Computer Science
University of Manchester

Formal Reuse of Hardware Design

Ana C. Vieira de Melo

Technical Report UMCS–95–4–1

Formal Reuse of Hardware Design[1]

Ana Cristina Vieira de Melo

Department of Computer Science
University of Manchester
Oxford Road, Manchester, UK.
melo@cs.man.ac.uk

April 1995

[1]Copyright ©1995. All rights reserved. Reproduction of all or part of this work is permitted for educational or research purposes on condition that (1) this copyright notice is included, (2) proper attribution to the author or authors is made and (3) no commercial gain is involved.

Recent technical reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from `ftp.cs.man.ac.uk` in the directory `pub/TR`. The files are stored as PostScript, in compressed form, with the report number as filename. They can also be obtained on WWW via URL `http://www.cs.man.ac.uk/csonly/cstechrep/index.html`. Alternatively, all reports are available by post from The Computer Library, Department of Computer Science, The University, Oxford Road, Manchester M13 9PL, UK.

Contents

List of Figures

Abstract

The complexity of hardware components has grown rapidly over the past two decades, with the advances in technologies for manufacturing integrated circuits. Also, the widely spread application of hardware components has required them to be competitive to the market. These factors have led to a requirement for new methods and techniques to enhance controllability, quality and productivity of hardware components. Reusability has been recognized as a basic principle for enhancing productivity and quality of engineering products. Additionally, formal development of hardware has emerged as a method to ensure quality and help handle the complexity of description of hardware components. Based on these premises, this thesis introduces a technique for formal reuse of hardware design.

The main idea of this project is to create a "context" in which an existing hardware component must be embedded in order to implement a new desired component. A solution for effective reuse of hardware components depends on mechanisms to compare the behaviour of these elements, where their similarities and dissimilarities are recognized. So, ways to relate components from a reuse point of view are introduced and, based on those, the necessary "context" is formally constructed.

A process algebra is used for the behavioural representation of hardware, and as a foundation for formally reasoning about the reuse of hardware design. Algebraic manipulation of processes allows for managing the representation of complex components. In order to permit reuse of complex components, the "context" is constructed via a "decomposition operator" defined for the particular process algebra. To obtain minimal "contexts", mechanisms for minimization of concurrent processes, in terms of the number of states and communications, are finally presented.

The contribution of this thesis is a basis for algebraic manipulation of reusable components, and a mechanism to create minimal "contexts" for formal reuse of hardware components.

Acknowledgements

I would like to thank Prof. Howard Barringer for his supervision, and for his comments to improve the legibility of the manuscript.

I am indebted to all in the Temple Group who gave me a friendly atmosphere over these research years, my gratitude to all them. I would like to thank, in particular, Alan Williams for the long discussions about EPA, the borrowed texts and the many suggestions to improve an early draft, Brian Monahan for the technical discussions and the many borrowed macros, and Graham Gough and David Carlisle for helping me to fix figures and macros in LaTeX.

I am grateful to my family for their encouragement to get on with my life and my work, and Roger for his unfailing support and encouragement, and his great efforts to teach me the meaning of "patience".

I am also grateful to friends and colleagues who made the research time not only enjoyable but also possible by granting me on countless occasions the chance of enjoying their company.

This work was supported by a grant from the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* (CAPES), Brazil, to whom I am greatly indebted for the financial support provided over the research years.

To my father, Alvaro, and Roger.

... É difícil defender,
só com palavras, a vida,
ainda mais quando ela é
esta que se vê, severina;
mas se responder não pude
à pergunta que fazia,
ela, a vida, o respondeu
com sua presença viva.
E não há melhor resposta
que o espetáculo da vida:
vê–la desafiar seu fio,
que também se chama vida,
ver a fábrica que ela mesma,
teimosamente, se fabrica,
vê–la brotar como há pouco
em nova vida explodida;
mesmo quando é assim pequena
a explosão, como a ocorrida;
mesmo quando é uma explosão
como a de há pouco, franzina;
mesmo quando é a explosão
de uma vida severina.

by João Cabral de Melo Neto,
"Morte e Vida Severina", 1955.

Chapter 1

Introduction

1.1 Motivation

To make engineering products competitive to the market, techniques must be provided to allow such products to be produced at a low cost, in a short period of time and with a desirable quality. These prerequisites for engineering products can be achieved by enhancing their productivity and quality. Reusability is today an important engineering principle whose importance derives from the desire to avoid duplication and to capture commonality in undertaking classes of inherently similar tasks. It is then recognized as a basic principle for developing engineering products. Hardware is essentially an engineering product. So, for reasons of economy, productivity, quality and time to market, it is highly desirable to reuse existing hardware components.

As with any other engineering product, hardware design is reused by the time any component is produced in bulk: the components' design is "shared" by all the end–users (microprocessors, for example). From a hardware designers point of view, hardware components defined on an implementation level are also frequently reused in a very informal way. Some elements that are widely known to the market are combined with others (new or existing components) to construct a new hardware component; new components are constructed looking forward to using functions of some widely known elements. This kind of reuse of existing hardware components is useful but it does not contribute very much in constructing new designs since it is restricted to cases where functionality of standard elements are known by designers, and reuse is then subjected to a restricted range of standard elements. Additionally, the reuse of standard elements has been shown insufficient to develop dedicated and complex hardware components. And the informal character in which these components are reused does not provide an accurate method of hardware development.

The interest of having hardware components dedicated to specific tasks emerges as fabrication of hardware become cheaper. The diversity in function of these dedicated components requires construction of new elements which largely differ from standard ones. This means that these elements must be of entirely new design, and sometimes only a very restricted part is matched to standard components. These dedicated elements rarely become standard because of their specific nature. However, some other dedicated elements might require a design quite similar to these elements already designed.

Standard elements are very often matched to subfunction of entire processes. Some elements constructed exclusively from standards seldom become standard elements. Nevertheless, construction of new elements could eventually match these existing elements that have never been considered as standard, and then their top–level design must be recreated. In addition, most complex elements are never considered as standard (except some widely used microprocessors). So, even when new elements are basically constructed from standards, they have their communication redesigned.

Another feature that must be pointed out is the informal character in which standard elements are reused. In general, these elements are catalogued in one or a set of libraries in such way that their functions are informally described, or in some other cases they are so simple that their logic descriptions are available as definition. In the first case, the elements are reused from their informal description, without formally proving whether their behaviour matches the required ones. The second case prevents reuse of complex components because their definition level makes them unmanageable.

A wide, and more formal, form of reuse hardware components has been addressed by techniques to synthesize these elements. Synthesis of hardware design has now been treated at the various levels of hardware description, from system down to logic level. Also, some attention has been paid to formalization of this task as noted by Busch, Nusser and Rössel in [BNR93]. Some formal methods have been employed to provide verification of hardware such as HOL Proof System [Gor86] and Boyer–Moore Theorem Prover [BM79], but very little synthesis–related work has been developed. The system–level synthesis is very concerned with *partitioning* of systems by adding structure to behavioural objects (in general, the systems are partitioned into standard components). By contrast, reusable components might not fit a natural partitioning of the desired components; a library component could, for example, embed some functionality not required by the specification element and still be useful for reuse. System–level synthesis is indeed an instance of reusability, in the sense that the existing component could eventually be a subsystem of the desired one.

Apart from the restricted form of reuse addressed by synthesis of hardware design, not much attention has been paid to reuse complex hardware components. But, reusability is not exclusively required for hardware development. The necessity of developing new techniques to reuse software was firstly advocated by McIlroy [McI76], in 1968, as an attempt to alleviate the software crisis. He noticed the increasing multiplication of similar codes to solve identical problems, and emphasized reuse of code. Nowadays, the increasing rate of complexity of new systems has shown that beyond code, design must also be a factor in reuse [Fre87a, BR89, HM89], and some studies on this subject has been carried out.

In the current stage of hardware development, high–level specifications of hardware components are similar to software specifications. Besides that, problems of complexity of hardware components, due to the diversification of application domains, lead the development of hardware to face identical software development problems. So, development of hardware and software are so close that makes techniques developed to reuse software be also applicable into the hardware domain.

In the present work, we advocate reuse of hardware design, considering its resemblance with software. From this point of view, concepts and techniques applied to software reuse can also be applied to hardware development.

In this chapter, the wide spectrum of reuse of hardware and software is introduced: certain technical and non–technical problems involved in reusability and several approaches that can be adopted to achieve reuse. Most of the ideas presented here are in the domain of software development. A parallel with specific features of hardware elements shall be made only when necessary. We deliberately use the word "software", considering that high–level definitions of hardware are "software–like" elements. The last section defines the specific problem that is going to be discussed on this thesis, and shows the organization of the present work.

1.2 The Reuse Problem

Reusability cannot be classified as an independent area in computer science, it is in a way an approach of looking at the software development problem [Fre87a]. It is basically a principle that requires certain techniques to be embedded in software/hardware development methods.

The idea of reusing existing designs has been practised since the very beginning of engineering products as much as software development. In the software field, operating systems, software packages such as editors, and spreadsheets, or even mathematical libraries, are forms of software reuse. Besides these end products, certain methodologies, such as some management and training encouraging reuse and some design techniques focusing reuse concepts, carry characteristics that enhance reusability. But, in most cases reusability has been practised in a very informal way; the current methods of developing software/hardware have been shown to be inefficient for reuse. The waste of people and resources has grown at such an uncontrollable rate, with the advent of the production of specific hardware and software, that the development of techniques for enhancing reusability is urged to develop hardware/software.

As P. Freeman [Fre87a] states,

> We are already reusing software to a limited extent today. Perhaps we do this unconsciously, informally and inefficiently, but the seeds of reusability already lurk in our current methods of software development and maintenance.

Construction of engineering products are in general obtained under the application of certain methods that allow for designing and manufacturing these products. The final products are a result of the effort of making people work in harmony with tools. Any activity that depends on the intervention of human beings must involve some managerial procedures. As a principle of engineering, reusability activity involves managerial as well as technical issues.

To take reusability seriously, a range of technical and non–technical problems must be sorted out. This section analyses what information can be reused in software/hardware development and certain problems that are found to make reusability an effective activity. Technical problems are given emphasis because of the nature of our work.

1.2.1 Reusable Information

The information that is actually reused on reuse–based technologies depends on the systems purpose and how much information can be formalized in those systems. However, we must ask ourselves what kind of information is handled through software/hardware development, and which information can be reused.

Reusability is an activity that involves people, concepts, methods and tools and, as such, it can be applied to either any of these elements in isolation or some combination of them. The use of tools to sketch architectural definition of products, for example, is an activity of both people and tools that involves information ranging over skills on the application domain of the problem, adequacy of tools to solve this problem and skills of designers on usage of those specific tools. In principle, all information used in developing software/hardware is reusable as long it can be formalized.

The diversity of information present in the activity of producing engineering products makes it hard to characterize which information can actually be reused. Freeman [Fre87a] divided this information into five levels: environmental, external, functional architectures, logical structures and code fragments. Jones [Jon87], however, defines reusable types of information as: data, architecture, design, programs and common systems and modules. Many other classifications of reusable information can be given accordingly, for example, to levels of information, types, phases of development and the approach used in the development. There is no universal classification of reusable information.

In a very simplistic view of systems information, it can be classified as: knowledge–based and description–based information. The knowledge–based information comprises all the designer's experience, usage of tools and methods involved in systems development, widespread knowledge of application domains, how existing systems are used in practice, and other information which involves acquiring knowledge from the environment of development or usage of existing products design. This information has not been largely explored (even for restricted application domains) due to its informal nature of representation (as far as our knowledge goes, part of this information could only be represented in natural languages). Apart from very few application domains, such as compilers, there are no established methods for developing systems, even for those widely spread systems like accounting or bank–transaction systems [HM89].

The description–based information concentrates upon structural and behavioural description of components. This involves information such as subsystems that are elements of the entire system, the specification of communications among these subsystems and behavioural descriptions of all parts. This descriptive information actually gives the specification of systems from high levels of descriptions down to implementation.

Due to the technical character of our work, and the desire of reusing hardware design undertaking a formal approach, we will concentrate on reuse of descriptive–based information of hardware/software development.

Code vs. Design

Within the universe of description–based information, distinct levels of definitions can be reused: from high–level specifications down to code. Reuse of code has been practised since the beginning of programming. A successful application of this technique can mainly be noticed on reuse of mathematical libraries. Biggerstaff and Richter [BR89] claim that the success of reusing libraries for numerical computation is due to: the domain is very narrow and well–understood and its underlying technology is quite static. To some extent, reuse of code is an unsuccessful approach because of the following factors:

- components become unmanageable in size because of the very detailed level of description;
- due to specificity of language descriptions, components can only be reused to substitute for a very close element; and
- for areas that are dynamically changing, so many details must be changed for each new version of a component that the investment to maintain such a library does not justify reuse.

Looking at reuse as "reuse of subroutines, objects or packages", it can be concluded that a high degree of specificity is induced by source code languages; the most highly reusable components tend to be small. Building systems out of small components leaves a lot of work to be done in building the architectural superstructures that binds the components into a whole system, and the cost of building this superstructure is typically very much larger than the savings afforded by reusing a set of small components.

Making an analogy with hardware components, code is a low–level definition of software as circuit–level components are low–level definitions of hardware. Hardware components at circuit level carry similar problems of reuse of software code. They are very specific and too detailed to be understood. Furthermore, their dependence on details at these levels of description, by the fact that they are implementation–driven descriptions, make their spectrum of reuse very narrow.

Apart from reusing software code or hardware components at the implementation level, the procedures adopted and decisions undertaken for designing systems are strongly desired to be reused. This information has been reused through experience of designers. It is widely known that expert designers can design far quicker than novices. This is a result of use of their experience: analogies are made with other systems they have already developed. The reuse of "designer's experience" requires formalization of this information under knowledge–based technologies, but formalization of such information is still under study.

Notice that design is a mixture of knowledge and descriptive–based information. Regarding descriptive information for hardware design, they can be represented at various abstraction levels, in a formal or an informal way. The specification of requirements and then the structural and behavioural description of systems are usually captured by schematic, timing diagrams or descriptive languages. To achieve reuse, the design of hardware components must be represented so that similarity between components can be identified.

1.2.2 Inhibiting Factors and Dilemmas of Reusability

Reusability presents a set of problems that delay its application to get started. The factors cited in this section were noted by Biggerstaff and Ritcher [BR89] as inhibiting factors and dilemmas of reusability. Some of these elements are purely managerial, but some others are partly technical and they have, perhaps, no general solution. We first present some managerial problems and then go to the other problems that might depend on analysis in the application domain.

"Not invented here:" apart from components that are widely known, such as mathematical libraries or standard hardware components, most designers resist reusing components that were developed by their peers. This problem is largely cultural; people do not believe in pieces of software or hardware that were not constructed in their own environment. Since it is purely cultural, some management procedures are able to amend such a problem by showing that reuse emphasizes creativity of designers. The use of a formal approach can, in some sense, alleviate this problem. Once processes are "proved correct", they tend to be more acceptable for reuse because there is no doubt about their functionality, but doubts on efficiency or selfishness of designers ("I can do better") still remain to be solved as a cultural problem.

Unestablished reuse strategies: there is no general consensus on the approach(es) that must be followed to make the best reuse of existing components, not even for specific areas. Furthermore, the exact gain in terms of cost, productivity, quality, etc, that can be obtained by application of certain techniques and concepts is not clear. "Management will generally not take positive steps until it has a strong notion of what seems like the best path" [BR89]. So, studies on the extension of reuse problem (which has partially been addressed in [Fre87a, BR89, Weg89, HM87, HC91, SPDM94]) and its various technical problems must be sorted out until reuse can be widely practised.

Initial capitalization: the creation of a large number of existing components to make their reuse feasible requires a lot of previous investment. Most companies expect construction of their products in a productive way without extra investment. To obtain reusable components, some time must be spent to make them available for reuse. They must be a standard library element with their appropriate documentation, at a desired level of abstraction, with a justification or correctness proof, etc. The long term investment in populating libraries of reusable elements discourage many companies to get reuse started.

Design representation: the degree of specificity of components on a low–level of description makes their reuse inefficient (sometimes, infeasible). On the other hand, representation of design is an unsolved problem. Design must be represented as factored and partial specifications to allow reuse of existing designs. This is one of the most important inhibiting factors to be solved to make reusability a feasible matter. Some machine–processible languages carry a high degree of specificity while very abstract representations lead to unprocessible design. This is indeed a tricky subject where no consensus has yet been established.

Generality of applicability vs. payoff: the more general the reuse systems, the more difficult it is to obtain effective results. If a reuse system is prepared to respond to a narrower range of application domains, it can do better because its spectrum of alternatives converges. Conversely, wide–spectrum systems require a big number of keystrokes to find a reasonable result. Despite widely applicable systems being highly desired, they can rarely provide a good payoff result.

Component size vs. reuse potential: like the previous item, what is gained in generality is, at the same time, lost in efficiency. Big components tend to be very specific and then rarely reused. On the other hand, if it is too generic, it might have no detailed information and what has to be developed to make it as a final product does not justify the cost of reuse. A balance of abstraction and specificity is very fine and might never be found even for very narrow application domains.

1.2.3 The Range of Reuse Operational Problems

Populating libraries of reusable elements is essential to turn reuse into reality. But, components can only be effectively reused if the following four mechanisms are available: *searching*, *understanding*, *modifying* and *composing* reusable components.

The searching task must be able to locate existing processes that best approximate the desired one. Effective searching mechanisms must offer a catalogue mechanism to allow components to be stored in a way that makes them available by the time a similar process is going to be constructed. Some mechanisms emphasizes catalogue such as the classification scheme presented by Prieto–Díaz in [PD89], or the very informal catalog mechanisms embedded in object–oriented languages where classes and subclasses are potentially reusable elements [MS92, Mey94]. Some other approaches exhibit a more accurate searching task as in PARIS [KRT89], where pre and post–conditions together with assertions on components properties are used to generate a list of candidates of reuse.

Even when efficient mechanisms of finding very close existing components are successfully applied, these elements rarely cover the entire functionality of the desired process. So, the existing component must be fully understood in order to be reused. In informal and systematic approaches for software development, understanding is a completely manual or semi–automatic task. Biggerstaff and Richter [BR89] suggest hypertext systems as a useful technology to help understanding. In fact, hypertext systems for software development [GS89b, GS89a, CFG91] have today surpassed the mere task of storing documents in a non–linear way to expand the view of widely spread information. These organizational structures provide quick access to the diverse information of the reuse candidates, facilitating their understanding. Besides that, they can also be used as implementation technology for searching mechanisms.

The third, and crucial, task in reusing components is modifying the existing element in order to achieve the functionality of the desired one. Again, when using a systematic approach to develop software or hardware, modification of existing components is largely a human domain task. Unfortunately, very few tools have been developed to help in this task.

And finally, representation of components to be composed is one of the most challenging tasks in reusability. Components must capture a complete functionality to be an independent entity. At the same time, they must be composed to create new components with new functionalities. Again, the balance on partiality of component definition and which features can be abstracted to keep components as independent entities is very fine. In addition, the mechanism for composing these elements must be able to deal with local and global compositions.

1.3 Concepts and Approaches for Reuse

Despite several managerial and technical problems making reusability a non–trivial task, some systems have been developed to help reuse. In this section we enumerate certain concepts underlying reusability, and some existing work dedicated to the reuse task in order to illustrate how application of distinct concepts and technologies can achieve reusability of certain aspects of software.

1.3.1 Some Concepts underlying Reuse

Concentrating on the operational part of reuse, we now explore some concepts upon which reuse mechanisms could be based[1].

Abstraction: Abstraction arises with a recognition of similarities between certain objects or processes where their similarities are considered and their differences are ignored. This is perhaps the most important concept to achieve reusability. Components are rarely found to match exactly the new element being constructed. If components are defined at a very specific level, they might never be reused. Also, very detailed description of components implies unmanageable size of elements. So, abstraction in component descriptions must be applied to make them manageable and free of unnecessary details.

