. For the case $d = 2$, the selected priorities are $a_2 < a_1$ and $a_3 < a_1$. This means that the negotiator of $a_3$ ($K_3$) has to send a READY message to the negotiator of $a_1$ ($K_2$) and the latter has to send back as a response a READY message which
8.1. SENSITIVITY OF THE PROTOTYPE

Figure 8.3: Sensitivity to the degree of prioritized conflict

makes 2 extra messages added comparing to the case of \( d = 0 \). This is confirmed by the experimental results.

Variation of metric response time  As expected, also the time required to execute the first interaction increases with the degree \( d \) of the system, which can also be seen in Figure 8.3. Again, adding a local conflict (as in the step from \( d = 0 \) to \( d = 1 \)) leads only to a small increase of the response time as the situation is handled locally. The increase is larger when a global priority is added. Note also that the increase in response time is more important than the increase of the number of messages: up to 20% for adding a (first) local priority and up to 50% for adding a (first) global priority. This may look surprising; but indeed, adding a priority requires adding some explicit threads for negotiation, and on the system configuration we use, the time is mainly spent for execution, whereas the communication time is relatively small. Figure 8.3 shows also the sensitivity of the sync time and the selection time of our prototype to the variation of \( d \). Theoretically, the average synchronization time is independent of the number of conflicting interactions in our system. Indeed, to decide the global readiness of a given interaction, a component has to send and receive a POSSIBLE message for this interaction, which is completely independent of whether this interaction is involved or not in a priority rule. This is confirmed by the results of sync time for \( d = 0 \), \( d = 1 \) and \( d = 2 \) (given in Figure 8.3), for which the synchronization time is almost the same. The synchronization times are slightly greater in case \( d = 3 \) and \( d = 4 \). This is due to the order in which messages are received. More precisely, for \( d = 2 \) priorities are \( a_2 < a_1 \) and \( a_3 < a_1 \) which implies that the component negotiating \( a_5 \) will send a READY message to the negotiator of \( a_1 \) to check its readiness. Thus, the negotiator of \( a_1 \) may receive and treat this READY message before reacting to the POSSIBLE messages for the other interaction. We can observe however that for increasing \( d \), the time required to actually choose an enabled interaction, increases considerably. This is not surprising. The fact that
the selection time remains relatively small with respect to the synchronization time allows the overall
response time increase to remain moderate.

8.1.2 Sensitivity to structural conflicts

Now we provide experimental results about the sensitivity of our prototype to the variation of the
structural conflict of a system. Structural conflict arises between interactions when they are all in
the possibleSet of a common component. To study how our algorithm performs with an increasing
number of structural conflicts, we have carried out a series of experiments on a system as depicted in
Figure 8.4. We use a set of systems $T_1, T_2, \ldots, T_n$ where each $T_k$ has $k$ binary interactions, referred
to as $a_i (i = 1, 2, \ldots, k)$, and $k + 1$ components, referred to as $K_i (i = 1, 2, \ldots, k + 1)$. Components
$K_i$ participate in interaction $a_i$, and $K_{k+1}$ participates in any interaction. Therefore, all interactions
are in structural conflict, and the degree of the structural conflict can be measured by the number of
components in the system.

![Figure 8.4: System pattern for experiments (T_k)](image)

Each experiment consisted in executing 100 interactions, and we have evaluated our metrics for up
to 5 conflicting interactions (a system with six components) for several executions of this experiment
for each degree of structural conflict.

Variation of message count We can see in the left side of Figure 8.5 that our algorithm requires
considerably less messages than $\alpha$-core, where we compare with the results provided in [PCT04]
for this same example. This is due to the fact that $\alpha$-core is connector-centric, that is, it creates
an additional component for each interaction whereas our algorithm is component centric, that is
all negotiations are hosted by some component and share the same memory space. This means that
our algorithm can exploit more local knowledge to execute interactions which reduces the number
of messages exchanged. When there is no conflict at all (in $T_1$) both algorithms exchange the same
number of messages, then when the degree of conflict increases, our algorithm performs better. The
system $T_1$ has no conflict, and to execute $a_1$, 3 messages are exchanged (one POSSIBLE and two
COMMIT), thus 300 messages are transmitted during the experiment. When there are conflicts,
8.1. SENSITIVITY OF THE PROTOTYPE

Figure 8.5: Sensitivity to the degree of structural conflict

for $T_2$ for example, again 3 messages are needed to execute an interaction in the best case, but every time an interaction is refused, at the worst case, a penalty of 3 messages is added (one POSSIBLE, one COMMIT and one REFUSE). To execute 100 interactions, 300 messages are needed in the best case, and 212 extra messages have been added for the situations where an interaction has been refused.