Some abstractions of particular elements have currently been embedded in programming (hardware description) languages or development methods. A usual practice in abstraction is found with representation of components as functions. *Function abstractions* may be specified by input–output relations; every input x determines a unique result, $f(x)$, and no other data is considered.

Another important current form of abstraction is *data abstractions*. In this case, transformation results depend on the effect of operation f on an input x as much as the internal state, like finite state machines. In function abstractions the implementation is hidden, while in data abstraction data and operations applied to them are hidden. Function abstraction emphasizes reusability of functions of varying data, while data abstraction emphasizes reusability of data objects and operations that can be applied to them.

Beyond these forms of abstraction, *process abstractions* appears as a third form. It is similar to data abstraction in having internal states and operations for transforming states, but it differs from that by having a control that determines the order in which operations become available for execution.

[1] Some of these concepts are presented by Wegner in [Weg87, Weg89].

Parameterization of functions, data and processes increases abstraction of them. Parameterization of a function, for example, makes abstraction on application domains, while parameterization of data makes abstraction on classes of applications. Parameterization is then an interesting mechanism that drives to abstraction.

Abstraction is indeed the key–point of reusability and the various other forms or concepts that drive to reusability are based on one kind of abstraction.

Objects, Classes, Hierarchy and Inheritance: These are the four basic concepts of the widely known object–oriented paradigm. *Object* is a term used in the computer science community to denote software components having a hidden state and a set of operations or capabilities for transforming the state. *Classes* congregate a collection of objects and operations for object transformations (by sharing common properties) and as a result, they achieve data abstractions.

Subclass holds the same properties from its source classes and incorporates other properties which specialize its data type. By using inheritance and hierarchy properties embedded into object–orientation, subclasses share attributes previously defined in their immediately superior classes and so on. So, class inheritance provides a structured way of organizing components in a library to be systematically reused.

These four concepts working together achieve data abstraction and provide an informal, but rather more organized, way of reusing software under certain constraints.

Reducibility and Equivalence: Reducing a problem to another one means that the reduced problem can be solved by using the same techniques previously used in solving the other problem. Reducibility is also a manner to express reusability. All reducibility results are aimed at mapping problems from a less familiar presentation to a previously encountered form. Notice that abstraction mechanisms must be applied to obtain reducibility. Reduction of one problem to another must abstract the features on which both processes differ and only pay attention to a restricted set of characteristics.

Equivalence is a bidirectional form of reducibility. Again, abstraction can be applied to stratify problems into primary and secondary attributes: relevant and irrelevant elements. In doing so, equivalence classes are created based on primary attributes, and problems sharing the same class can be interchangeably reused. Different abstractions on parameters that are regarded as fixed or variable determine different choices of equivalence classes. So, equivalence of problems by making abstraction of secondary attributes is also an useful concept that leads to reusability.

Models: Mathematical formalisms are concerned with the development of models of valid reasoning that may be reused for all informal interpretations in specific domains about which we want to reason. Model building is closely concerned with reusability. In order to achieve reusability, an abstraction notation to model computation should be chosen, considering the purposes for which the abstraction is to be used. Structural and semantic models can be defined by using mathematical frameworks, e.g., first–order predicate calculus, temporal logic, modal logic, process algebras or operational and denotational semantics. In order to develop a unified approach to the analysis of components we need them to lay on a unique model of description.

The usage of models to describe components helps in identification of their relevant features. And, beyond that, these models provide automatic or semi–automatic criteria to describe components making the method itself a source of reusable concepts and tools. Reducibility and equivalence can only be efficiently applied under component modelling.

Invariance and Pattern Recognition: Invariants are properties of a system that remain non–mutable. The input–output relation of a function, for example, is an invariant for all algorithms that realize such a function. In other words, an invariant defined for one domain determines a uniform rule of computation for all arguments in that domain (classifying computations and concepts in terms of reusable invariants).

Pattern recognition aims to discover reusable patterns in classes of phenomena. Objects defined as undergoing the same approach and goals (regularly defined) are often easier to be matched than objects defined without any method. Hence, pattern recognition and regularity are features required for reusable object definitions.

These concepts are also concerned with reducibility and equivalence. Recognizing patterns means reducing the entire system to certain entities where those components are identical either in the model of development applied or in their actual functionality.

Most of these forms of reuse depend on some other concepts. Reducibility and equivalence, for example, cannot be applied without a model underlying definition of components. In practice, reuse can be achieved by applying some of those concepts together with an implementation technology which supports the required concepts.

1.3.2 Different Approaches

Many solutions have been given to drive reuse of code and design. These solutions might concentrate on languages that emphasize abstraction on specifications of systems, or on recognition of patterns that, in a transformational approach, ends up with final implementations, or on patterns that represent distinct features of systems and are plugged together to make the entire system. We shall now cite some existing work on reuse to show how it can be approached in different ways[2].

Management. As we have already seen, reusability involves managerial as well as operational procedures. Some companies like Hartford and Toshiba have enhanced reuse of software by applying managerial procedures. At Hartford, for example, some procedures for promoting awareness and acceptance of reusable code as an effective tool to enhance productivity [Cav89] showed some improvement on reuse of existing code. Reuse of code was obtained by massive investment in producing modules into clean, generic, and supportable units of code, and a great discipline on reusability.

The reuse programme developed for Toshiba [Mat87] is aimed at providing an environment to design, program, test, ship, install and maintain commercial software in a unified manner. The programme basically implements some standard practices to be followed in each phase of software development (accompanied by the required tools) where reuse is emphasized. Since the development method is largely used, identification of elements produced for each phase becomes more feasible. The author claims, in [Mat87], reuse of their commercial software at a rate of 52.1%.

Catalogue. Prieto–Díaz developed a classification of reusable modules [PD89] to address the problem of *finding* the appropriate existing components. He claims that classification of components provides access to existing components as well as making understanding and adaptation easier. He classifies software components by reusability–related attributes (descriptors): the function they perform, the way they perform it, and their implementation details. A conceptual graph to measure conceptual closeness among terms is used, together with an evaluation mechanism to rank functionally equivalent components into a prototype library system. Components are classified accordingly to their rank functionality and placed into the graph.

Environment/Languages. Emphasis on reuse can also be given to features of languages that enhance reuse. Meyer advocated the object–oriented paradigm as a step forward for programming languages and for design to achieve reusability when he defined Eiffel [Mey88, Mey89]. In fact, some concepts embedded in the object–oriented paradigm of design (classes, inheritance, etc) lead to reuse. However, the classification mechanisms found in object–oriented environments are rather informal, and without a well–established discipline of classification, reuse is subjected to designers having identical criteria of classification. Populating libraries of objected–oriented components in a regular manner is one of the directions adopted to overcome this problem. Meyer, for example, presents methodological rules to provide a systematic, explicit and uniform conventions to develop library components in [MS92, Mey94].

[2]This section does not survey software reusability in general. It is aimed at showing distinct approaches in reusing software. For a wider view of this subject, the reader is referred to [Fre87b, BP89b, BP89a, SPDM94].

Some other environments emphasize use of a specification language as a key for reuse. Prywes and Lock claim improvement on reuse activity by use of a nonprocedural equational–type language and its respective compilers [PL89]. MODEL [PPS79] is used in an environment integrating documentation and prototyping together with the MODEL automatic generator of programs [PP83] to support software development. The abstraction level of definitions used in specification languages allows reuse of a single definition for various applications. An experiment carried out on accounting systems shows a successful application of such environment in the sense that it is less error prone and the extensions of existing systems are time effective.

Composition–based Systems. From a technological point of view, some systems are strongly based on a mechanism that permits reuse. The Templates Method [VK89], for example, factorizes the programming process into two activities: algorithm specification and implementation. The first element, the templates, specifies the essential properties of the algorithms without paying attention to data types and their representation. Data types are implemented in an imperative language and are the implementation specifications. Once a template is chosen, it can be automatically combined with the possible data types in order to generate the various implementations. Instantiations of templates are created from *instantiation directives* by identifying the template and a context for instantiations. The composition operator is indeed the generator of programs that plug algorithm specification and data type implementation together.

The composition–based technology is also applied to the system PARIS [KRT89] that is based on partially interpreted schemas. The system maintains a library of schemas in which each entity is defined by: entity list, applicability conditions, result assertions, section conditions and schema body. Quantified preconditions, postconditions and invariants, and temporal assertions are allowed as definitions in these schemas. The schema body contains constant and abstract entities that are interpreted into concrete entities. The user states a problem by defining entity list, applicability conditions and resulting conditions. The system searches the schema library to find a possible candidate. Then, the abstract entities are interpreted and non-program elements are matched manually or automatically. In both cases, the Boyer–Moore theorem prover is invoked to do the formal verification and then, in interaction with the user, the program sections are settled.

Transformation–based Systems. An interesting system for reuse of software design is Draco [Nei89]: a transformation rules and generator–based system. It is based on an approach to construct software systems from reusable parts influenced by a practical point of view. It respects the natural procedure for developing software by cycling through three phases of definition: analysis of the application domain, model of the application domain and then design; all of which are assisted by the system. Once the domain analysis is completed, a domain language and library of concepts are created and stored in a knowledge–based database. This framework is used to set up new specifications where the *analysis information* is reused. Then, reuse of *design information* is achieved by interaction between the system designer and Draco. From the refinement of the problem down to executable code, the designer decides on different implementations ranged in the Draco's domain.

Knowledge–based Systems. A knowledge–based system for reuse in the hardware domain has been developed by Nagaraj [Nag92]. OPSYN (OASYS Based Pseudosynthesis Tool) is aimed at assisting designers in the production of analog circuits. This is a knowledge–based pseudosynthesis tool to explore existing design styles and modify them suitably to meet the design objectives, and hence make the design knowledge global. The design library contains the knowledge of IC designers and the simulation library contains details about the performance function evaluation for each circuit block. The Best Module Selector selects the best circuit template for a given design specification.

1.4 The Thesis Outline

As we have already stated, reusability is a principle required for engineering methods as basis for enhancing productivity and quality. Tools, concepts and people involved in constructing hardware are not part of the final product, even though they are reusable elements of specific technologies [Weg89]. People and tools might be reused continuously in an informal way; managerial directives should be responsible for encouraging such reuse. For reuse of descriptive–information of hardware design, however, mechanisms that capture a way of reusing existing hardware components must be provided.

This work is aimed at providing mechanisms for promoting effective reuse of synchronous hardware components defined in the hardware description language ELLA [MC93]. The ELLA Project run at the University of Manchester has as the main goal supporting development of design and formal verification of hardware components. To reach such a formal approach, a synchronous process algebra (EPA [BGMW94]) is used as foundation for semantic definition of ELLA. Thus our project is, in particular, a study on reuse of ELLA components under a formal approach. Because of that, the model used for representing hardware components is EPA instead of ELLA.

The Thesis Organization

In this first chapter we have discussed about the nature of the reuse problem in the software domain: the information that can be reused, the problems of getting reuse started, and certain concepts and approaches that make reuse a feasible task. The framework used for formally reusing hardware design and a solution for such a problem ought to be presented in the following chapters:

- Chapter 2: This chapter is a brief presentation of the process algebra EPA. It has as the main goal introducing a model for representing and reasoning about hardware components.

- Chapter 3: Many approaches can be adopted to reuse hardware design. In this chapter we propose a particular framework for accomplishing reuse of hardware. The first part is devoted to studying cases of reusable components, and then a proposed approach to achieve formal reuse of hardware design is presented.

- Chapter 4: A solution for effective reuse of hardware design requires mechanisms for comparing partial definitions of processes. This chapter presents new relations on processes to help reuse. Besides that, a classification and decidability of such relations shall be discussed.

- Chapter 5: This chapter presents a mechanism for synthesizing synchronous processes by creating an interface between a library component and the desired specification. The synthesis mechanism is defined via a "decomposition operator" for EPA.

- Chapter 6: Due to the generality of the interface process generated by the synthesis mechanism, minimization of such a process must be provided. This chapter presents certain techniques necessary to minimize concurrent processes and, in particular, a minimization mechanism for the interface process.

- Chapter 7: Conclusions of the present work and some directions for future work shall be presented in this chapter.

Chapter 2

Representation of Hardware Components

2.1 Introduction

A formal approach for reusing hardware components requires a semantic definition of the representations of these components. EPA [BGMW94] is a process algebra originally created for describing the formal foundations of ELLA. This chapter provides a brief presentation of EPA as a model for representing hardware components and as a mathematical foundation for hardware verification.

Process algebras have been introduced in a variety of application areas, notably in formal verification of hardware design [Mil94]. Many features have been introduced to the pioneering CCS [Mil80] and CSP [Hoa81] to provide process algebras for modelling systems with different characteristics. Process algebras with timing elements like Timed CSP [DS92], with value passing [HI91, Mil91], with a *zero* object such as [BB90] and [Win83], with action refinements [AH91] or action structures [Mil93] are all examples how communicating processes can be characterized in different ways.

Some attention has been paid to synchronous process algebras due to their more regular treatment of combinators [Mil89] and expressiveness [dS85]. Circal [Mil85, Mil94] (a calculus for VLSI design), SCCS [Mil89], Meije [Bou85] and many other calculi have been developed undertaking a synchronous approach. EPA is also a synchronous process algebra developed from a Process Algebra Framework (PAF) [BGMW93] and aimed at supporting formal verification of ELLA.

A semantic definition for ELLA, using EPA, can be found in [BGMW94]. In order to provide a formal reuse of hardware components described in ELLA, EPA will be used for representing hardware in the present work.

2.2 Representing Processes using EPA

EPA is originally aimed at representing synchronous processes and based on the foundational work of SCCS [Mil89].

Unlike a few other process algebras which allow for realizing one (indivisible) communication at a time, EPA has "structured actions" which permit a predefined set of actions being synchronously communicated. Some other process algebras, such as the value–passing process algebra by Hennessy [HI91] and Circal, have also adopted the idea of structured actions. This technique enriches actions with a structure in order to obtain different units of process models.

Considering that our main goal is modelling hardware, we first present what is necessary in a language to describe hardware components, in particular, a language that supports formal reasoning. The following sections thus discuss certain prerequisites for having hardware components specified and then present a brief description of EPA basic elements.

2.2.1 Representation of Hardware Components

As an engineering product, hardware must have a medium to communicate its design. It would be impossible, or at least not sensible, to start building an engine without a previous sketch. From specification of systems requirements down to implementation, there must exist a formalism through which the design of systems can be expressed. Besides that, the use of a formalism is necessary as basis to validate systems and as a basic element over which tools can be constructed [Mil94].

Nowadays many approaches can be adopted to express design of hardware components. A variety of descriptive techniques such as schematics, timing diagrams and description languages have been used to specify hardware design. Some of these formalisms are used to capture a particular feature of hardware, such as timing diagrams that capture timing information and block diagrams that capture structure. Hardware description languages, such as ELLA, VHDL [LMS86], Verilog [TM91] and many others, tend to express all the hardware design features using a unique medium. It is clear that a medium for specifying hardware design is necessary, but what are the elements necessary to express it?

The great challenge in modelling physical elements, like hardware, is how to represent, in a clear and precise manner, the physical and behavioural properties of such elements [Mil94]. The physical properties must define the *structure* of elements: their shapes, how they interact with other elements and how their components interact to each other. The *behavioural* representation must express how the system reacts to the various stimuli given by its environment.

Structural description is concerned with representation of interconnection of components to construct more complex elements. Due to the complexity of current hardware components, a natural manner of describing the hierarchical structure of complex elements must be provided by the model. *Behavioural description* must capture how the hardware component reacts when excited by its external environment. Besides expressing behaviour in a concise and accurate manner, the behavioural description must also reflect the natural hierarchy defined in the structural model.

Thus, a representation of digital hardware is useful for description and analysis only if it can naturally and accurately express the physical element, and if the validation techniques are formally sound.

Hardware components can be described at many levels: from system down to circuit level. For each of these abstraction levels, structural and behavioural definitions must be provided. The topmost definition is the most abstract one. As definition goes down in abstraction, each component at a lower abstraction level must also have its structure and behaviour specified. The fact that complexity of components increases along with the introduction of more detailed information, a *hierarchical* description is required to manage complex elements. A *composition* mechanism is then essential to “plug” smaller components.

For structural descriptions, the composition mechanism must determine how components are wired to form a more complex element. A regular manner must be applied to interconnect the building blocks at the various levels of hardware description. No matter whether the building blocks are integrated circuits, gates or chips mounted on a circuit board, the composition mechanism must represent these distinct levels of wires in a regular way.

Like structural descriptions, behavioural composition of components must also be provided for representing hardware. Independent from the level of definition of components, a well–established way of composing components must be able to construct new elements from the behaviour of smaller components.

In contrast to the constructive mechanism of hardware components, mechanisms for *abstractions* of internal structures of a building block must also be provided for behavioural and structural descriptions of hardware. This is the reverse way of constructing elements: once a block is constructed, we want a mechanism that transforms that composite block into a unique element by hiding its internal structure. A mechanism like this provides representation of components at a higher hierarchical level.

The abstraction mechanism must be reflected on both structural and behavioural descriptions. Again, it must be presented as a uniform mechanism overall levels of hardware description.

In order to provide formal verification of hardware design, the medium for representing hardware must be semantically defined. Then, the laws given as semantic definition can be used together with some mathematical techniques to validate the hardware specification throughout the various levels of refinement.

Thus, formal development of hardware can only be obtained if the language used for its representation is able to express structural and behavioural descriptions of hardware in a natural way, and it is defined on a mathematical basis in order to provide formal validation. The "natural way" of representing hardware requires then abstraction and composition mechanisms.

In fact, process algebras and some other formalisms such as HOL [GM93] have been proved to provide an adequate formalism for hardware description and validation [Mil94]. The following sections present EPA as a formalism able to represent and formally validate hardware.

2.2.2 Basic Definitions

In this first part, we discuss the representation and interpretation of certain basic elements defined in EPA:

Processes. Hardware components are essentially reactive elements that respond to their environment whenever a stimulus is given. Processes are defined in EPA as reactive elements that receive stimuli and respond to their environment. They are thus elements that transform an input event into an output one as a way of reacting and, consequently, communicating with the external environment. Stop is a particular process that does not respond to any external stimulus.

Events and Process Communication. Considering EPA processes as synchronous reactive elements, they are indeed static objects that communicate with the environment whenever the clock is pulsed. Events are manifestation of incoming and outcoming process stimuli. When a process is activated by a clock pulse, an incoming and an outcoming event are simultaneously communicated with the environment.

Since events are represented as simultaneous activation of an input and an output event, EPA actions are defined as a synchronous communication of an input and an output action[1]:

$$\alpha \in Act ::= \epsilon \mid \boxtimes \mid (i/o)$$

$\boxtimes$ (*zero action*) is a distinguished action that has "no meaning" in terms of behaviour. It was introduced in EPA to give a uniform treatment for partial behaviour and hiding [BGMW93] (this uniform treatment will be better explained when composition of actions is presented in Section 2.2.3). ϵ is an empty action, and i and o are, respectively, input and output actions that simultaneously occur to form a single event.

All actions that represent process behaviour, such as (i/o) and ϵ, are called "proper" actions. $\boxtimes$ is, however, an "improper" action because it does not represent process behaviour.

Actions and Ports of Communication. To be activated, a process must have "ports of communication" with the external environment; they are vehicles of interchanging information between processes and their external environments.

Since input and output events are communicated simultaneously, EPA distinguishes incoming from outcoming events. As a result, these events are communicated through distinct "ports of communication". EPA input and output actions (io–actions) are, therefore, *structured actions* that can communicate a set of *subactions* at a time[2]:

$$i, o \in IOAct ::= \epsilon \mid \boxtimes \mid io_1 \oplus io_2 \mid pa$$
$$\epsilon/\epsilon = \epsilon$$
$$\boxtimes/\boxtimes = \boxtimes$$

[1] A glossary of symbols is included in Appendix A

[2] io_n is a general notation for input or output actions: it can represent either an input action (i_n) or an output action (o_n).