Variation of response time Figure 8.5 shows also the selection and the response time. Again, the average selection time is in principle independent of the number of interactions in structural conflict. Because, when no priorities are added and when an interaction becomes ready, only two COMMIT messages are exchanged to execute an interaction. Thus the average selection time should be of about $2 \times \lambda$, where $\lambda$ is the average message transmission time which in our experimental architecture is $\lambda = 0.2$ ms.

Figure 8.5 shows that the measured response time is higher. The reason for this is that our implementation is written in Java, and the loop used to send $k$ POSSIBLE messages by the component $K_{k+1}$ leads to computational overhead. More precisely, when $K_{k+1}$ enters the loop to send $k$ POSSIBLE messages to the different peers, the component $K_i$ which will get the first message sent, will set the interaction $i$ to ready and send back a COMMIT. However, $K_{k+1}$ will not treat this message before the termination of this loop. As the actual communication time is low, the possibleSet of $K_{k+1}$ may contain many interactions, which increases the selection time (only one interaction is committed, all others must be refused).

146
CHAPTER 8. IMPLEMENTATION AND EXPERIMENTAL RESULTS

8.2 The dining philosophers example

In this section, we focus on the well-known Dining philosophers problem, for which we have carried out a series of tests. We consider a variant of the dining philosophers problem inspired from [Pad] and we propose here to deal with this problem using priorities. Philosophers are seen as components who provide thoughts if they are given two forks. These forks represent a shared resource. A problem may arise if each philosopher grabs the fork on its right, and then waits for the fork on its left to be released. In this case a deadlock occurs and all philosophers starve.

This deadlock can be avoided by giving higher priority to requests closer to completion. The priority order that is needed here is \( \{fork_{1}^\alpha < fork_{2}^\beta, fork_{1}^\beta < fork_{2}^\alpha\} \). For readability reasons, in Figure 8.6, the interaction \( fork_{1,2}^{\alpha,\beta} \) in the behavior of the component \( Forks \) corresponds to the interactions \( \{fork_{1}^\alpha, fork_{2}^\alpha, fork_{1}^\beta, fork_{2}^\beta\} \) of the two philosophers. As the component \( Forks \) participates in all interactions involved in these priorities, \( Forks \) is designated negotiator for involved interactions and can ensure locally that priorities are respected.

Experiments have been carried out for the system with the mentioned priorities (depicted in Figure 8.6); then we have also considered a system with two philosophers and separate components for each fork, where deadlock is avoided by the fact that both philosophers first request \( Fork_{1} \), and then \( Fork_{2} \).

<table>
<thead>
<tr>
<th>Dining philosophers</th>
<th>message count</th>
<th>execution time (ms)</th>
<th>execution time(_{\text{Philo}}) (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>With priorities</td>
<td>6</td>
<td>8</td>
<td>30</td>
</tr>
<tr>
<td>Without priorities</td>
<td>6</td>
<td>11</td>
<td>45</td>
</tr>
</tbody>
</table>

Table 8.1: Message count for the dining philosophers.
Table 8.1 shows our measurements for the message count and the response time metrics for both systems. We have also measured the average time required for one philosopher to execute a complete cycle (take forks, think and release forks) which we denote by execution time $\text{Philo}^*$. We observe that the number of messages exchanged is identical in these two systems. Indeed, priorities are local thus do not induce additional messages. However, using priorities leads to a slight increase of the execution time, as we have already observed in our first example, and the explanation remains unchanged. An additional reason is that the system with priorities has only one component to handle both forks, this making the system less concurrent than the system without priorities. This effect of concurrency is particularly visible in the results for execution time $\text{Philo}^*$. 
Part IV

Conclusions and Perspectives
Chapter 9

Conclusion and Perspectives

In this chapter, we conclude the thesis describing the main objectives of the work, the goals we have achieved, the future work directions and its perspectives.