Intuitively, an io–action can be a single subaction, denoted as pa ($pa \in pIOAct$), a *primitive action*; an empty action, ϵ; a meaningless communication, $\boxtimes$; or still a *summation* ($\oplus$) of simpler io–actions. At the ground level, an io–action is a summation of primitive actions.

In fact, EPA processes are defined over a predefined set of input and output typed ports of communication – called *channels* ($c_1, ..., c_n$) – that are identified by channel names c_i ($c_i \in ChanNames$). A process event is realized by synchronous excitement of all the predefined channels. An EPA action is thus a pair of an input and an output action where each io–action is a summation of primitive actions (primitive actions are represented by a bound channel name and value $-c_i(v_i)$):

$$\begin{aligned} i/o &= c_1(v_1) \oplus ... \oplus c_n(v_n) / c_{n+1}(v_{n+1}) \oplus ... \oplus c_{n+m}(v_{n+m}) \\ &= c_1(v_1), ..., c_n(v_n) / c_{n+1}(v_{n+1}), ..., c_{n+m}(v_{n+m}) \end{aligned}$$

The sequence of primitive actions ($c_1(v_1), ..., c_n(v_n) / c_{n+1}(v_{n+1}), ..., c_{n+m}(v_{n+m})$) is a short–hand for the extended definition of summation of primitive actions; it will be used in preference to the full notation.

The set of ports of communication of a process P is defined as $\mathsf{Chs}(P)$, while the sets of input and output channels are $\mathsf{Chs}_i(P)$ and $\mathsf{Chs}_o(P)$ respectively.

Process Definition and Process Call. An EPA process is defined over a separate set of input and output channels and a set of value–variables that is assigned when the process is called. A process P formally defined over a set of input channels $\{c_1, c_2\}$ and output channels $\{c_3, c_4\}$ is as follows:

$$\mathsf{process}\ P(vb_1 \colon ty_0;\ c_1 \colon ty_1, c_2 \colon ty_2 / c_3 \colon ty_3, c_4 \colon ty_4) \underline{\triangle} \ ...$$

The right–hand side must have a process term that describes the behaviour of P. On the left–hand side, P is defined over three groups: a set of variables that are used in the process and instantiated in the *process call* $\{vb_1, ...\}$; the set of input channels $\{c_1, c_2\}$; and the set of output channels $\{c_3, c_4\}$. Notice that the type associated to each of these channels defines the range of actions that can be passed through such a port of communication.

All variables and channels declared in a process definition are formal parameters that can be instantiated in a *process call*. Assuming all types declared in P are already defined, P can be instantiated as a process term (or subterm) of other processes. *CallP*, for example, has an instance of P as a subterm:

$$\begin{aligned} &\mathsf{type}\ ty_0\ \underline{\triangle}\ \{v_1, v_2\} \\ &\mathsf{process}\ CallP(x \colon\ ty_0, y \colon\ ty_1 / z \colon\ ty_2, w \colon\ ty_3)\ \underline{\triangle}\ \ldots P(v_1;\ x, y / z, w) \ldots \end{aligned}$$

In this instance of P, variable vb_1 declared as type ty_0 ($\{v_1, v_2\}$) is instantiated by the value v_1, while the input and output channels are called by the actual channel names of *CallP*. There is no limit for the number of times a process is instantiated in a process term. P, for example, could eventually be called several times in the definition of a unique process.

Sorts. The sort of all actions through which a process P is able to communicate with its external environment is defined by:

$$\mathsf{Sort}(P).$$

The sort of initial actions (*prefixing actions*) P can perform is given by:

$$\mathsf{Sort}_1(P).$$

Since EPA actions are, at the top level, structured as input and output actions, separate sorts for these distinct actions can also be defined. The sorts of input and output actions are as follows:

$\mathsf{Sort}_i(P)$ and $\mathsf{Sort}_o(P)$.

The initial input actions (denoted as $\mathsf{Sort1}_i(P)$) and the initial output actions (denoted as $\mathsf{Sort1}_o(P)$) also have their sorts defined.

Deterministic Processes. In spite of being defined over synchronous input and output actions, determinacy of EPA processes is defined over *input actions*. This criterion is obvious from the point of view that a process is deterministic if it behaves identically whenever excited by identical events. Since the excitement of EPA processes is given by input actions when the clock is pulsed, processes are deterministic if their input actions are so.

2.2.3 EPA Combinators

In this section we informally present the EPA combinators through examples (however a transitional semantics for EPA combinators has been included in Appendix B). These examples are used to show how EPA supports structural and behavioural descriptions, and how mechanisms for abstraction and composition are embedded in EPA.

We have already introduced, in the previous section, that EPA processes have their input and output ports of communication established in the process definition. Now, the behaviour of processes must be defined. Let us assume P and Q are processes (or agent expressions), C is a set of channel names ($C \subseteq \mathit{ChanNames}$), $\vec{x}$ is a set of variables and α is an action ($\alpha \in \mathit{Act}$). The EPA combinators are as follows:

$\alpha :: P$	prefixing operator
$P + Q$	binary summation
$\Box\, \vec{x} \cdot P$	process choice
$P \mid Q$	composition
$P \backslash C$	hiding operator

Instead of formally defining these combinators, let us give a few examples[3] and informally explain the semantics of such combinators. First, a simple gate is defined: the nor–gate.

$$\begin{aligned}\textsf{process } \mathit{Nor}_p(c_1, c_2\!: \mathit{bool}/c_3\!: \mathit{bool}) \;\underline{\triangle}\; & ((c_1(\texttt{t}), c_2(\texttt{t})/c_3(\texttt{f}))+\\ & (c_1(\texttt{t}), c_2(\texttt{f})/c_3(\texttt{f}))+\\ & (c_1(\texttt{f}), c_2(\texttt{t})/c_3(\texttt{f}))+\\ & (c_1(\texttt{f}), c_2(\texttt{f})/c_3(\texttt{t}))) :: \mathit{Nor}_p(c_1, c_2/c_3)\end{aligned}$$

The nor–gate is defined as a process over input channels c_1 and c_2, and the output channel c_3. It moves for one of the four initial actions and then evolves into the process following the *prefixing* operator. The *summation* operator provides an exclusive choice of one of those initial actions at a time. Thus, the process Nor_p can engage one of the initial actions and then reach the following process and so on.

Another elementary hardware component is a unit–delay: the input is delayed for one step. Process boolean–delay is generically defined as $\mathit{DelBool}$. A process boolean–delay which responds "false" in its initial state is defined by Del_p:

$$\begin{aligned}&\textsf{process } \mathit{DelBool}(x\!: \mathit{bool};\; c_1\!: \mathit{bool}/c_2\!: \mathit{bool}) \;\underline{\triangle}\\ &\qquad \Box\, v\!: \mathit{bool} \cdot c_1(v)/c_2(x) :: \mathit{DelBool}(v;\; c_1/c_2)\\ &\textsf{process } \mathit{Del}_p(c_1\!: \mathit{bool}/c_2\!: \mathit{bool}) \;\underline{\triangle}\; \mathit{DelBool}(\texttt{f};\; c_1/c_2)\end{aligned}$$

[3] The examples defined here are from [BGMW94].

Variable x defines the "state" of process *DelBool*. The process chooses (*process choice* operator) an input value v (in the range of type *bool*) for channel c_1 and binds the actual "state" to the output channel c_2. After undertaking this initial action, it evolves into a recursive call in which the actual input value v is bound to the process's "state". This means that for each value chosen as input, the same value appears as output of this process by the time it engages the following step.

Notice the semantic similarity between the *binary summation* and the *process choice*: both allow choice of an exclusive action at a time. In fact, binary summation is an instantiation of process choice where all the initial actions are explicitly defined instead of parametrically chosen from input variables.

Del_p is an instance of *DelBool* where the initial state is `f`. Whatever value is chosen as input in the first step, Del_p will output `f` and then start outputting the input value chosen in the previous step and so on. Suppose x is a variable of type *bool*, the possible initial actions for Del_p and the following processes are as follows:

$$Del_p(c_1/c_2) \xrightarrow{c_1(\mathtt{t})/c_2(\mathtt{f})} DelBool(\mathtt{t};\ c_1/c_2) \xrightarrow{c_1(x)/c_2(\mathtt{t})} DelBool(x;\ c_1/c_2)\cdots$$
$$Del_p(c_1/c_2) \xrightarrow{c_1(\mathtt{f})/c_2(\mathtt{f})} DelBool(\mathtt{f};\ c_1/c_2) \xrightarrow{c_1(x)/c_2(\mathtt{f})} DelBool(x;\ c_1/c_2)\cdots$$

Composition and Abstraction

The combinators presented so far support a "flat" representation of hardware behaviour. The binary summation and process choice are able to represent the alternative actions over which a hardware component is defined, while the prefixing operator supports representation of sequences of events. A hierarchical representation of components requires mechanisms for *composition* and *abstraction*.

The *composition* operator is, perhaps, the most important combinator for process algebras; it determines the way with which processes are going to communicate. A more detailed explanation of this combinator is then carried out in this section.

Another trivial hardware component is a nor–gate which delays its output for one step: the nor–gate–delay. This component behaves like a nor–gate, except that its output is not readily shown, it waits for another step. Clearly, this component is a composition of a nor–gate with a delay process:

$$\mathsf{process}\ NorDel_p(c_1, c_2\colon\ bool/c_3\colon\ bool)\ \triangleq\ \Big(\ Nor_p(c_1, c_2/c_4) \mid Del_p(c_4/c_3)\ \Big)\backslash\{c_4\}$$

From the structural point of view, the *composition* operator wires the two components, Nor_p and Del_p, on channel c_4 (connections between processes are denoted by identical channel names). Also, this "wire" is hidden ($\backslash\ \{c_4\}$) after composition. This means that the hidden "wire" is abstracted from the structure of the process $NorDel_p$. Thus, $NorDel_p$ is constructively defined from two other subcomponents, Nor_p and Del_p, and, at the same time, its internal structure is no longer observed by the time the *hiding* operator is applied. The parallel composition supports *hierarchical* description of processes while the hiding operator provides an *abstraction* from internal structures.

Apart from being structurally defined, processes must be behaviourally described. A semantic definition for the EPA parallel composition operator is as follows:

$$\boxed{\mathsf{Product}}\ \frac{P \xrightarrow{\alpha} P', \quad Q \xrightarrow{\beta} Q'}{P \mid Q \xrightarrow{\alpha \star \beta} P' \mid Q'}$$

A meaning of the EPA combinators is given by using a general notion of labeled transition systems [Arn94]. A Transition system for P ($TS(P) = \{\mathsf{Sts}_P, \mathcal{A}_P, T^P, s_0^P\}$) consists of a set of states Sts_P representing the process terms, a set of labels $\mathcal{A}_P$ representing actions, a transition relation T^P over process terms which gives the semantics of the system and an initial state s_0^P. The transition of composite processes are defined in terms of the transitions of the component elements. The general rule is defined by: given the transition(s) of the component process(es), as $P \xrightarrow{\alpha} P'$ and $Q \xrightarrow{\beta} Q'$ in the definition above, the transition of the composite process is inferred from the given transition(s) by application of the combinator being defined, $P \mid Q \xrightarrow{\alpha \star \beta} P' \mid Q'$. The rules associated to each combinator is well understood as giving the meaning of that combinator.

The EPA parallel composition provides communication of processes in a synchronous manner. Transitions of the composite process are given by a synchronous communication of actions of the component processes. The action product is defined as follows:

$$
\begin{aligned}
&\star : Act^2 \longrightarrow Act \\
&\forall\ i_p/o_p, i_q/o_q \in Act \bullet \\
&\qquad (i_p/o_p) \star (i_q/o_q) = ((i_p \oplus i_q) \ominus (o_p \oplus o_q))/(o_p \oplus o_q)
\end{aligned}
$$

Intuitively, two actions can only be combined if the value bound to their coincident channels are identical. The concept of *conflict freedom* of actions (Appendix B) [BGMW93] emerge to define which actions can be combined. Actions are *conflict free* if whenever a channel name coincide in both actions, the actual values passing through these channels are identical; they are conflicting otherwise.

Two other operators are used to define the action product[4]: summation ($\oplus$) and subtraction ($\ominus$). Roughly speaking, summation of actions io_1 and io_2, $(io_1 \oplus io_2)$, combines all subactions of io_1 and io_2 if they are *conflict free*, and returns $\boxtimes$ otherwise. On the other hand, subtraction $(io_1 \ominus io_2)$ returns all elements in io_1 that have no match with elements in io_2.

Due to propagation of $\boxtimes$ over action summation and subtraction, the action product results in either $\boxtimes$ if actions are conflicting, or summations and subtractions of input and output actions (as defined above) otherwise. Composition of $Nor_p(c_1, c_2/c_4)$ with $Del_p(c_4/c_3)$, for example, comprises the following initial actions:

	Nor_p action	$\star$	Del_p action		product action
(1)	$(c_1(\mathtt{t}), c_2(\mathtt{t})/c_4(\mathtt{f}))$	$\star$	$(c_4(\mathtt{f})/c_3(\mathtt{f}))$	$=$	$(c_1(\mathtt{t}), c_2(\mathtt{t})/c_4(\mathtt{f}), c_3(\mathtt{f}))$
(2)	$(c_1(\mathtt{t}), c_2(\mathtt{t})/c_4(\mathtt{f}))$	$\star$	$(c_4(\mathtt{t})/c_3(\mathtt{f}))$	$=$	$\boxtimes$

In the product action (1) shown above, the component actions are conflict free and the result is as defined in the action product operator. The second case, however, results in a $\boxtimes$ because the component actions conflict in values for channel c_4. An important feature of this composition operator is the observation of all the output channels of both processes, even when they are also used as input. Conversely, the input channels that coincide with output ones are hidden (c_4 does not appear as input channel of the product action).

The product actions presented above does not represent $NorDel_p(c_1, c_2/c_3)$ actions because the channel c_4 is hidden in such a process. So, the actual action in $NorDel_p(c_1, c_2/c_3)$ that corresponds to that one in item (1) is given as $(c_1(\mathtt{t}), c_2(\mathtt{t})/c_3(\mathtt{f}))$.

Mechanisms for *composition* and *abstraction* of behavioural description of components are also achieved with the composition and hiding operators. *Composition* of processes is performed in a regular manner and *abstraction* turns out with the hiding operator making channels no longer observable. Another form of abstraction is the implicit hiding performed by the composition operator: the input channels that coincide with output ones are all internalized in the compound process.

Thus, structural and behavioural descriptions are supported by EPA. Furthermore, mechanisms for composition and abstraction of behavioural and structural descriptions are supported by the composition and hiding operators.

2.2.4 Transition Graphs

Transition graphs [Arn94] are used for behavioural representation of processes. In fact, many process algebras, such as CCS and CSP, provide mechanisms for comparing processes by using transitional graphic

[4] These operators are algebraically defined in Appendix B.

representations. The present work will interchange algebraic and graphic representations depending on the problem being presented.

EPA actions are graphically represented as input and output actions. The directed graph has the usual meaning: processes are represented by nodes, and arrows represent actions coming from a process to another one. The main process will always be named in the graph. The processes reachable from the main process term, however, will only be named when necessary. Figure 2.1 shows representations of the same process: one of them has all the reachable processes named while the other one has only the main element named.

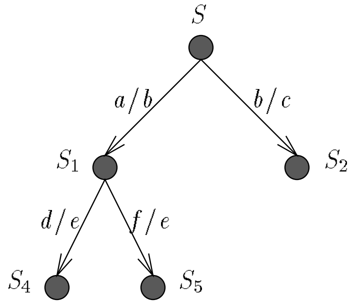

Figure 2.1: Transition Graph for EPA Processes

Input and output actions are summation of primitive actions. The graphical representation of processes showing the action values will drop the channel names for simplification; only the action values are shown as sequences of elements for input and output actions. An action like $(c_1(v_1), c_2(v_2)/c_3(v_3))$ is represented in the graph by $(v_1, v_2/v_3)$ or $(v_1 v_2/v_3)$. The process $NorDel_p(c_1, c_2/c_3)$ $(NorDel_p(c_1 c_2/c_3))$, for example, has its transitional graph with values shown in Figure 2.2.

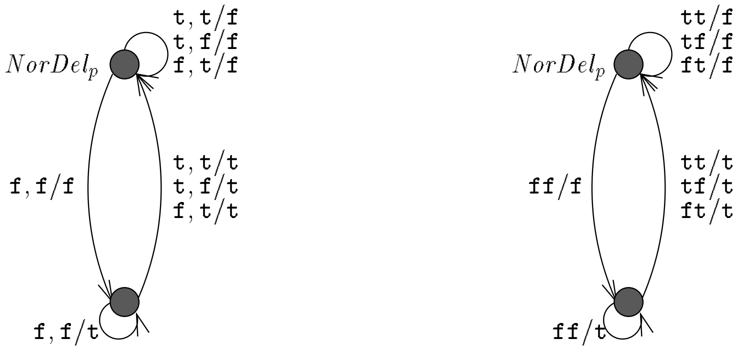

Figure 2.2: Transition Graph for $NorDel_p(c_1, c_2/c_3)$

Whenever the action values representation is used, the sequence of input and output channels will be shown in the text. The reachable processes assume the same sequence of channels defined in the main process.

2.3 Equivalence of Processes

Process algebras are used for representing and validating hardware due to the mathematical foundations over which their equivalence relations are defined. A variety of process behavioural equivalences have been defined. An early model for process semantics is found in the literature of automata theory

[Gil62, HU79, CL89]: *trace equivalence*, an equivalence relation based on traces that ensures substitutivity for deterministic processes. Nevertheless, for nondeterministic processes, many other equivalence relations have been defined concerning with the capacity of processes in performing identical actions and deadlocking. Relations like strong bisimulation [Mil89], observational bisimulation [Mil89], failures equivalence [Hoa85], testing equivalence [Hen88] and symbolic bisimulations [HL92], are all equivalence relations defined for comparing nondeterministic processes[5].

Strong bisimulation [Mil89] is defined for EPA as the equivalence relation for validation of hardware components translated from ELLA, considering deterministic processes free of silent actions. In the present work, strong bisimulation is adopted in order to provide substitutivity of components; reuse requires substitution of processes. From now on, we use the term "bisimulation" to mean "strong bisimulation".

Bisimulation was defined for EPA processes in [BGMW94] as:

Definition 2.1 *Bisimulation $\mathcal{B}$ is a symmetric binary relation on processes such that if $(P, Q) \in \mathcal{B}$ (abbreviated by $P \sim Q$), then for every "proper" action $\alpha \in Act$:*

1. *whenever $P \xrightarrow{\alpha} P'$ then $Q \xrightarrow{\alpha} Q'$ for some Q' such that $P' \sim Q'$,*
2. *whenever $Q \xrightarrow{\alpha} Q'$ then $P \xrightarrow{\alpha} P'$ for some P' such that $P' \sim Q'$.*

Processes are bisimilar if they can perform identical actions and then reach bisimilar processes. So, for each action in P there exists an identical action in Q and vice–versa. Also, for any event that P deadlocks, Q does so and vice–versa. The transition labeled systems of processes are compared for deciding their bisimilarity.