9.1 Conclusions

In this thesis, we have focused on three aspect to reason about complex systems namely design, verification and implementation. We have proposed in a first time (Part II) a contract-based design and verification methodology where systems are modeled as component-based systems. Then, in a second time (Part III) we have focused in their implementation in particular in a distributed setting.

The proposed methodology have been extended to reason about arbitrary sized systems by building systems according to a given grammar of components and where the verification steps of a top level property are reduced to the verification of a dominance problem for each grammar rule.

In this methodology we have used contracts as a means to constrain, refine and implement systems. The notion of contract we use, makes a clear separation between the assumption, that is property of the environment, and the guarantee, that is the property that the system under study has to ensure. Such a separation improves reusability, which is useful in particular when dealing with a system acting in different possible environments. In addition, the contracts we use take into account the structural aspect of the system under study. In fact, glues (interaction models) used to compose components are explicitly represented in our contracts. This allows us to easily model, refine and implement these glues which represent in general low level protocols used for communication between the parts of the system.

We have based our work on a generic contract-framework, that we have instantiated for a component framework allowing the expression of progress properties. This instantiation handles variables and data transfer between components using a rich description of glues.

We have also successfully applied our extended design and verification approach to a case study which consists of an algorithm of sharing resources in a networked system of arbitrarily size for which a progress property has been checked. Then a prototype tool have been developed to perform the different design and verification steps automatically.
9.2 Perspectives

In this section, we discuss about interesting work directions which are either currently being undertaken, or will be planned for future work.

There are several interesting directions to be explored concerning the proposed contract-based design methodology. First, we have excluded the use of contracts for assume/guarantee reasoning: we use contracts as design constraints for implementations which are maintained throughout the development and life cycle of the system. On the other hand, in assume/guarantee based compositional verification, assumptions are used to deduce global properties (see [dRdBH+01a]). We could integrate this into our methodology: as an example, in our network application, it would be enough to ensure that assumptions express sufficient progress to show conformance of a node contract to node progress. We would like also to use our methodology to verify multiple requirements, possibly by using multiple contracts and multiple hierarchical decomposition of the system. A second interesting direction could be to focus on non-functional properties which are in general properties of the glues by applying the same approach, that is defining contracts for these glues and thus refining them later by protocols for example. In the second part of this thesis, we have proposed a protocol to distribute a global controller into a set of local controllers ensuring given properties. The proposed protocol handles binary interactions between these controllers, an extension of this protocol to multi-party interactions as well as an implementation are actually under work. Another improvement on our protocol would be to combine it with the knowledge approaches [RR00a, RW92a, GPQ10]. Indeed, improving the algorithm by adding a first phase of knowledge computation to find out when a locally enabled interaction is guaranteed to be (not) enabled might in some cases significantly reduce the number of message exchanges without reducing the potential degree of concurrency. The reasoning about interaction models and properties as controllers in Part III could be applied to web service applications, when interactions between services and resources play an important role. In this context, our notion of global memoryless controller could be seen as an orchestrator and the distribution of such a controller as a choreography.

We think that this may lead to different challenges for achieving distribution than those considered generally in the domain of web services. In [CHY07, MH05, NCS04], for example, efficient distributions of web service orchestrators have been proposed, where the system description is based on WS-CDL and WS-BPEL. The use of our concepts of interaction models may lead to more concise specifications of webservices which can be more adequate to provide an understanding of the global behavior, and which are also more adequate for the verification of global properties of web services.
applications. Similar formal and abstract specification has been proposed in [QZCY07] with a simple process language for describing behaviors of services from a local viewpoint with formal syntax and semantics.

We can provide some new emerging service through the composition of a set of existing services which may execute a set of tasks on their local memory and may impose constraints on the order in which these tasks can be executed. The new service is then defined by set of service components, and a set of interactions and priorities. If such a service specification contains a component that is involved in all interactions and that imposes order constraints on them, this component corresponds then typically to what is called an orchestrator in the domain of web services. This component would then also interact with the client(s) In absence of such a centralizing component, the emerging service is given in the form of a choreography.

High-level functional and non-functional properties have to be verified on this kind of systems. This is the reason why we propose to keep them as concise and readable as possible, and whenever appropriate, describe interactions amongst several services that should be executed in an atomic fashion by a rendez-vous rather than describing them in the form of some protocol. In this respect, our approach could be more efficient as it allows writing more concise specifications. On the other hand, an obvious drawback is that a generic protocol implementing systems with arbitrary multi-party interactions and global priorities is likely to be less efficient than a hand-crafted protocol for a given purpose and a given set of components.
Bibliography


<table>
<thead>
<tr>
<th>Ref</th>
<th>Authors</th>
<th>Title</th>
<th>In</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>BBBS08</td>
<td>Ananda Basu, Philippe Bidinger, Marius Bozga, and Joseph Sifakis</td>
<td>Distributed semantics and implementation for systems with interaction and priority.</td>
<td>Proc. of FORTE’08, volume 5048 of LNCS</td>
<td>116–133, 2008</td>
</tr>
<tr>
<td>BBPS</td>
<td>Ananda Basu, Saddek Bensalem, Doron Peled, and Joseph Sifakis</td>
<td>Priority scheduling of distributed systems based on model checking.</td>
<td>CAV’09, LNCS, pages 79–93</td>
<td></td>
</tr>
<tr>
<td>BBS06a</td>
<td>Ananda Basu, Marius Bozga, and Joseph Sifakis</td>
<td>Modeling heterogeneous real-time components in BIP.</td>
<td>SEFM, pages 3–12</td>
<td></td>
</tr>
<tr>
<td>BBS06b</td>
<td>Ananda Basu, Marius Bozga, and Joseph Sifakis</td>
<td>Modeling heterogeneous real-time components in bip.</td>
<td>SEFM, pages 3–12</td>
<td></td>
</tr>
<tr>
<td>BCP07a</td>
<td>Albert Benveniste, Benoît Caillaud, and Roberto Passerone</td>
<td>A generic model of contracts for embedded systems.</td>
<td>CoRR, abs/0706.1456, 2007</td>
<td></td>
</tr>
<tr>
<td>BCP07b</td>
<td>Albert Benveniste, Benoît Caillaud, and Roberto Passerone</td>
<td>A generic model of contracts for embedded systems.</td>
<td>CoRR, abs/0706.1456, 2007</td>
<td></td>
</tr>
<tr>
<td>BCP+07c</td>
<td>Albert Benveniste, Benoît Caillaud, Roberto Passerone, Eric Badouel, Bernhard Josko, and al.</td>
<td>Heterogeneous rich component definition: Hrc behavioural core, mathematical semantics.</td>
<td>Speeds project deliverable d2.1.a, December 2007</td>
<td></td>
</tr>
<tr>
<td>Bei90</td>
<td>Boris Beizer</td>
<td>Software testing techniques (2nd ed.)</td>
<td>Van Nostrand Reinhold Co., New York, NY, USA, 1990</td>
<td></td>
</tr>
</tbody>
</table>


BIBLIOGRAPHY


167


<table>
<thead>
<tr>
<th>Reference</th>
<th>Authors</th>
<th>Title</th>
<th>Journal/Conference</th>
<th>Pages</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>[NH87]</td>
<td>Rocco De Nicola and Matthew Hennessy</td>
<td>Ccs without τ’s.</td>
<td>Proc. of TAPSOFT</td>
<td>138–152</td>
<td>1987</td>
</tr>
<tr>
<td>[OGY06]</td>
<td>Iulian Ober, Susanne Graf, and Yuri Yushtein</td>
<td>Using an uml profile for timing analysis with the if validation tool-set.</td>
<td>MBEES</td>
<td>75–84</td>
<td>2006</td>
</tr>
<tr>
<td>[Pnu85a]</td>
<td>A. Pnueli</td>
<td>In transition from global to modular temporal reasoning about programs.</td>
<td>Logics and models of concurrent systems</td>
<td>123–144</td>
<td>1985</td>
</tr>
</tbody>
</table>

173


BIBLIOGRAPHY


BIBLIOGRAPHY