Distinct process definitions can drive to bisimilar processes. Suppose two distinct process definitions are given for the nor–gate–delay process:

$$\textsf{process } NorDel_p(c_1, c_2\colon bool/c_3\colon bool) \triangleq \Big(Nor_p(c_1, c_2/c_4) \mid Del_p(c_4/c_3) \Big)\backslash\{c_4\}$$

$$\begin{aligned}
\textsf{process } NorDel_{\mathtt{f}}(c_1, c_2\colon bool/c_3\colon bool) \triangleq\ & ((c_1(\mathtt{t}), c_2(\mathtt{t})/c_3(\mathtt{f}))+ \\
& (c_1(\mathtt{t}), c_2(\mathtt{f})/c_3(\mathtt{f}))+ \\
& (c_1(\mathtt{f}), c_2(\mathtt{t})/c_3(\mathtt{f}))) :: NorDel_{\mathtt{f}}(c_1, c_2/c_3)+ \\
& (c_1(\mathtt{f}), c_2(\mathtt{f})/c_3(\mathtt{f})) :: NorDel_{\mathtt{t}}(c_1, c_2/c_3) \\
\textsf{process } NorDel_{\mathtt{t}}(c_1, c_2\colon bool/c_3\colon bool) \triangleq\ & ((c_1(\mathtt{t}), c_2(\mathtt{t})/c_3(\mathtt{t}))+ \\
& (c_1(\mathtt{t}), c_2(\mathtt{f})/c_3(\mathtt{t}))+ \\
& (c_1(\mathtt{f}), c_2(\mathtt{t})/c_3(\mathtt{t}))) :: NorDel_{\mathtt{f}}(c_1, c_2/c_3)+ \\
& (c_1(\mathtt{f}), c_2(\mathtt{f})/c_3(\mathtt{t})) :: NorDel_{\mathtt{t}}(c_1, c_2/c_3)
\end{aligned}$$

Despite having distinct process terms as definitions, $NorDel_p$ and $NorDel_{\mathtt{f}}$ are bisimilar because they describe identical transition systems, as shown in Figure 2.3.

Determinism upon Input/Output Actions

Apart from the determinism defined over input actions for EPA processes, determinacy of processes can also be defined over pairs of input/output actions. In order to simplify future definitions, let us define the concepts of processes deterministically defined for a particular action, and determinism of processes upon the synchronous input/output actions:

Definition 2.2 *A process P is deterministically defined for action α (denoted as $\mathcal{D}!_\alpha(P)$) if whenever $P \xrightarrow{\alpha} P'$ and $P \xrightarrow{\alpha} P''$ then $P' \sim P''$.*

[5] A classification of equivalence relations is found in [Pom85, vG90].

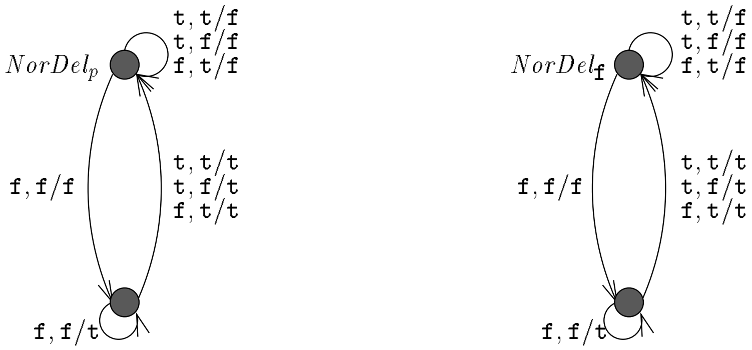

Figure 2.3: Bisimilar Processes

Definition 2.3 *Determinism over input/output action is a property defined over processes such that if a process P is deterministic on input/output actions (denoted as $\mathcal{D}!(P)$), then for every "proper" action $\alpha \in Act$: whenever $P \xrightarrow{\alpha} P'$, $\mathcal{D}!_\alpha(P)$ and $\mathcal{D}!(P')$.*

A process is deterministic on input/output actions if its entire communication with the environment is deterministically defined. The whole communication is defined by the stimulus the environment gives to such process, together with the response it is waiting for. In this sense, processes are deterministic when there exists only one choice for each entire communication, instead of being deterministic when it gives a predictable response for each input stimulus.

Notice that a process can be deterministic for an input/output action and still be nondeterministic if this particular action coincides in inputs with another action(s). For instance, a process S that perform the initial actions

$$(c_1(0)/c_3(1), c_2(0))$$
$$(c_1(0)/c_3(0), c_2(1))$$

is nondeterministic because it has a double choice for the input action $c_1(0)$ but, at the same time, it is deterministically defined on input/output actions.

2.4 Adding Combinators to EPA

Despite EPA being able to represent hierarchical structure and behaviour, further combinators are now defined for EPA in order to facilitate some definitions presented in this thesis. In particular, we introduce a new combinator to abstract from internal structure of processes, *restriction to channels*, and to abstract upon the set of actions a process can undertake, *restriction to actions*.

Restriction to Channels

The hiding combinator makes a set of channels unobservable. Nevertheless, in certain cases, an operator that allows a process to be restricted to a set of channels, and then hide the other channels, is of great use. Definition of "channel restricted bisimulation" (it is further defined in Section 4.2.4), for example, becomes far simpler when a restriction to channels operator is used.

Considering C as a set of typed channel names ($C \in ChanNames$), a semantics for *restriction to channels* operator is defined as:

$$\boxed{\textsf{RestrChannels}}\ \frac{P \xrightarrow{\alpha} P'}{P \restriction_c C \xrightarrow{res\text{-}ch(\alpha,\, C)} P' \restriction_c C}$$

$res\text{-}ch\colon\ Act \times \mathcal{F}(ChanNames) \rightarrow Act$
$res\text{-}ch(\epsilon, C) = \epsilon$
$res\text{-}ch(\boxtimes, C) = \boxtimes$
$res\text{-}ch(i/o, C) = (res\text{-}io\text{-}ch(i, C)/res\text{-}io\text{-}ch(o, C))$

$res\text{-}io\text{-}ch\colon IOAct \times \mathcal{F}(ChanNames) \rightarrow IOAct$
$res\text{-}io\text{-}ch(\epsilon, C) = \epsilon$
$res\text{-}io\text{-}ch(\boxtimes, C) = \boxtimes$
$res\text{-}io\text{-}ch(c(v), C) = res\text{-}prim\text{-}ch(c(v), C)$
$res\text{-}io\text{-}ch(io_1 \oplus io_2, C) = res\text{-}io\text{-}ch(io_1, C) \oplus res\text{-}io\text{-}ch(io_2, C)$

$res\text{-}prim\text{-}ch : pIOAct \times \mathcal{F}(ChanNames) \rightarrow pIOAct \cup \{\epsilon\}$

$$res\text{-}prim\text{-}ch(c(v), C) \ \underline{\triangle}\ \begin{array}{l} \textsf{if } chan\text{-}of(c(v)) \in C \\ \textsf{then } c(v) \\ \textsf{else } \epsilon \end{array}$$

Intuitively, $P \restriction_c C$ hides all subactions passing through the channels not included in the given set C. Notice the similarity between this operator and *hiding*. Both are based on the idea of abstracting on the structure by making some channels unobservable, they consider however distinct sets of channels to make such an abstraction. The process nor–gate–delay, for example, can be redefined using the restriction to channels operator:

$$\textsf{process } NorDel'_p(c_1, c_2\colon\ bool/c_3\colon\ bool)\ \underline{\triangle}$$
$$\Big(\ Nor_p(c_1, c_2/c_4) \mid Del_p(c_4/c_3)\ \Big) \restriction_c \{c_1, c_2, c_3\}.$$

Certain properties on restricted processes and restricted actions turns out with definition of the restriction to channels combinator:

Proposition 2.1 *Whenever* $P \restriction_c C \xrightarrow{\alpha'} P' \restriction_c C$ *then* $res\text{-}ch(\alpha', C) = \alpha'$.

Proposition 2.2 *For every "proper" action* α, *whenever* $P \restriction_c C \xrightarrow{\alpha'} P' \restriction_c C$ *then* $P \xrightarrow{\alpha} P'$ *such that* $res\text{-}ch(\alpha, C) = \alpha'$

These two propositions are trivially proved by definition of $\restriction_c$: Proposition 2.1 is obvious since restriction to channels is applied to all single actions performed by P; Proposition 2.2 is also trivial, each P's action has a corresponding action in $P \restriction_c C$ which is a restricted version of the P's action.

Restriction to Actions

The fact that EPA has structured actions, operators can be applied to the various elements: the whole process, such as process call, composition, binary summation and process choice; only channels, such as hiding and now restriction to channels; or even actions like the prefixing operator. *Abstraction* of structures has been obtained in EPA with application of the hiding combinators (hiding and restriction to channels), but partial definition of processes requires hiding input/output actions instead of channels. Given a process, we want to define a new process which only responds to a subset of actions from the original process.

The operator restriction to actions is defined as follows:

$$\boxed{\mathsf{RestrActions}}\ \frac{P \xrightarrow{\alpha} P', \quad \alpha \in \mathcal{A}, \quad \mathcal{A} \subseteq Act}{P \restriction_a \mathcal{A} \xrightarrow{\alpha} P' \restriction_a \mathcal{A}}$$

For instance, a nor–gate that does not respond when the value $\mathtt{t}$ is given to both of its input channels, can be defined as follows:

$$\begin{aligned}\mathsf{process}\ \mathit{Nor_p\text{-}Rest}(c_1, c_2\colon \mathit{bool}/c_3\colon \mathit{bool}) \ \triangleq\ \mathit{Nor_p}(c_1, c_2/c_4) \restriction_a \{&(c_1(\mathtt{t}), c_2(\mathtt{f})/c_3(\mathtt{f})),\\ &(c_1(\mathtt{f}), c_2(\mathtt{t})/c_3(\mathtt{f})),\\ &(c_1(\mathtt{f}), c_2(\mathtt{f})/c_3(\mathtt{t}))\}\end{aligned}$$

Roughly speaking, $\mathit{Nor_p\text{-}Rest}$, is able to behave identically to $\mathit{Nor_p}$ except that it deadlocks for the input action $(c_1(\mathtt{t}), c_2(\mathtt{t}))$.

2.5 Algebraic Manipulation of Synchronous Processes

Using bisimulation as the equivalence relation for EPA, a set of algebraic laws can be defined over EPA process terms. A large number of laws can be defined to show bisimilarity of process terms, but we will only present a small number of them. Furthermore, only intuitive arguments shall be used for validating such laws.

The laws are presented in blocks of particular combinators. First, the "pure behavioural" combinators are shown and then the composition and hiding operators are presented.

The basic laws for binary summation are as follows:

Proposition 2.3 *Binary Summation Laws:*

1. $(P + Q) \sim (Q + P)$
2. $(P + (Q + R)) \sim ((P + Q) + R)$
3. $(P + P) \sim P$
4. $(P + \mathsf{Stop}) \sim P$

Commutativity of summation operator is straightforward by the fact that this combinator allows for exclusive choice of an initial action from one or another processes. The first law states commutativity of summation. Associativity of this operator is also proved using the same argument. In the third law, all alternative actions are restricted to the P's initial actions and then it evolves into bisimilar processes after performing identical actions. So, the alternative actions for P can only generate P. In the fourth law, considering that Stop is unable to undertake any action, its summation with P is only able to perform the initial actions defined for P and then evolve into P's following processes and so on. It is indeed bisimilar to P.

Let us now establish a few laws involving the *zero–action*:

Proposition 2.4 *Zero Action Laws:*

1. $(\boxtimes :: P) \sim \mathsf{Stop}$
2. $(\square\, x\colon ty \ \cdot\ \boxtimes :: P) \sim \mathsf{Stop}$

3. $(\alpha :: \boxtimes :: P) \sim (\alpha :: \mathsf{Stop})$

4. $(P + (\boxtimes :: Q)) \sim P$

Considering that the zero–action has no behavioural value, any process preceded by a zero–action is bisimilar to Stop. The first two laws are concerned with a zero–action as the initial action of process terms. As a result, these process terms are equivalent to Stop. The third law presents a process term where an initial action is followed by a zero–action, it is then equivalent to a process that only performs such an initial action and then reaches Stop. In the fourth law, summation of a process term with another one initialized by a zero–action (process Stop) is bisimilar to the first process term because of the summation law given in Proposition 2.3-*4*.

Certain basic laws can also be defined for process terms involving the composition operator:

Proposition 2.5 *Composition Laws:*

1. $(P \mid Q) \sim (Q \mid P)$
2. $(P \mid (Q \mid R)) \sim ((P \mid Q) \mid R)$
3. $(P \mid (Q + R)) \sim ((P + Q) \mid (P + R))$
4. $(P \mid \mathsf{Stop}) \sim \mathsf{Stop}$
5. $((\alpha :: P) \mid (\beta :: Q)) \sim ((\alpha \star \beta) :: (Q \mid P))$

Commutativity and associativity of parallel composition can be proved by using commutativity of actions summation. Since the action product is defined over subtraction and summation of input and output actions, composition of processes is also commutative and associative. Distributivity of composition over summation is also trivially proved using the arguments: synchronous combination of all initial actions and exclusive action choice of the binary summation.

Considering synchronization of EPA composition, whenever Stop is composed with any other process, it can only result in Stop; no combination of actions is possible. Also, due to synchronization of component processes, the initial actions of the process terms are combined and then the following processes are composed as stated in law 2.5–*5*.

The hiding operator must also be given several laws:

Proposition 2.6 *Hiding Laws:*

1.	$P\backslash C \sim P$	*if*	$\mathsf{Chs}(P) \cap C = \emptyset$
2.	$P\backslash C_1\backslash C_2 \sim P\backslash(C_1 \cup C_2)$		
3.	$(P \mid Q)\backslash C \sim (P\backslash C) \mid (Q\backslash C)$	*if*	$\mathsf{Chs}(P) \cap \mathsf{Chs}(Q) \cap C = \emptyset$
4.	$(P + Q)\backslash C \sim P\backslash C + Q\backslash C$		
5.	$(P\backslash C) \mid Q \sim (P \mid Q)\backslash C$	*if*	$\mathsf{Chs}(Q) \cap C = \emptyset$

The first and second laws are trivial by the semantics of the hiding operator. Actions can only be combined in a proper action if they are conflict free. Considering the side condition of law 2.6–*3*, conflict freedom between actions of P and Q will be maintained even when the channels in C are hidden. So, the law 2.6–*3* holds.

The fourth law is also trivially proved by the exclusive choice of binary summation. Based on synchronism of the parallel composition operator and disjointness between Q channels and C, the fifth law is also trivially proved: conflict freedom of actions of these processes is identical on both hand sides considering the side condition.

Respecting the contrast between hiding and restriction to channels operators, a few laws, based on laws for the hiding combinator, can be defined for restriction to channels:

Proposition 2.7 *Restriction to Channels Laws:*

1. $P \restriction_c C \;\sim\; P$ $\quad$ *if* $\;\mathsf{Chs}(P) \subseteq C$
2. $P \restriction_c C_1 \restriction_c C_2 \sim\; P \restriction_c (C_1 \cap C_2)$
3. $(P \mid Q) \restriction_c C \;\sim\; (P \restriction_c C) \mid (Q \restriction_c C)$ $\quad$ *if* $\;\mathsf{Chs}(P) \cap \mathsf{Chs}(Q) \subseteq C$
4. $(P + Q) \restriction_c C \;\sim\; (P \restriction_c C) + (P \restriction_c C)$

The first law, considering the side–condition, is trivial. If channels of P are in C, all the P channels are observable with the application of the restricted to channels combinator and then, this process term is bisimilar to P. The successive application of restriction to channels combinator makes observable only channels that are elements of all those restricted channels, as stated in the second law. The third and fourth laws are also trivially proved. Composition of processes depends on conflict freedom of actions. The third law is proved by the argument that the conflict freedom of actions of both processes is maintained over both process terms, considering the side–condition. Using the argument of exclusive choice of binary summation, the fourth law can also be proved.

The other combinator concerned with abstraction on process definition is restriction to actions. A few laws are also defined for this combinator:

Proposition 2.8 *Restriction to Actions Laws:*

1. $P \restriction_a \mathcal{A} \;\sim\; P$ $\quad$ *if* $\;\mathsf{Sort}\, P \subseteq \mathcal{A}$
2. $P \restriction_a \mathcal{A}_1 \restriction_a \mathcal{A}_2 \sim\; P \restriction_a (\mathcal{A}_1 \cap \mathcal{A}_2)$
3. $(P + Q) \restriction_a \mathcal{A} \;\sim\; (P \restriction_a \mathcal{A}) + (P \restriction_c \mathcal{A})$

If all actions of a process are included in the set of actions to which such a process is restricted, then all its actions are observable and this process term is bisimilar to the unrestricted process, as stated in the first law. The second law states that successive restriction to actions makes only actions that are elements of all those sets observable, it is trivially proved by definition of the combinator restriction to actions. Exclusive choice of actions stated in the binary summation and definition of the restriction to actions operator are used as arguments to prove the third law.

The laws presented in this section represent a narrow spectrum of the number of laws that can be defined for all EPA combinators. They have been presented here to give a notion of rephrasing EPA process terms rather than establishing a complete characterization of algebraic manipulation of processes. In fact, all this chapter has been devoted to presenting a model that provides mathematical foundations for formally reusing hardware design rather than a full exploitation of process algebras, or even EPA. Many of the EPA features are omitted in favor of concentrating upon definitions that are used throughout this thesis; all the hardware components in the present work shall be represented by EPA processes.

Chapter 3

An Interface Approach for Reuse

3.1 Introduction

We have already discussed, in Chapter 1, that reuse can be approached in different ways, and the approach to be adopted depends on the information that is going to be reused. In order to reuse hardware design, we must first define which kind of information is expected to be reused and which approach is taken to make such reuse feasible.

This chapter is devoted to studying some behavioural similarities between hardware components that make us believe they can be efficiently reused. Besides that, the framework we adopt to reuse hardware components will be determined.

In the first part, some case studies on variations of traffic light controllers are presented and an informal discussion on their similarities is conducted. Such a discussion aims at defining a range of similarities between processes behaviour that make us believe some existing components can be reused to implement new ones.

The second part defines the information we want to reuse and which approach is adopted to achieve that. Once our reuse framework is presented, the operational problems to be solved for the adopted approach and possible solutions for them will be discussed finally.

3.2 Case Studies

We have already expressed our desire to reuse hardware, but what kind of reuse is feasible in the hardware application domain? In this section, we identify some problems that exist when attempting to reuse hardware components.

To clarify the nature of the problem we are addressing, let us define some processes to be constructed and, other, existing components which could eventually be reused to implement these new elements. Our samples range over variants of traffic light controllers. A couple of traffic light controllers are needed to be developed when a particular component has already been designed.

At an abstract level, a traffic light controller is a process that cycles through a trivial sequence of three light colours: green, amber and red, and then restarts the sequence[1]. Some variations on sequences of light colours can be obtained if a particular colour is given priority. A traffic light controller which gives priority to green, for example, besides performing the trivial sequence of light colours, might signal the short sequence [green,amber] and then go back to green without going for red, or even stay on green indefinitely. Similarly, several other variations could be obtained if amber or red lights are given priority.

That is a very abstract specification of traffic light controllers. In general, this kind of system receives a signal to make it change the light colour. So, at a more detailed level, the traffic light controller must be defined with at least one port of communication through which a "change colour" event is input.

[1] This sequence of lights is concerned with American standards, instead of the British one, because the examples presented here are originated from [PW87].

We could have, for example, a traffic light controller that depends on two events to change the light colour: a signal for time units, and a signal to communicate the occurrence of an event. This traffic light controller has some predefined intervals during which a colour must remain and after which such a colour can be changed. A time counter synchronized with a master clock is necessary for signaling these time units. At the same time, this time counter must receive a signal to be initialized every time the traffic light has changed the colour. Additionally, a sensor is wired to this traffic light controller in order to signal the other incoming event. Thus, this traffic light controller, TLC_1, is an agent that interacts with both a sensor ($Sens$) and a time counter ($Tcnt$). A structural definition for these agents is shown in Figure 3.1.

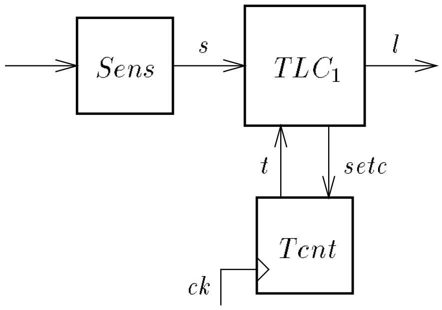

Figure 3.1: A Traffic Light Controller Interacting with a Time Counter and a Sensor

For this particular traffic light controller, the sensor must detect the presence of cars waiting to cross the road. The sensor communicates, via the channel s, the signal car when a car is detected and $\overline{car}$ when this event does not occur. TLC_1 communicates with the time counter through two channels: $setc$ to initialize such counter when a light colour is changed, and t to receive a time unit from the counter. To some extent, a distinct interval of time could be set for each state through which the traffic light controller is cycled. But, to make it simpler, let us settle the interval t_f to control the "traffic flow", and t_s to control the "safety" time of the system (the light, amber, for example, is given for safety, then t_s is concerned with the time interval during which this kind of light colour must be signaled). And finally, the light colour is communicated to the environment through channel l. The sorts of these incoming and outcoming events are as follows:

$$\text{type } colour \triangleq \{G, A, R\}$$
$$\text{type } scount \triangleq \{st, \overline{st}\}$$
$$\text{type } intervs \triangleq \{t_s, \overline{t_s}, t_f, \overline{t_f}\}$$
$$\text{type } sensor \triangleq \{car, \overline{car}\}$$

We have settled above that the time counter outputs time units run by a master clock. The time intervals for safety and traffic flow are interpreted by the traffic light controller as a short representation for those time units. If the time unit signaled by the time counter is within these intervals, they are signaled positive (t_s, t_f) otherwise they are negated ($\overline{t_s}$, $\overline{t_f}$). So, by the time the time counter is initialized, both intervals are positive and they will only start to be negated when the counter produces a time unit out of those intervals.

In order to concentrate on the specification of the traffic light controller, let us assume that the time counter and sensor processes are already defined.

For this traffic light controller, the green light is given priority and the sequences of light colours are as defined above: green, amber and red; green and amber; or only green. The light must remain green for a minimum interval t_f and should only cycle to amber with the presence of cars waiting to cross the road, even if this interval has expired. Furthermore, whenever a car is no longer detected, the system must choose the shortest sequence of cycles to reach the green light.

The maximum time the light can remain amber is t_s. After that, it must cycle to another colour in the predefined sequence. The maximum period the light can remain red to control the traffic flow is t_f. So, even if cars are still detected when t_f has expired for the red light, the traffic light controller must be cycled to another state.

A specification of this traffic light controller is given as follows:

$$
\begin{aligned}
&\textsf{process } TLC_1(s\colon sensor, t\colon intervs/l\colon colour, setc\colon stcount) \triangleq \\
&\qquad (s(car), t(\overline{t_f})/l(A), setc(st)) :: Amb_1(s, t/l, setc) + \\
&\qquad (s(car), t(t_f)/l(G), setc(\overline{st})) :: TLC_1(s, t/l, setc) + \\
&\qquad (s(\overline{car}), t(\overline{t_f})/l(G), setc(\overline{st})) :: TLC_1(s, t/l, setc) + \\
&\qquad (s(\overline{car}), t(t_f)/l(G), setc(\overline{st})) :: TLC_1(s, t/l, setc)
\end{aligned}
$$

$$
\begin{aligned}
&\textsf{process } Amb_1(s\colon sensor, t\colon intervs/l\colon colour, setc\colon stcount) \triangleq \\
&\qquad (s(car), t(\overline{t_s})/l(R), setc(st)) :: FRed_1(s, t/l, setc) + \\
&\qquad (s(car), t(t_s)/l(A), setc(\overline{st})) :: Amb_1(s, t/l, setc) + \\
&\qquad (s(\overline{car}), t(\overline{t_s})/l(G), setc(st)) :: TLC_1(s, t/l, setc) + \\
&\qquad (s(\overline{car}), t(t_s)/l(G), setc(st)) :: TLC_1(s, t/l, setc)
\end{aligned}
$$

$$
\begin{aligned}
&\textsf{process } FRed_1(s\colon sensor, t\colon intervs/l\colon colour, setc\colon stcount) \triangleq \\
&\qquad (s(car), t(t_f)/l(R), setc(\overline{st})) :: FRed_1(s, t/l, setc) + \\
&\qquad (s(car), t(\overline{t_f})/l(R), setc(st)) :: SRed_1(s, t/l, setc) + \\
&\qquad (s(\overline{car}), t(\overline{t_f})/l(R), setc(st)) :: SRed_1(s, t/l, setc) + \\
&\qquad (s(\overline{car}), t(t_f)/l(R), setc(st)) :: SRed_1(s, t/l, setc)
\end{aligned}
$$

$$
\begin{aligned}
&\textsf{process } SRed_1(s\colon sensor, t\colon intervs/l\colon colour, setc\colon stcount) \triangleq \\
&\qquad (s(car), t(\overline{t_s})/l(G), setc(st)) :: TLC_1(s, t/l, setc) + \\
&\qquad (s(\overline{car}), t(\overline{t_s})/l(G), setc(st)) :: TLC_1(s, t/l, setc) + \\
&\qquad (s(car), t(t_s)/l(R), setc(\overline{st})) :: SRed_1(s, t/l, setc) + \\
&\qquad (s(\overline{car}), t(t_s)/l(R), setc(\overline{st})) :: SRed_1(s, t/l, setc)
\end{aligned}
$$

Several points must be clarified as to the behaviour of this traffic light controller. Thus, if the green light is given priority, then, when the amber light is signaled, the colour can be turned back to green without cycling through red. It might only occur with the absence of cars on the opposite side when the light is signaled amber (process Amb_1). It would look bizarre if the sensor detected the presence of cars on the opposite side, and then detected no cars before the light has turned red. However, a faulty sensor could signal an unexpected signal, and besides that, a car detected on the opposite side could eventually go forward without obeying its red light. This traffic light, in particular, also responds to these "abnormal cases".

The other idea to be pointed out is why this traffic light controller keeps the light red during a traffic flow and a safety interval. A traffic light controller for a single light like this one is supposed to have an opposite traffic light running in parallel, such that their light colours do not overlap. By the time the opposite traffic light is green, this light must be red. In order to allow the opposite light to cycle through amber to red, this traffic light must remain red for a safety period. So, the traffic light must be kept red for both intervals.

Let us assume this traffic light controller for a single light has already been designed and is available as a library component. Now, suppose another traffic light controller for double lights is required. Could this existing component be reused to implement the required element? In the following sections, we show some other variations of traffic light controllers and discuss the possibility of reusing existing components.

3.2.1 A Major/Minor Roads Traffic Light Controller

Suppose we want to develop a hardware component to control a traffic light at a junction of a major and minor road[2]. The major road is very busy and the traffic light gives opportunity to cars coming through the minor road to cross the major road (Figure 3.2). The minor road has sensors that activate the traffic light when a car is detected (the major road does not need any sensors because, by default, its light is green). In the absence of cars waiting on the minor road, the major road traffic light will remain green with the minor road light red. The latter becomes green only with the presence of cars.

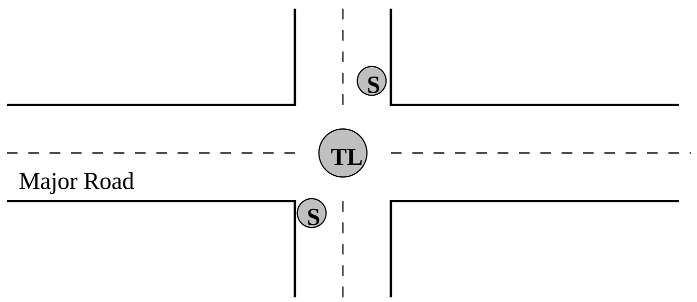

Figure 3.2: Major/Minor Roads–Road Traffic Light

Thus, the major road traffic light is given priority, but not to the extent that the minor road traffic can be stalled indefinitely. The major road light can only cycle through amber to red if it has been green for a pre–established minimal period of time. Also, the minor road light can only remain green for a maximum period while the sensors indicate the presence of cars. After this period, it cycles through amber to red and the major road light turns green. In order to control the time cars are waiting on the minor road and also how long a red light is signaled to the major road, a time counter must be introduced.

The external environment of this process is ready to activate it by indicating the presence of cars produced by the minor road sensors and giving a time element to control maximal or minimal periods in which each light colour is signaled. On the other hand, the environment receives a response from this process in order to change the colors of the traffic lights. To illustrate this environment, its structural definition is given as in Figure 3.3. The communications with sensors and the time counter are as in the previous component. Unlike the previous component, l_{mj} and l_{mn} signal the colour for the major and minor roads traffic lights respectively.

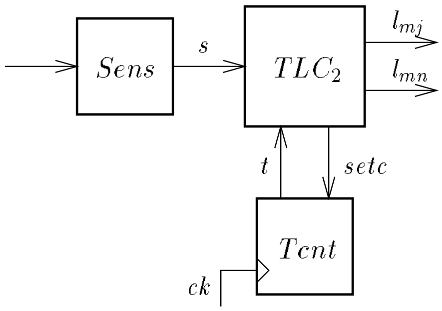

Figure 3.3: The Traffic Light Controller and its Environment

The control of the traffic signals involves four times: the minimum time the major road light is green,

[2]This problem is a variant of the Traffic Light Controller for a highway and a farm presented by F. P. Prosser and D. E. Winkel in *The Art of Digital Design* [PW87].

the maximum time the minor road light is green, and the times when the major or the minor roads are amber. As with the previously defined traffic light controller, the time interval to maintain both lights amber is the safety interval $(t_s, \overline{t_s})$, and the minimum or the maximum intervals for green are specified by the traffic flow interval $(t_f, \overline{t_f})$.

Let us also make some other assumptions about these two intervals. The "safety interval" is given to avoid the opposite traffic light turning green when there is a car still crossing the critical area. Obviously, this safety interval must be shorter than the traffic flow interval. This means that whenever t_s is positive, t_f is also $(t_s \Rightarrow t_f)$, and consequently $\overline{t_f} \Rightarrow \overline{t_s}$, but there exists a subinterval in which t_s is negated $(\overline{t_s})$ and t_f is positive.

This major/minor roads traffic light controller is specified as follows (a transition graph for this process is included in Appendix D):

$$
\begin{aligned}
&\textsf{process } TLC_2(s\colon sensor, t\colon intervs/l_{mj}, l_{mn}\colon colour, setc\colon stcount) \triangleq \\
&\qquad (s(car), t(\overline{t_f})/l_{mj}(A), l_{mn}(R), setc(st)) :: Amb_2(s, t/l_{mj}, l_{mn}, setc) + \\
&\qquad (s(car), t(t_f)/l_{mj}(G), l_{mn}(R), setc(\overline{st})) :: TLC_2(s, t/l_{mj}, l_{mn}, setc) + \\
&\qquad (s(\overline{car}), t(\overline{t_f})/l_{mj}(G), l_{mn}(R), setc(\overline{st})) :: TLC_2(s, t/l_{mj}, l_{mn}, setc) + \\
&\qquad (s(\overline{car}), t(t_f)/l_{mj}(G), l_{mn}(R), setc(\overline{st})) :: TLC_2(s, t/l_{mj}, l_{mn}, setc)
\end{aligned}
$$

$$
\begin{aligned}
&\textsf{process } Amb_2(s\colon sensor, t\colon intervs/l_{mj}, l_{mn}\colon colour, setc\colon stcount) \triangleq \\
&\qquad (s(car), t(\overline{t_s})/l_{mj}(R), l_{mn}(G), setc(st)) :: FRed_2(s, t/l_{mj}, lmn, setc) + \\
&\qquad (s(car), t(t_s)/l_{mj}(A), l_{mn}(R), setc(\overline{st})) :: Amb_2(s, t/l_{mj}, l_{mn}, setc) + \\
&\qquad (s(\overline{car}), t(\overline{t_s})/l_{mj}(G), l_{mn}(R), setc(st)) :: TLC_2(s, t/l_{mj}, l_{mn}, setc) + \\
&\qquad (s(\overline{car}), t(t_s)/l_{mj}(G), l_{mn}(R), setc(st)) :: TLC_2(s, t/l_{mj}, l_{mn}, setc)
\end{aligned}
$$

$$
\begin{aligned}
&\textsf{process } FRed_2(s\colon sensor, t\colon intervs/l_{mj}, l_{mn}\colon colour, setc\colon stcount) \triangleq \\
&\qquad (s(car), t(t_f)/l_{mj}(R), l_{mn}(G), setc(\overline{st})) :: FRed_2(s, t/l_{mj}, l_{mn}, setc) + \\
&\qquad (s(car), t(\overline{t_f})/l_{mj}(R), l_{mn}(A), setc(st)) :: SRed_2(s, t/l_{mj}, l_{mn}, setc) + \\
&\qquad (s(\overline{car}), t(\overline{t_f})/l_{mj}(R), l_{mn}(A), setc(st)) :: SRed_2(s, t/l_{mj}, l_{mn}, setc) + \\
&\qquad (s(\overline{car}), t(t_f)/l_{mj}(R), l_{mn}(A), setc(st)) :: SRed_2(s, t/l_{mj}, l_{mn}, setc)
\end{aligned}
$$

$$
\begin{aligned}
&\textsf{process } SRed_2(s\colon sensor, t\colon intervs/l_{mj}, l_{mn}\colon colour, setc\colon stcount) \triangleq \\
&\qquad (s(car), t(\overline{t_s})/l_{mj}(G), l_{mn}(R), setc(st)) :: TLC_2(s, t/l_{mj}, l_{mn}, setc) + \\
&\qquad (s(\overline{car}), t(\overline{t_s})/l_{mj}(G), l_{mn}(R), setc(st)) :: TLC_2(s, t/l_{mj}, l_{mn}, setc) + \\
&\qquad (s(car), t(t_s)/l_{mj}(R), l_{mn}(A), setc(\overline{st})) :: SRed_2(s, t/l_{mj}, l_{mn}, setc) + \\
&\qquad (s(\overline{car}), t(t_s)/l_{mj}(R), l_{mn}(A), setc(\overline{st})) :: SRed_2(s, t/l_{mj}, l_{mn}, setc)
\end{aligned}
$$

If the top–down approach is taken to develop this system, its top partitioning would obviously be defined as two processes running in parallel: the major road and the minor road processes. In this case, we could develop these two processes in parallel and then compose them synchronously in order to obtain the whole system. The major road process is a controller for a single traffic light which gives priority to green and is activated by a sensor and a time counter. In fact, the major road process is an instance of the library component previously defined:

$$\textsf{process } TLC_{mj}(s\colon sensor, t\colon intervs/l_{mj}\colon colour, setc\colon stcount) \triangleq TLC_1(s, t/l_{mj}, setc)$$

By contrast, the minor road traffic light controller gives priority to red and can be defined as a four state process which cycles through: red light for the traffic flow, red light for safety, green and amber.

To construct the major/minor roads traffic light controller, the existing component could be reused to implement the major road process and only the minor road process is required to be designed. But, is the minor road process the minimal component necessary to achieve the behaviour of the whole system? This is one of the questions we have to find out throughout this work.

3.2.2 A Major/Minor Roads Traffic Light Controller in a Perfect Environment

Another interesting case of reuse is when a new process being defined performs only a subset of actions of the existing process.

Suppose that at this point the major/minor roads traffic light controller has been designed and is available as a library component. A new major/minor roads traffic light controller is required with similar properties of the previous component, except that only the trivial sequence of lights [green, amber, red] is permitted. That is, whenever the major road traffic light is turned amber, it must cycle to red instead of also having the option to move back to green. This means that this new element does not respond to those "abnormal cases" cited above because a perfect environment is considered.

This new traffic light controller is defined as follows (a transition graph for this process is also included in Appendix D):

$$\begin{aligned}
&\textsf{process } TLC_{perf}(s\colon sensor, t\colon intervs/l_{mj}, l_{mn}\colon colour, setc\colon stcount) \triangleq \\
&\quad (s(car), t(\overline{t_f})/l_{mj}(A), l_{mn}(R), setc(st)) :: Amb_{perf}(s, t/l_{mj}, l_{mn}, setc) + \\
&\quad (s(car), t(t_f)/l_{mj}(G), l_{mn}(R), setc(\overline{st})) :: TLC_{perf}(s, t/l_{mj}, l_{mn}, setc) + \\
&\quad (s(\overline{car}), t(\overline{t_f})/l_{mj}(G), l_{mn}(R), setc(\overline{st})) :: TLC_{perf}(s, t/l_{mj}, l_{mn}, setc) + \\
&\quad (s(\overline{car}), t(t_f)/l_{mj}(G), l_{mn}(R), setc(\overline{st})) :: TLC_{perf}(s, t/l_{mj}, l_{mn}, setc)
\end{aligned}$$

$$\begin{aligned}
&\textsf{process } Amb_{perf}(s\colon sensor, t\colon intervs/l_{mj}, l_{mn}\colon colour, setc\colon stcount) \triangleq \\
&\quad (s(car), t(\overline{t_s})/l_{mj}(R), l_{mn}(G), setc(st)) :: FRed_{perf}(s, t/l_{mj}, lmn, setc) + \\
&\quad (s(car), t(t_s)/l_{mj}(A), l_{mn}(R), setc(\overline{st})) :: Amb_{perf}(s, t/l_{mj}, l_{mn}, setc)
\end{aligned}$$

$$\begin{aligned}
&\textsf{process } FRed_{perf}(s\colon sensor, t\colon intervs/l_{mj}, l_{mn}\colon colour, setc\colon stcount) \triangleq \\
&\quad (s(car), t(t_f)/l_{mj}(R), l_{mn}(G), setc(\overline{st})) :: FRed_{perf}(s, t/l_{mj}, l_{mn}, setc) + \\
&\quad (s(car), t(\overline{t_f})/l_{mj}(R), l_{mn}(A), setc(st)) :: SRed_{perf}(s, t/l_{mj}, l_{mn}, setc) + \\
&\quad (s(\overline{car}), t(\overline{t_f})/l_{mj}(R), l_{mn}(A), setc(st)) :: SRed_{perf}(s, t/l_{mj}, l_{mn}, setc) + \\
&\quad (s(\overline{car}), t(t_f)/l_{mj}(R), l_{mn}(A), setc(st)) :: SRed_{perf}(s, t/l_{mj}, l_{mn}, setc)
\end{aligned}$$

$$\begin{aligned}
&\textsf{process } SRed_{perf}(s\colon sensor, t\colon intervs/l_{mj}, l_{mn}\colon colour, setc\colon stcount) \triangleq \\
&\quad (s(car), t(\overline{t_s})/l_{mj}(G), l_{mn}(R), setc(st)) :: TLC_{perf}(s, t/l_{mj}, l_{mn}, setc) + \\
&\quad (s(\overline{car}), t(\overline{t_s})/l_{mj}(G), l_{mn}(R), setc(st)) :: TLC_{perf}(s, t/l_{mj}, l_{mn}, setc) + \\
&\quad (s(car), t(t_s)/l_{mj}(R), l_{mn}(A), setc(\overline{st})) :: SRed_{perf}(s, t/l_{mj}, l_{mn}, setc) + \\
&\quad (s(\overline{car}), t(t_s)/l_{mj}(R), l_{mn}(A), setc(\overline{st})) :: SRed_{perf}(s, t/l_{mj}, l_{mn}, setc)
\end{aligned}$$

Notice that this process only differs from the previous one when it is signaling amber. It performs indeed a subset of actions of TLC_2. Considering the behavioural similarity between these two processes, we want to reuse the previous traffic light controller to simulate all the actions this new one is able to perform. But, how could we reuse such a process and still guarantee behaviour of this new traffic light controller? This is another reuse issue we want to address in this work.

3.2.3 A Pedestrian/Road Traffic Light Controller

Now, suppose a new problem of traffic light control is given to us: a busy road must have a pedestrian traffic light in order to make it safe for people to cross this road (Figure 3.4). As usual, the pedestrian traffic light is activated by a button. The road is given priority and it is only going to cycle through amber to red in the event of a pedestrian pressing the button. Furthermore, there is a minimum period in which the road light must remain green.

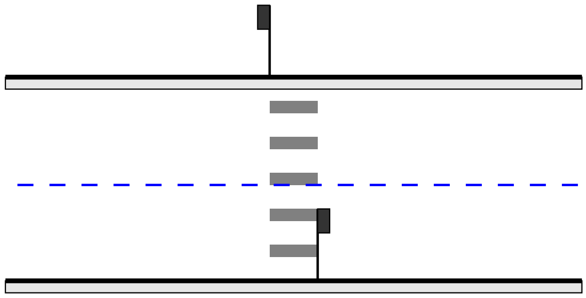

Figure 3.4: Pedestrian/Road Traffic Light

The road traffic light cycles through three colours: green (G), amber (A) and red (R). The pedestrian traffic light, however, cycles through different colours: red, green and a flashing green (FG). The time the pedestrian light is green is predefined by an interval for traffic flow (t_f). After that, the pedestrian light flashes for an interval t_s for safety and then turns back to red. In this problem, the safety time is denoted by the interval the road light is amber, and the time the pedestrian light is flashing. Also, the minimum time the road traffic light is green is given by t_f. Obviously, a time counter is also necessary to set these traffic flow and safety intervals, and a button signal ($ped, \overline{ped}$) must also be given as input for this process. Communication between this traffic light controller and the other elements are shown in Figure 3.5.

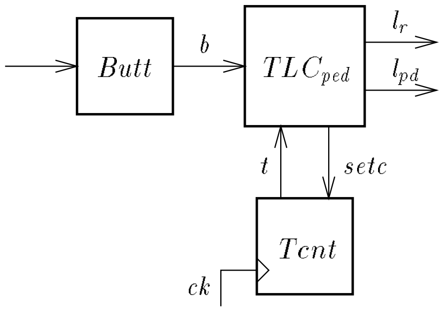

Figure 3.5: Communication of the Pedestrian/Road Traffic Light

The behavioural specification of this traffic light controller is given as follows (its transition graph is included in Appendix D):

type $colour_p \triangleq \{G, FG, R\}$

type $button \triangleq \{ped, \overline{ped}\}$

process $TLC_{ped}(b\colon\ button, t\colon\ intervs/l_r\colon\ colour, l_{pd}\colon\ colour_p, setc\colon\ stcount) \triangleq$

$$(b(ped), t(\overline{t_f})/l_r(A), l_{pd}(R), setc(st)) :: Amb_{ped}(b, t/l_r, l_{pd}, setc) +$$
$$(b(ped), t(t_f)/l_r(G), l_{pd}(R), setc(\overline{st})) :: TLC_{ped}(b, t/l_r, l_{pd}, setc) +$$
$$(b(\overline{ped}), t(\overline{t_f})/l_r(G), l_{pd}(R), setc(\overline{st})) :: TLC_{ped}(b, t/l_r, l_{pd}, setc) +$$
$$(b(\overline{ped}), t(t_f)/l_r(G), l_{pd}(R), setc(\overline{st})) :: TLC_{ped}(b, t/l_r, l_{pd}, setc)$$

process $Amb_{ped}(s\colon\ sensor, t\colon\ intervs/l_r\colon\ colour, l_{pd}\colon\ colour_p, setc\colon\ stcount) \triangleq$

$$(b(ped), t(t_s)/l_r(A), l_{pd}(R), setc(\overline{st})) :: Amb_{ped}(b, t/l_r, l_{pd}, setc) +$$
$$(b(ped), t(\overline{t_s})/l_r(R), l_{pd}(G), setc(st)) :: FRed_{ped}(b, t/l_r, lpd, setc) +$$

process $FRed_{ped}(s\colon\ sensor, t\colon\ intervs/l_r\colon\ colour, l_{pd}\colon\ colour_p, setc\colon\ stcount) \triangleq$

$$(b(ped), t(t_f)/l_r(R), l_{pd}(G), setc(\overline{st})) :: FRed_{ped}(b, t/l_r, l_{pd}, setc) +$$
$$(b(\overline{ped}), t(t_f)/l_r(R), l_{pd}(G), setc(st)) :: FRed_{ped}(b, t/l_r, l_{pd}, setc) +$$
$$(b(ped), t(\overline{t_f})/l_r(R), l_{pd}(FG), setc(st)) :: SRed_{ped}(b, t/l_r, l_{pd}, setc) +$$
$$(b(\overline{ped}), t(\overline{t_f})/l_r(R), l_{pd}(FG), setc(st)) :: SRed_{ped}(b, t/l_r, l_{pd}, setc)$$

process $SRed_{ped}(s\colon\ sensor, t\colon\ intervs/l_r\colon\ colour, l_{pd}\colon\ colour_p, setc\colon\ stcount) \triangleq$

$$(b(ped), t(\overline{t_s})/l_r(G), l_{pd}(R), setc(st)) :: TLC_{ped}(b, t/l_r, l_{pd}, setc) +$$
$$(b(\overline{ped}), t(\overline{t_s})/l_r(G), l_{pd}(R), setc(st)) :: TLC_{ped}(b, t/l_r, l_{pd}, setc) +$$
$$(b(ped), t(t_s)/l_r(R), l_{pd}(FG), setc(\overline{st})) :: SRed_{ped}(b, t/l_r, l_{pd}, setc) +$$
$$(b(\overline{ped}), t(t_s)/l_r(R), l_{pd}(FG), setc(\overline{st})) :: SRed_{ped}(b, t/l_r, l_{pd}, setc)$$

Looking at the nature of the problems of the major/minor roads and pedestrian/road traffic light controllers, many similarities can be noticed. First of all, the major road and the pedestrian–road lights only cycle through amber to red with the presence of cars or pedestrians respectively. By the time the major road and pedestrian–road lights are green, they can only be turned amber when a minimal interval has expired and a car or a pedestrian are detected. Also, under the presence of their respective events these lights cycle through amber to red with a similar behaviour and so on. So, it is clear that these processes have some similarities in behaviour, but they are defined over distinct sorts. Even knowing these problems are defined over different sorts, could we reuse the major/minor roads traffic light controller to simulate the pedestrian/road one? How much effort should be employed to design this new system by reusing the existing one? Is it possible to do that? Well, these are questions we also want to be able to answer through this work.

3.2.4 Different Forms of Reuse

In the examples presented above, different aspects of process behaviour[3] were desired to be reused. When the major/minor roads traffic light controller is reused to implement the traffic light controller in the perfect environment, the need to reuse partial definitions of components is evident. This means that for existing components that are able to respond to a range of actions wider than the component that is actually being developed, we want to reuse only the behaviour over which both components are defined. Notice that if the existing traffic light controller is reused to replace that one in the perfect environment, then part of this process is wasted because it would never be stimulated by the perfect environment. On the other hand, it performs, precisely, all actions demanded by the new component.

When reuse of the major road process is required to simulate the major/minor roads process, we can observe that the existing process is unable to simulate any action from the desired process. But, if an action is defined as a summation of subactions, then we might also be able to reuse processes that simulate a subset of subactions instead of the full set of input/output actions. So, reuse of subsets of actions is also required for efficient reuse of hardware design.

In the case of developing the pedestrian/road traffic light controller by reusing the major/minor roads one, processes are defined over distinct sorts. Apart from the communication with the time counter, all the other incoming and outcoming events are defined over distinct sorts. However, sequences of actions

[3] Since hardware components are represented by EPA processes, the terms "component" and "process" are used with identical meaning. The terms "desired process", "desired specification" or "specification" are used interchangeably for meaning the new component to be developed. The library component, however, will be referred as "existing process", "existing component" or "component".

of the pedestrian/road process can be mapped on to sequences of actions from the major/minor roads process. In this case, reuse is focussed on similarities that both processes have in performing sequences of corresponding actions that cannot directly simulate each other, but an interface is able to identify them.

Notice that reuse can focus on sequences of input/output actions being simulated, on sequences of subactions being simulated, or even on sequences of actions that can be interfaced by a map over their actions. This work is aimed at reusing processes that are similar in at least one of these ways.

3.3 A Proposed Approach for Reuse of Hardware Components

In Chapter 1, we pointed out what kind of information can be reused and some different ways of getting this information reused. We have also seen that the techniques for synthesizing hardware are insufficient to reuse complex systems. This section is devoted to discussing the information we want to reuse from hardware design, which approach is taken for reusing such information, some technical problems that arise with the adopted approach, and what is necessary to solve them.

The greatest interest in improving engineering products, like hardware, is economic. In order to make products economically competitive, two aspects must be considered: productivity and quality. We may therefore make hardware components more competitive by enhancing its productivity and quality. A formal approach is adopted to develop hardware as a way of achieving better quality, while reusability is adopted as a technique to enhance productivity.

As with software reuse, different approaches can be adopted to reuse hardware design, such as creating generators of hardware components from patterns, or defining hardware description languages that capture reusability principles. Our aim, however, is to improve the current state of developing hardware instead of proposing a method that contrasts to the existing ones. What we actually have, in practice, are libraries of components described in a hardware description language. We do not intend to change the way of developing hardware, but provide mechanisms to support reuse of existing components. As an attempt to improve hardware development, we propose reuse of hardware design by identifying similarities between a desired and the existing components, and then automatically generating the context in which the existing component must be embedded to achieve the desired behaviour. This practical view of reuse was earlier noticed by P. Freeman in [Fre87a]:

> One of the primary deterrents to reuse is the ever–present need to change a reuse candidate "just slightly". In reality, those minor changes to make a part work the way we want it in the new application are usually so difficult that we wind up building a new part. Tools that permit us to modify existing components in effective and controlled ways are going to be a high–demand item once reuse becomes widespread.

Reuse of existing components is concerned with the idea of simplifying the task of designing systems by trying to find out if that particular system, or a similar one, has already been developed. Once a similar component is found, the design effort concentrates upon the part of the desired component that has not been developed yet. The "marriage" of reusability to a formal approach for hardware development improves productivity and quality. Nowadays, complexity is one of the most challenging problems in developing new hardware components, even when an appropriate development method is adopted. Considering that high–level definitions of components are supported by formal methods and only subcomponents are developed as reuse of existing components is obtained, reuse of hardware components undertaking a formal approach helps to decrease the complexity of elements that are going to be designed.

Regarding productivity and complexity of components, another important aspect must be pointed out: the desire of reusing all the hardware design refinements down to implementation. The cost of producing hardware components concentrates upon their design and test. So, not only high–level specifications must be reused, but also, all their refinements. This means that hardware components must be represented in a suitable way to have their design reused. Due to the adoption of a formal approach, representation of design shall be restricted to the description of hardware components; the knowledge–based information is not reused. Thus, we shall confine ourselves, in the present work, to reusing descriptive information of hardware components.

Our work is centred on the idea of providing mechanisms that allow existing processes to be formally reused. Given a desired and an existing process, we want to be able to "transform" the existing behaviour into the desired one. Besides that, considering that a formal approach has been adopted in this project, the element that is going to replace the desired component must be proved identical in functionality.

3.3.1 Settling our Reuse Framework

We have already settled the information we want to reuse and the approach to be followed for reuse of hardware design. Now, let us define our framework and some directions we adopt to achieve a solution.

The first element to be chosen for reuse of components is a *model* to represent hardware components. The present work is aimed at giving support to reuse of hardware components defined in ELLA, undergoing a formal approach. The formal approach for reuse requires a semantic definition of processes in order to compare their functionality. The underlying formalization for ELLA is EPA (ELLA Process Algebra) [BGMW94], a process algebra over which a kernel of ELLA is semantically defined [BGMW92]. Even knowing that EPA is a synchronous process algebra that can represent processes other than ELLA components, it is adopted throughout this work as the model for representation and algebraic manipulation of reusable processes[4] (an introduction to EPA has been presented in Chapter 2).

The most important concept underlying reusability is *abstraction*. Abstraction on representation of components is supported by the process algebra used as a model; process algebras provide *processes abstractions*. To make reuse feasible, however, some mechanisms to identify processes under certain abstractions are also necessary. This means that the behaviour of a component must be *reduced* to another by abstracting away irrelevant details. Despite the fact that process algebras provide some support for abstract representation of processes, the key–point for reuse in our work is abstraction on mechanisms for identification of components. Other work for formal reuse, like [GM88], [Gog89], [KRT89], [VK89], [PL89] and [WHS89], achieve reuse by providing abstractions on representation of processes. By contrast, our approach is based on abstractions for identification of processes instead of giving emphasis to the language for representing processes. So, *reducibility* and *equivalence* of processes must be achieved by abstracting irrelevant details when comparing them.

The other important aspect is the approach adopted to actually achieve the desired behaviour by reusing a component. The ideal case of reuse lies in situations where no new processes need to be designed, i.e., the whole component has already been developed. In practice, this ideal case is rarely achieved. On the whole, we are only able to find an existing process that "looks like" the new one, but it does not match the entire functionality of the new process. In these cases, even knowing that the desired process cannot be entirely replaced, the existing component can still be reused. Then, the way to yield behaviour of the desired process is by "transforming" an existing process into the desired one.

Basically, two approaches can be adopted to transform an existing process into a desired one: modifying the design of such a component, or composing the existing process with another one to achieve the desired behaviour. The task involved in the first approach is *redesigning* an existing system into a new one; it is then called *Redesign Approach*. There, design of the existing process is reused to build the desired one under application of transformation rules. In the second approach, the *Interface Approach*, the existing process is reused without any change in design, but a new process (the *interface process*) must be developed. Hence, the existing process has its behaviour "transformed" by composition with the new process.

Both approaches have their advantages and disadvantages. If the aim is to reuse processes at their topmost definition, the interface approach has the advantage of reusing an existing design (from specification down to implementation) without change, while the redesign approach involves the task of redesigning throughout the process refinement. On the other hand, redesign ends up with a process that comprises the exact required behaviour, while in the interface approach the existing process might include some behaviour not required by the desired process. Summarizing, redesign could eventually

[4] Considering that reusable *components* are defined as EPA *processes* these terms will be used interchangeably.

generate a more efficient process and the interface approach might require less effort on designing by following a mixture of decompositional ("divide and conquer") and constructive techniques.

Still comparing these approaches, in the interface approach the new process can be created automatically (under certain constraints) from the behaviour of the desired and existing components. Conversely, the task of redesigning processes may involve automatic as well as manual procedures, and then reuse undertaking redesign might require semi–automatic procedures. Some other aspects of these approaches can be compared, such as domain of problems in which they are more effective, time required to develop processes undertaking both approaches, etc, but we confine ourselves to discussing aspects that are relevant for our application domain: hardware design.

From a practical point of view, the interface approach is attractive for reuse of hardware design because hardware components are reused without redesigning; they can be directly reused at the implementation level. The other aspect which makes this approach attractive is the possibility of having an automatic procedure to find the interface process. Finally, efficiency of hardware components is very much concerned with their implementations. So, the claim that the redesign approach might generate more efficient processes depends on the method applied to synthesize hardware. Since efficiency of hardware components generated by automatic synthesis is still a subject under discussion, comparison of efficiency of processes generated by application of redesign or interface approaches is subjective. Thus, the interface approach appears an attractive way of reusing hardware design.

The use of a formal approach and the desire to reuse components without change, reduce the so–called *reuse operational problems* (Chapter 1) to: searching components and creating the interface process. The automatic generation of interface processes makes *understanding* and *modification* of processes unnecessary. Composition is established, in advance, by the model adopted for design representation. From the problems of searching components and creating the interface process, we confine ourselves to finding a mechanism to automatically create the interface process.

Process algebras support algebraic manipulation of processes. In order to provide algebraic manipulation of reusable processes, the creation of interface processes must be performed by a *combinator*. So, we have reduced the formal reuse of hardware design to a single operational problem: "creating an interface process".

Assumptions

To reuse components effectively, these components must be classified in a suitable way to make them available. Besides that, mechanisms to search components that are the best approximations for the desired one must also be provided. We assume the existence of these mechanisms and consider that whenever a new component is given, if there is any similar component in the library, we are able to find it. Also, we assume that the components to be reused satisfy all the non–behavioural requirements of the desired specification, such as technology of implementation, maximum size, minimal speed, power dissipation and power consumption.

Hardware components can be described in various levels of definition. Hence, reuse of hardware components can be addressed at as many levels of definition as hardware can be described. However, the more abstract the level of definition of hardware, the more concise its description and then, at abstract levels even complex systems become manageable. We assume that any element of hardware must be specified (statement of the problem to be solved) before a solution is given. Another important aspect to be emphasized is that description of a hardware element in an abstraction level other than the topmost specification is considered as a *design step* (or refinement). It is also assumed that all the library components are defined upon their abstract level as well as on some other more detailed levels. This means that we should be able to retrieve all the design refinements of components (including implementation) from the specification level. So, if a process is reused from its specification level, then all of its successive design steps (from specification throughout implementation) are also reused.

As a general attempt to reuse ELLA components, we must also assume that there exists a library of ELLA components together with their corresponding EPA representation. Any new element being developed can pick up one (or some) of those existing elements as a candidate for reuse, and then check how it could be efficiently reused. Also, in order to provide reuse of design, we assume that whatever component is collected, all its refinements are retrieved in a straightforward manner.

Notice that the present work is very concerned with a solution to a technical problem on reuse of hardware design. In spite of being aware of managerial problems in the reuse field, we deliberately leave those problems, as they have been sorted out, to concentrate upon the technical problem.

3.3.2 What Must Be Solved?

What does "creation of an interface process" mean when the desired and existing processes are defined in a process algebra? In this section, we rephrase the problem of "creating an interface process" in terms of EPA. Some related work is presented and a discussion on a solution for such a problem will be conducted later.

As we have stated above, the interface process must be composed with an existing component to provide the behaviour of the desired process. Considering that (strong) bisimulation is adopted as the behavioural equivalence relation in EPA, the resulting composition must indeed be bisimilar to the desired process. Since the set of observable channels is known from the beginning, the *Interface Equation* for this particular reuse problem (synchronous composition and hiding channels) can be defined as:

$$(P \mid X) \upharpoonright_c \mathsf{Chs}(S) \sim S \tag{3.1}$$

In this equation, S is the desired process while P is the existing component, and X is the interface process to be found. Notice that the observable channels of the compound process are restricted to the set of channels of the desired process; any other communication port is internalized in this compound process.

In the equation above, S is decomposed into two other processes: P and X. Reuse undergoing the interface approach is very much concerned with the idea of decomposing the desired process into submodules, so that one of those submodules matches the existing component. Thus, the reuse problem is reduced to a *pseudo–decomposition*[5] of the desired process and the effort involved in performing such decomposition is concentrated upon interfacing the existing process with the desired one.

Looking at reuse as a case of decomposition, the interface process to be created is indeed a context in which the existing process is embedded. This decomposition problem has already been given solution, by Shields [Shi89], for deterministic processes represented in CCS [Mil89] and taking weak bisimulation as equational equality [Shi89]:

$$(P \mid X) \setminus C \approx S \tag{3.2}$$

S, P and X are as in our interface equation and C represents the set of communications to be internalized. Roughly speaking, a solution for this equation is a process that provides the S's actions not simulated by P and internalizes all the P's actions that do not simulate S.

Shields's interface equation has not been proposed to solve the problem of reuse. But, assuming a top–down design of complex systems where the system specification is partitioned into a communication network of submodules, and having developed some of these submodules, this interface equation is aimed at finding the remaining system's behaviour that has not yet been developed. In the case of reuse, the existing process does not necessarily need to be a "natural" subcomponent of the desired process, but an element that carries enough behavioural similarities that justify its reuse.

It is clear that the nature of the problem addressed by Shields and his peers, who solved variants of the same problem [QL91, Par89], differs from ours, but could we still use their techniques to solve the reuse problem?

The interface equations for both problems look similar, but some differences emerge. A solution for these equations strongly depends on the composition operator defined for their respective process algebras. EPA composition is strictly synchronous while CCS composition allows one or both processes communicating with the environment at a time. Moreover, CCS has indivisible actions (one action is communicated) while EPA has structured actions (a set of subactions is communicated at a time).

[5] This term is used to differentiate from the usual decomposition approach.

The EPA structured actions are represented by channels together with their corresponding subactions; channels denote ports of communication and give structure to actions. Then, the EPA hiding operators hide ports of communication in the sense of preventing processes from communication with the external environment through those channels. This means that application of a hiding operator hides subactions instead of the whole action. Besides that, we adopted strong bisimulation as equality instead of weak bisimulation.

Summarizing, beyond the difference on the nature of the problems, we are addressing an interface equation for a strictly synchronous composition with structured actions and hiding subactions. The interface process required as a solution is then largely different.

3.3.3 Looking for a Solution

Having settled the interface equation for synchronous processes, we can now analyse what is necessary to have a possible solution. Since reuse has as one of its main goals to minimize the effort of developing new processes by making use of existing ones, these processes must indicate some similarity in their behaviour. Otherwise, reuse of the existing process could not improve efficiency on the development of the desired one because all the desired behaviour should be provided by the interface process.

So, a solution for the reuse problem must provide the interface process and make the best reuse of similarities between the desired and existing processes. This means that an action from the existing process that is going to simulate a desired one, must be similar in some sense. If any similarity exists between the existing and desired processes, it must be reused.

The techniques presented to find a solution to the interface equation for asynchronous processes are restricted to deterministic processes. Notice that similarities of processes are fully reused in such techniques because the interface process provides only the desired actions that are not present in the existing process. For deterministic processes represented in CCS, identification of actions can be realized "on–the–fly" when the interface process is being created; actions are entirely identified.

However, in the synchronous compositional reuse, actions are represented by summation of some other simpler actions (primitive actions). As a result, when an action of the existing process is being reused to simulate the desired one, it might do not cover the whole behaviour of the desired action because the remaining subactions can be supplied by the interface process. This fact permits reuse of processes that simulate the desired specifications partially. Thus, synchronous composition of processes allows, for example, the major road process to be reused to simulate the major/minor roads traffic light controller. So, how could we interface behaviour of similar actions when this similarity is not denoted by full identification of actions?

The first problem that arises with partial identification of actions is nondeterminism, even when the addressed processes are deterministic. If similarity of actions is restricted to a subset of channels rather than the entire action, it might be possible to identify nondeterministic subactions even for deterministic processes. Suppose an EPA process S can perform actions $c_1(0), c_2(0)/c_3(0)$ and $c_1(0), c_2(1)/c_3(0)$ that are, respectively, simulated by actions $c_1(0), c_4(0)/c_3(0)$ and $c_1(0), c_4(1)/c_3(0)$ from a process P. Despite being deterministic, S can have each of its actions identified with both actions from P and vice–versa, because the set of coincident channels $\{c_1, c_3\}$ are identical for all of these actions. So, if P is interfaced to S "on–the–fly", both S actions are identified with both P actions. But, we have said that $c_1(0), c_4(0)/c_3(0)$ must simulate $c_1(0), c_2(0)/c_3(0)$ while $c_1(0), c_4(1)/c_3(0)$ is going to simulate $c_1(0), c_2(1)/c_3(0)$. Hence, creation of the interface process "on–the–fly" is insufficient to provide an efficient reuse of P to simulate S since identification of subactions is nondeterministic.

The other problem in identification of actions (or subactions) using an "on–the–fly" procedure arises with processes that have some behavioural similarity but are defined over distinct sorts. Even knowing that major/minor roads and pedestrian/road traffic light controllers enjoy some behavioural similarities, their actions cannot be identified with each other because they are defined over distinct sorts. How could we then denote the action from major/minor process that is going to simulate each action from pedestrian/road process?

Essentially, the desire of reusing processes is related to the idea of making reuse of similarities of processes in order to reduce the effort in developing new systems: it is not necessary to "reinvent the wheel" once its design is already known. But, looking at the examples mentioned above, some behavioural similarities of processes are not explicitly denoted by simple identification of actions or subactions. Creation of interface processes "on–the–fly" is a technique sufficient to find interface processes for deterministic processes represented in CCS, but it is not widely applicable to the synchronous problem using structured actions.

A solution for this problem can be found, however, if the similarities between processes are known in advance. If so, the interface process can be created by taking advantage of such similarities.

Nevertheless, how could we establish similarity of processes when their actions are only partially comparable? Furthermore, how could we find behavioural similarity of processes defined over distinct sorts? And finally, what kind of similarity of processes is desirable to be reused in a synchronous approach? The forthcoming chapter is devoted to answering these questions by stating a set of relations on processes that help in reusing existing elements.

Chapter 4

Behavioural Similarity from a Reuse Point of View

4.1 Introduction

To compare the behaviour of processes in order to enable effective "reuse" of an existing component, various notions of similarity can be defined. In this chapter we explore a range of similarity notions which are useful in helping solve the interface equation defined using a synchronous composition operator.

Reuse of processes is characterized by substituting a given component for a desired specification. The notion of substitution has been explored for many equivalence relations defined over deterministic and nondeterministic processes. In automata theory, for example, behaviour of machines are identified by the traces of actions they can perform (trace equivalence). In a process algebra like CCS, behaviour of processes are identified by providing identical actions choice at any time (strong bisimulation). In testing equivalence, processes are equivalent if one passes all tests or experiments that the other passes and vice–versa. Failures equivalence is held when two processes refuse to engage the same set of events after realizing identical traces. In all these relations, processes can precisely substitute for each other.

The notion of substitutivity in the interface equation, however, allows an interface process to run in parallel with the existing component. Thus, bisimulation (the equivalence relation defined for EPA) is not the only relation between the specification and the component that allows for substitution in the interface equation. Certain similarity relations, weaker than bisimulation, must be defined to provide effective reuse of components using the fact that an interface process can be run in parallel.

The following sections are devoted to: defining similarity relations that capture the notion of reusing synchronous processes, classifying such relations, and showing how processes can be constrained to achieve stronger relations. Finally, several algorithms are given for deciding if pairs of processes hold one of the presented relations.

4.2 Similarities between Synchronous Processes

The behaviour of processes is characterized by actions they can perform at their initial and subsequent states. On the ground level, identification of process behaviour is concerned with identifying actions. In equivalence relations that ensure substitutivity, identification of actions is realized by their "equality". Equality of synchronous actions, like EPA actions, is determined by identifying all subactions. To realize that two synchronous actions perform identical communication with their environment, all their subactions must be identified with each other. Furthermore, identical primitive actions are determined by equality of the actual channel name and the corresponding action passing through it. So, synchronous actions are identified by equality of all their primitive actions. To some extent, the channel names of both processes could be identified by a one–to–one map. However, in order to simplify comparison of processes, identical channel names are used to identify potential primitive actions from both processes.

Although substitutivity of processes requires identical actions (identical channel names and sorts), we want to be able to reuse processes under a synchronous interface approach even when specifications and components are defined over distinct sorts. This means that for realizing potential reusable processes, mechanisms other than a conventional "equality" of actions must be provided. The fact that a synchronous action is a summation of various subactions running in parallel, synchronous actions can be "identical" in a subset of those actions. So, a relation considering partial identification of actions is required for effective reuse of synchronous processes. Besides that, abstraction on the sorts of actions over which the processes are defined must also be provided for comparing processes.

In a synchronous calculus like EPA, compound actions (actions performed by compound processes) are realized by all component processes running synchronously. Therefore, similarity of processes in a synchronous interface approach is only useful if each action from the specification can be identified with an action from the given component. Otherwise, synchronous composition of the component with the interface process would never yield the behaviour of the specification. Hence, useful similarities for reusing components undertaking a synchronous interface approach must consider partial identification of actions defined over distinct sorts, such that each specification action is identified with a component action.

Reuse of processes is concerned with the idea of reducing the effort of developing new components. So, similarity of processes must be based on relations that allow a minimal interface process to be created. This means that the behaviour of processes must be identified as much as possible in order to require a minimal interface. Ideally, reuse of processes ought to require no new processes to be constructed. This special case is achieved, in particular, by bisimilar processes. But, other similarity relations can be created to allow for substitutivity when an interface process is composed with the given component. Considering that only synchronous processes are described in ELLA, the EPA processes representing ELLA components are free of silent actions. So, the relations presented here are concerned with finitely branching and concrete processes.

In this section we present a range of similarity relations useful for reusing synchronous processes. Looking forward to reusing existing components, we first consider a relation that permit, under certain constraints, substitution without constructing an interface process. Second, other similarities are defined for relating processes that require a single state process as interface. And finally, several other similarity relations are defined in order to require multi–state interface processes.

4.2.1 Simulation

In principle, processes can replace each other if they are bisimilar. Since reuse only requires the given component substitute for the specification one, a simulation relation is also useful for reuse. For such a simulation relation, both processes must behave identically for all input actions defined for the specification, but some actions performed by the given component might not be defined as communication of the specification. This means that the specification cannot respond to some of the events that the component can.

A simulation useful for reuse is based on the idea of definedness of processes[1]. Roughly speaking, a process is more defined than another if it can respond to all stimulus the other process can, or even more. But, whenever both processes respond to identical stimulus, they must behave bisimilarly. The idea of definedness is concerned with partial definition of processes: a process is less defined than another if it is a partial definition of the other.

Let us illustrate definedness of processes by the examples of Stop, R_1 and R_2 shown in Figure 4.1.

Suppose Stop, R_1 and R_2 are embedded in an environment that communicates using actions in a sort $\mathcal{A}_1$. Stop refuses all stimuli from the environment. Therefore, they never communicate with each other. R_1, unlike Stop, is able to communicate with the environment undertaking action a/x. R_2, however, can communicate with the environment using actions a/x or b/y.

From a definedness point of view, R_1 is more defined than Stop: it can communicate with the external environment while Stop is unable to do so. Defining processes definedness as the ability of communicating

[1] This term has been used by Roscoe, in [Ros88], to order divergent processes. But it is redefined here to order non–divergent processes.

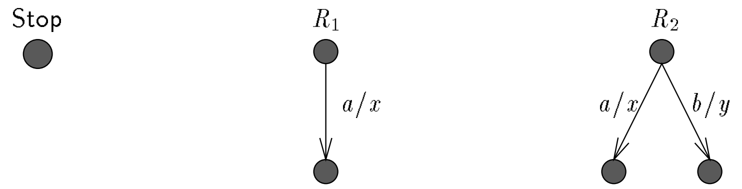

Figure 4.1: Stop Process

with the external environment, R_1 can do everything Stop does plus some communication that Stop is unable to perform. For the same reason, R_2 is more defined than R_1.

This concept of definedness leads to an ordering of processes: Stop is less defined than R_1, and consequently lower ordered than it. Similarly, any other process defined over sort $\mathcal{A}_1$ is higher ordered than Stop; R_2 is higher ordered than R_1 and consequently higher ordered than Stop.

The examples presented above have shown an ordering of deterministic processes. But, what about nondeterministic processes? Using bisimulation as a principle for comparing processes, simulation can be considered as a "one–way bisimulation" where all moves performed by the lower ordered process are identically performed by the other process. This means that deterministic as well as nondeterministic processes can be ordered from a definedness point of view. But, such an ordering does not relate processes that respond differently to identical events.

Let us set the concept of definedness for actions. Since EPA processes are stimulated by input events when the clock is pulsed, definedness of actions is defined over input actions. Processes are equivalently defined for an input action a if they make identical moves (deterministic or nondeterministic) for such an action. Processes R_1 and R_2, for example, are identically defined for action a; they behave bisimilarly for such an action. Thus, processes can only be ordered if all the simulated actions (actions from the simulated process) are identically defined for both processes. Process R_3 (Figure 4.2), for example, cannot be ordered with R_1 or R_2 because it is nondeterministic for action a while those other processes are deterministic.

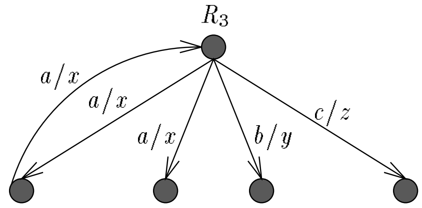

Figure 4.2: R_3 Process

This definedness ordering is very much concerned with reuse. A process can only be reused if it responds identically to the simulated process for all stimulus coming from the environment. The idea of ordering processes based on definedness principles was earlier addressed by Roscoe [Ros88, Bar93] for divergent CSP processes (there, only divergent processes are comparable). He ordered divergent processes that are extensions of others: a process is higher ordered than another if it can still communicate with the environment, without diverging, by the time the simulated process diverges. Despite addressing definedness, our ordering cannot be compared to the Roscoe's one because no divergent processes have been considered in the current work. In other process orderings, such as the Hoare's [Hoa85] and Hennessy's [Hen88] orderings, deterministic processes can be ordered with nondeterministic ones, but "spare actions" are not allowed. So, these orderings cannot be classified with our ordering. The simulation semantics introduced by Milner [Mil71] and Park [Par81] leads to a simulation preorder where "spare actions" are permitted. Our ordering is a restriction of the Milner's one in the sense that "spare actions" differ from the set of simulated actions (a range of preorder relations is surveyed by van Glabbeek in

[vG93]).

Let us now formalize our relation for ordering processes under definedness point of view: Simulation.

Definition 4.1 *Simulation $\mathcal{S}$ is a binary relation on processes such that if $(P, Q) \in \mathcal{S}$ (abbreviated by $P \sqsubseteq Q$), then for every "proper" action*[2] *$i_1/o_1 \in Act$ such that $i_1 \in \mathsf{Sort}_{1i}(P)$*[3]*:*

1. *whenever $P \xrightarrow{i_1/o_1} P'$ then $Q \xrightarrow{i_1/o_1} Q'$ for some Q' such that $P' \sqsubseteq Q'$,*
2. *whenever $Q \xrightarrow{i_1/o_1} Q'$ then $P \xrightarrow{i_1/o_1} P'$ for some P' such that $P' \sqsubseteq Q'$.*

A process can simulate another only if for any input action defined for the simulated process, the simulation process can only communicate with the environment identically. Notice that only simulated input actions are taken into account ($i_1 \in \mathsf{Sort}_{1i}(P)$). Input actions other than those accepted by P are ignored because they are irrelevant for simulating P.

Suppose process S (Figure 4.3) is a desired specification[4] while processes P_{nosim} and P_{sim} (Figure 4.4) are possible candidates for simulating S.

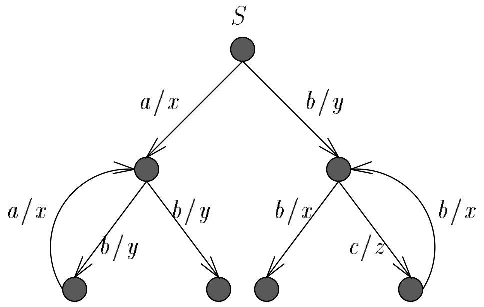

Figure 4.3: Process S

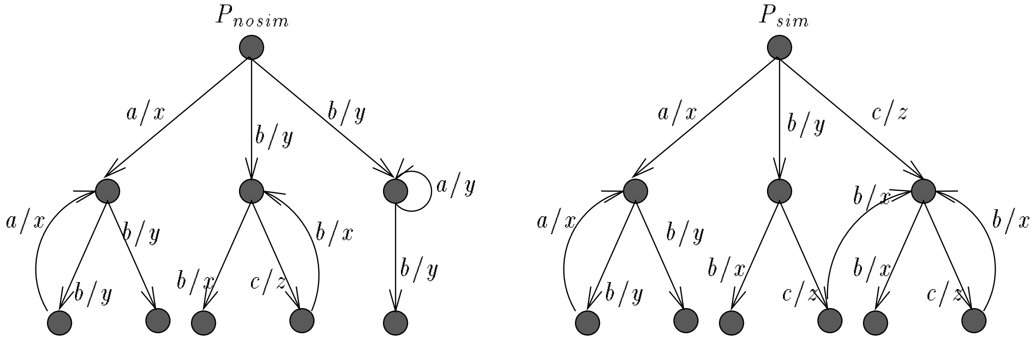

Figure 4.4: Processes Dissimilar and Similar to S

Apparently, P_{nosim} is able to behave like S. But, P_{nosim} has a double choice for the input action b while S has a deterministic choice for the same action. By the time the input action b is demanded, if P_{nosim} chooses the alternative action b on the right–hand side, this process will deadlock for any other action performed by S after making this communication. Because of that, P_{nosim} cannot substitute for S.

[2] Actions that differ from $\boxtimes$.

[3] Act represents all existing actions and $\mathsf{Sort}_{1i}(P)$ comprises all initial input actions of P.

[4] In this Chapter, derivation trees of processes (instead of process definitions) are shown for comparing processes behaviour.

P_{sim}, however, is a simulation for S. Despite P_{sim} accepting the input actions a, b and c while S is ready to communicate using actions a/x and b/y, P_{sim} can simulate all the S's actions. If b is demanded as input action, for example, P_{sim} behaves equivalently to S for all the alternative actions after performing this initial action: they either reach Stop when performing b/x, or undertake c/z and then b/x, and so on. So, P_{sim} behaves bisimilarly to S whenever S's events are demanded.

Simulation is a preorder upon all processes. The set of all processes is a complete partial order under simulation where the least upper bound process is Stop. Additionally, processes that have all their actions restricted to a set of actions (denoted as $\lceil_a \mathcal{A}_n$) are also lower ordered than the unrestricted process.

Lemma 4.1 *Simulation as preorder of processes:*

1. $\mathsf{Stop} \sqsubseteq P$
2. $P \lceil_a \mathcal{A}_n \sqsubseteq P$

Processes restricted to a set of actions are obviously lower ordered than the original process with no restrictions. If some actions are prevented from a process (when restriction to actions is applied), then the restricted process is obviously less defined than the unrestricted one. If process R_2, for example, is restricted to the set of actions $\{a/x\}$, then it becomes bisimilar to R_1 which is clearly simulated by R_2.

4.2.2 Simulation under a Total Functional Interface

Weaker relations can be established to classify processes as possible candidates for reuse. Many processes can fit the interface process, but, we are, initially, interested in the simplest process: a single state process. Let us now establish a relation on processes such that the behaviour of the specification can be reached by composition of the given component with a single state process. In the sense of reusability, if processes are related in such a way, we can anticipate the complexity[5] of the interface process to be created: a single state process.

In order to have a single state process as interface, a function must be found ranging over the sorts of input and output actions of both processes. The idea of mapping on sorts suggests that besides making comparison on processes with identical sorts, we can also abstract from sorts and still check if a process can be simulated by the other.

This functional transformation of actions can be easily checked when, for example, a comparison on processes $NorDel_p(c_1, c_2/c_3)$ and $Del_p(c_4/c_3)$ (Figure 4.5) is made.

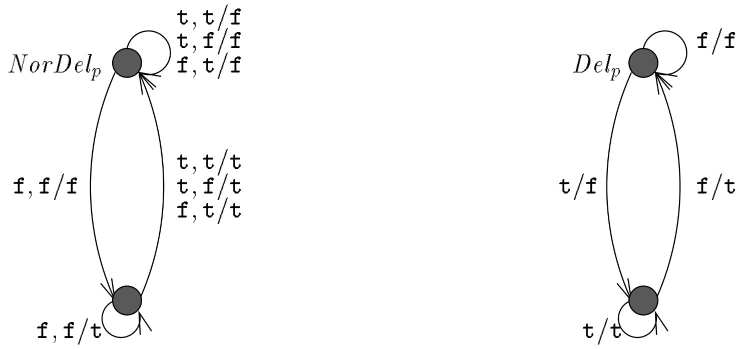

Figure 4.5: Transition Graphs for Processes $NorDel_p(c_1, c_2/c_3)$ and $Del_p(c_4/c_3)$

In comparing the behaviour of $NorDel_p(c_1, c_2/c_3)$ against $Del_p(c_4/c_3)$, we can realize they have identical output behaviour, but their input behaviours are incomparable. However, a function can be

[5] In a very simplistic view of complexity of processes, we adopt the "number of states" as a measure for complexity. In terms of hardware, an one state process corresponds to a circuit with no memory.

$f_{ND\text{-}Del}$
$c_1(\mathtt{t}),\, c_2(\mathtt{t}) \mapsto c_4(\mathtt{f})$
$c_1(\mathtt{t}),\, c_2(\mathtt{f}) \mapsto c_4(\mathtt{f})$
$c_1(\mathtt{f}),\, c_2(\mathtt{t}) \mapsto c_4(\mathtt{f})$
$c_1(\mathtt{f}),\, c_2(\mathtt{f}) \mapsto c_4(\mathtt{t})$

Table 4.1: Interfacing $NorDel_p(c_1, c_2/c_3)$ and $Del_p(c_4/c_3)$ IO–Actions

found to map on $NorDel_p(c_1, c_2/c_3)$ input actions to $Del_p(c_4/c_3)$ input actions (this function is shown in Table 4.1).

If such a function is applied to input actions of $NorDel_p(c_1, c_2/c_3)$, actions of both processes become comparable. Action $c_1(\mathtt{f}), c_2(\mathtt{f})/c_3(\mathtt{f})$ from $NorDel_p(c_1, c_2/c_3)$, for example, can be identified with action $c_4(\mathtt{t})/c_3(\mathtt{f})$ from $Del_p(c_4/c_3)$ under application of $f_{ND\text{-}Del}$:

$$f_{ND\text{-}Del}(c_1(\mathtt{f}), c_2(\mathtt{f}))/c_3(\mathtt{f}) = c_4(\mathtt{t})/c_3(\mathtt{f}).$$

Function $f_{ND\text{-}Del}$ represents the nor–gate: a single state process. This means that if a nor–gate process is composed with $Del_p(c_4/c_3)$, then behaviour of $NorDel_p(c_1, c_2/c_3)$ is yielded. In fact, $NorDel_p(c_1, c_2/c_3)$ has been defined in Chapter 2 (Section 2.2.3) as a composition of the nor–gate with a delay process, and the relation "simulation under a total functional interface" is aimed at identifying such a similarity between processes.

With the example above, we have shown that input actions of certain processes can be identified (through a function) if their output behaviours are identical. But, this is not the weakest relation that can be found to relate processes via a function, output actions can also be mapped. For instance, if processes S and P_{tfsim} (Figure 4.6) are compared, it can be noted that their input and output actions differ from each other.

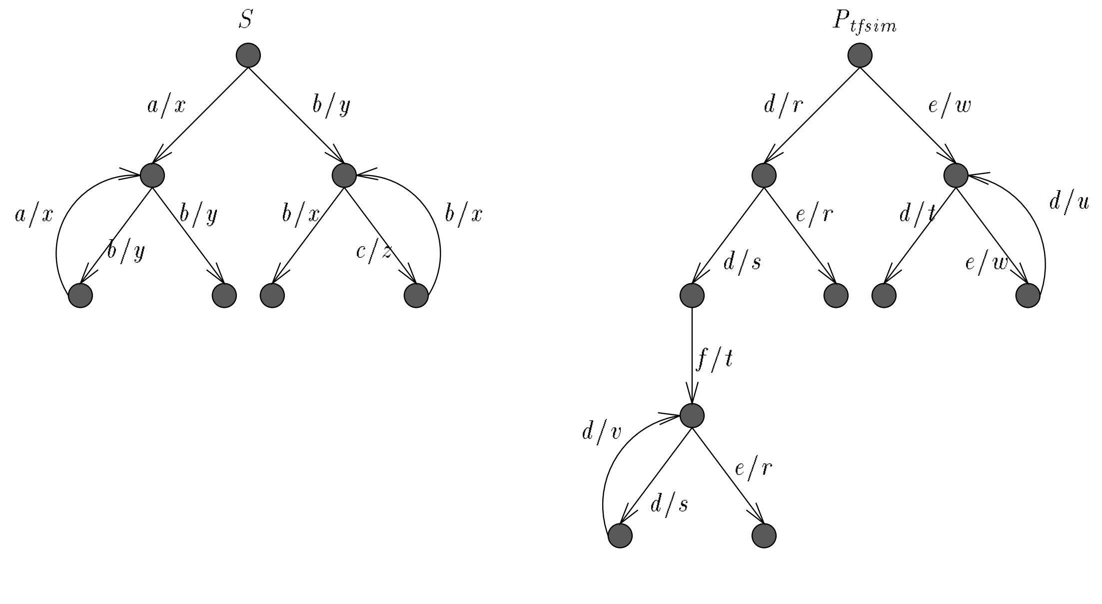

Figure 4.6: Process P_{tfsim}

What information is available in the domain and codomain of functions which interface input and output sorts of S and P_{tfsim}? The S input events are basically the information available for running such a compound process (at each clock pulse, the process responds to the environment depending on the

$f_{S\text{-}P_{tfsim}}$
$(a, d/r) \mapsto x$
$(b, e/w) \mapsto y$
$(b, d/s) \mapsto y$
$(b, e/r) \mapsto y$
$(b, d/t) \mapsto x$
$(c, e/w) \mapsto z$
$(a, f/t) \mapsto x$
$(b, d/u) \mapsto x$
$(a, d/v) \mapsto x$

Table 4.2: Interfacing S and P_{tfsim} IO–Actions

input communicated from the environment). Furthermore, as we are dealing with synchronous processes running in parallel, P_{tfsim} must make some move for each S input event; each S input event must stimulate P_{tfsim} in order to simulate the corresponding S output action. So, a function to be found can map input actions of S together with P_{tfsim}'s actions to the corresponding S output actions. Table 4.2 shows a particular function which interfaces P_{tfsim} with S.

The idea of having a function ranging over such a domain and codomain is very much concerned with the fact that the component and the interface processes are run in parallel. Given an input stimulus to these component processes, they can interchange some information but must communicate the right S output to the environment. Figure 4.7 shows how P_{tfsim} must be connected to an interface process (P_f) in order to behave like S.

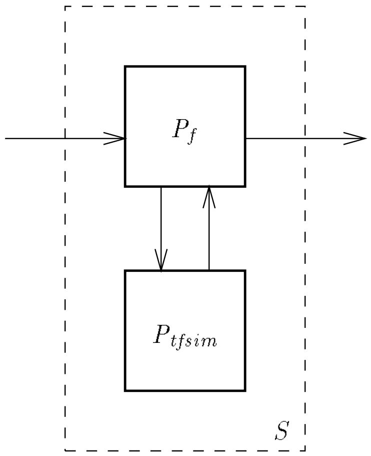

Figure 4.7: Connections of P_{tfsim} with an Interface Process

The idea of relating processes under a total functional interface is more formally defined as follows:

Definition 4.2 *Simulation under a Total Functional Interface $\mathcal{S}_{tf}$, for $f\colon \mathsf{Sort}_i(P) \times \mathsf{Sort}(Q) \longrightarrow \mathsf{Sort}_o(P)$, is a binary relation on processes such that if $(P, Q) \in \mathcal{S}_{tf}$ (abbreviated by $P \overset{f}{\approx} Q$), then for every "proper" action $i_p/o_p, i_q/o_q \in Act$:*

1. *whenever* $P \xrightarrow{i_p/o_p} P'$ *then* $Q \xrightarrow{i'_q/o'_q} Q'$ *for some* i'_q/o'_q *and* Q' *such that* $f(i_p, i'_q/o'_q) = o_p$ *and* $P' \overset{f}{\approx} Q'$,
2. *whenever* $Q \xrightarrow{i_q/o_q} Q'$ *then* $P \xrightarrow{i'_p/o'_p} P'$ *for some* i'_p/o'_p *and* P' *such that* $f(i'_p, i_q/o_q) = o'_p$ *and* $P' \overset{f}{\approx} Q'$.

f ranges over all actions of the specification and component processes. Since f is a function, it can be implemented by a one state process (P_f): for every proper action in the P_f's sort, P_f evolves to itself. More formally,

Definition 4.3 *Given a function* f *with a signature* $\mathsf{Sort}_i(P) \times \mathsf{Sort}(Q) \longrightarrow \mathsf{Sort}_o(P)$, P_f *is a process that:*

1. *for all* $(i_p, i_q/o_q) \mapsto o_p \in f$, $P_f \xrightarrow{(i_p \oplus o_q) \ominus o_p/(i_q \oplus o_p) \ominus i_p} P_f$.

Notice that P_f interfaces all input and output actions between P and Q. The specification and component processes can carry some coincident input and/or output channels. In order to avoid overlapping input and output channels in P_f (by definition, EPA processes are not permitted to overlap input and output ports of communication), the specification output channels are eliminated from P_f's input while the component input channels are eliminated from its outputs. In spite of such subtractions, P_f comprises all channels ranging over f's domain and codomain. As a result, each component action can be exclusively combined with actions from P_f to simulate the specification actions; P_f is the "interface process" that transforms component actions into specification ones.

The fact that each specification state must be simulated by a component one does not restrict the component to have as many states as the specification. Process P_{tfsim}, for example, is similar to S under a total functional interface although having distinct number of states.

Moreover, each S action can be related to more than one action from P_{tfsim}. Action a/x from S, for example, is related to f/t and d/v from P_{tfsim} by maps $(a, f/t) \mapsto x$ and $(a, d/v) \mapsto x$ in $f_{S\text{-}P_{tfsim}}$. Conversely, a P_{tfsim} action can also be related to several actions from S. For instance, e/w from P_{tfsim} is related to b/y and c/z from S.

Another important feature is the possibility of relating deterministic with nondeterministic processes. Process S, for example, is nondeterministic for action b/y (after performing a/x), but these nondeterministic actions are simulated by deterministic actions from P_{tfsim}: d/s and e/r. In spite of that, these deterministic actions can simulate such a nondeterministic behaviour of S when running in parallel with a process that implements function $f_{S\text{-}P_{tfsim}}$.

Suppose $f_{S\text{-}P_{tfsim}}$ is implemented by the single state process $P_{S\text{-}P_{tfsim}}$. When S is ready to perform action b/y, P_{tfsim} is ready to communicate through actions d/s or e/r. But, at the same time, $P_{S\text{-}P_{tfsim}}$ is ready to perform actions $(b \oplus s) \ominus y/(d \oplus y) \ominus b$ (it corresponds to map $(b, d/s) \mapsto y \in f_{S\text{-}P_{tfsim}}$) or $(b \oplus s) \ominus y/(e \oplus y) \ominus b$ (it corresponds to map $(b, e/r) \mapsto y \in f_{S\text{-}P_{tfsim}}$). If b is given as input action, process $(P_{tfsim} \mid P_{S\text{-}P_{tfsim}})$ is nondeterministic in choosing the compound actions:

$$(d/s \star ((b \oplus s) \ominus y/(d \oplus y) \ominus b)) \text{ or}$$
$$(e/r \star ((b \oplus s) \ominus y/(e \oplus y) \ominus b)).$$

This nondeterministic behaviour is bisimilar to S's behaviour when excited by b. Likewise, a deterministic processes can be simulated by nondeterministic ones.

Regarding hardware design, the great interest in relating processes under a total functional interface is concerned with the possibility of creating state free elements whose response depends on the actual input event (in contrast to sequential processes which depend on the state assignment). Combinational circuitries are represented in process algebras by processes which evolve to themselves when performing any action. So, single state processes are implemented by simple combinational circuitries.

4.2.3 Simulation under a Functional Interface

Simulation under a total functional interface allowed us to find a *total function* to interface S with P_{tfsim}. This means that all moves performed by the component process are used to simulate S.

However, S could also be simulated by a process that includes "spare actions"; some component actions which do not simulate any S's action. Simulation under a functional interface is a relation on processes weaker than the total functional interface version, where the presence of component actions not related to S actions is permitted.

Definition 4.4 *Simulation under a Functional Interface $\mathcal{S}_f$, for $f\colon \mathsf{Sort}_i(P) \times \mathsf{Sort}(Q) \nrightarrow \mathsf{Sort}_o(P)$, is a binary relation on processes such that if $(P, Q) \in \mathcal{S}_f$ (abbreviated by $P \overset{f}{\sqsubseteq} Q$), then for every "proper" action $i_p/o_p, i_q/o_q \in Act$ and $i'_p \in \mathsf{Sort}_{1i}(P)$:*

1. *whenever $P \xrightarrow{i_p/o_p} P'$ then $Q \xrightarrow{i'_q/o'_q} Q'$ for some i'_q/o'_q and Q' such that $f(i_p, i'_q/o'_q) = o_p$ and $P' \overset{f}{\sqsubseteq} Q'$.*
2. *whenever $Q \xrightarrow{i_q/o_q} Q'$ and $(i'_p, i_q/o_q) \in \mathsf{dom}\, f$ then $P \xrightarrow{i'_p/o'_p} P'$ for some o'_p and P' such that $f(i'_p, i_q/o_q) = o'_p$ and $P' \overset{f}{\sqsubseteq} Q'$.*

To illustrate such a relation, assume S as the specification process while P_{fsim} (Figure 4.8) is the component.

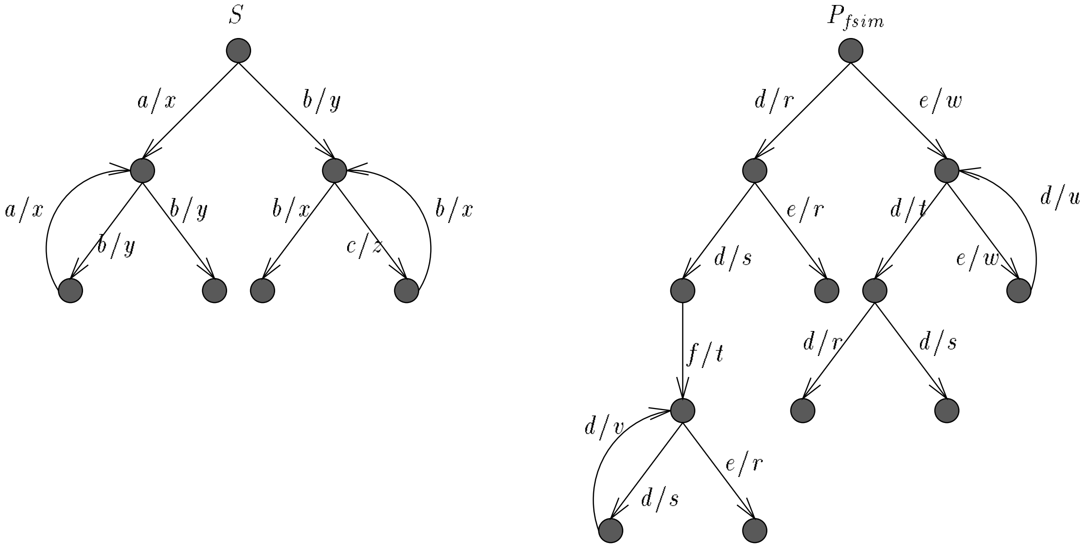

Figure 4.8: Processes S and P_{fsim}

For this particular case, the interface is provided by function $f_{S\text{-}P_{tfsim}}$ already defined for interfacing S with P_{tfsim} (Table 4.2), because P_{fsim} is a simulation of P_{tfsim}. Actions d/r and d/s are wasted since no corresponding event could excite S by the time P_{fsim} has performed d/t; at that point, S has already reached the Stop process.

4.2.4 Channel Restricted Bisimulation

Bisimulation, simulation and variants of simulation under a functional interface are concerned with the existence of an empty or a single state interface process. Processes not related through the above could still indicate some similarity in behaviour such that a multi–state interface process is required.

Abstraction on processes' sorts has been employed to compare processes similar under a total functional interface (a functional interface). However, all channels of both processes were considered in those relations. Obviously, processes can also be compared if only a subset of channels are taken into account.

In comparing action $(c_1(0), c_2(1))/(c_3(0), c_4(0))$ with $(c_5(1), c_2(1))/(c_3(0), c_4(0))$, for example, it can be noticed they coincide in channels c_2, c_3 and c_4. These actions are then identical when restricted to the set of channels $\{c_2, c_3, c_4\}$. Thus, we may also find processes that are identical when restricted to a set of channels.

An RS–flipflop specified by composition of nor–gate–delay processes, for example, can be compared against a nor–gate–delay process if restricted to a subset of channels:

$$\textsf{process } NorDel_p(c_1, c_4\colon bool/c_3\colon bool) \triangleq \Big(Nor_p(c_1, c_4/c_5) \mid Del_p(c_5/c_3) \Big) \backslash \{c_5\}$$
$$\textsf{process } RSff(c_1, c_2\colon bool/c_3, c_4\colon bool) \triangleq NorDel_p(c_1, c_4/c_3) \mid NorDel_p(c_3, c_2/c_4)$$

The transition graphs representing action values of these processes are shown in Figure 4.9. These processes are not similar under a functional interface because no function can be found to map their sorts, but they have identical behaviour when restricted to the set of channels $\{c_1, c_3, c_4\}$.

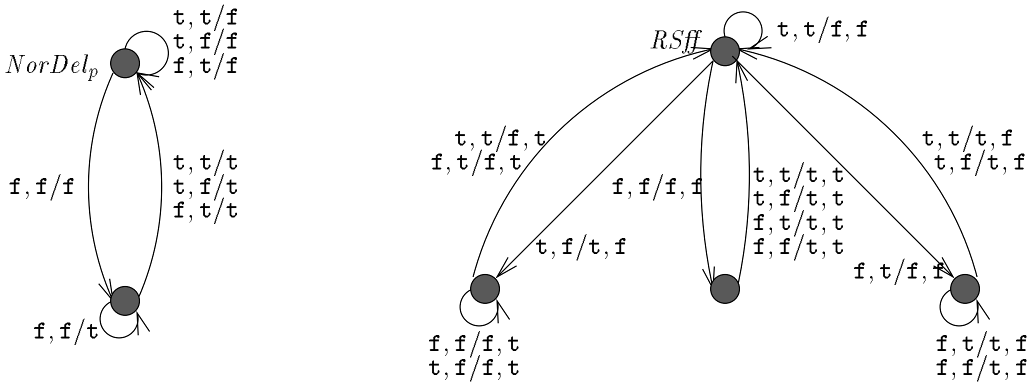

Figure 4.9: Transition Graphs for $NorDel_p(c_1, c_4/c_3)$ and $RSff(c_1, c_2/c_3, c_4)$

A relation for comparing processes restricted to the set of their coincident channel names is defined as follows:

Definition 4.5 *Channel Restricted Bisimulation $\mathcal{B}_c$ is a symmetric binary relation on processes such that if $(P, Q) \in \mathcal{B}_c$ (abbreviated by $P \overset{c}{\sim} Q$), then there exists a non–empty set of channels $C = (\mathsf{Chs}(P) \cap \mathsf{Chs}(Q))$ such that $P \restriction_c C \sim Q \restriction_c C$.*

This relation is indexed by a set of channels of both processes that have identical names; processes are channel restricted bisimilar if they are bisimilar when restricted to the set of channels C. This means that a family of channel restricted bisimulations turns out with the general channel restricted bisimulation, each one indexed by a set of channels.

Channel restricted bisimulation is then an equivalence relation where each C restricted bisimulation forms an equivalence class of processes.

Notice that to be channel restricted bisimilar, the coincident channels of both processes are either input or output channels. If an input channel name of one process coincides with an output channel name of the other process, actions of both processes could never be identified with each other.

Lemma 4.2 *Suppose $S \overset{c}{\sim} P$, then P is channel restricted bisimilar to S only if $\mathsf{Chs}_i(S) \cap \mathsf{Chs}_o(P) = \emptyset$ and $\mathsf{Chs}_o(S) \cap \mathsf{Chs}_i(P) = \emptyset$.*

4.2.5 Channel Restricted Simulation

Channel restricted simulation relates processes up to a restricted set of channels, but it is even weaker than the channel restricted bisimulation in the sense that some "spare actions" could be found in the simulation process.

Definition 4.6 *Channel Restricted Simulation $\mathcal{S}_c$ is a binary relation on processes such that if $(P, Q) \in \mathcal{S}_c$ (abbreviated by $P \overset{c}{\sqsubseteq} Q$), then there exists a non–empty set of channels $C = (\mathsf{Chs}(P) \cap \mathsf{Chs}(Q))$ such that $P \restriction_c C \sqsubseteq Q \restriction_c C$.*

To illustrate such a relation, compare processes $S(c_1, c_2, c_3/c_4, c_5, c_6)$ and $P_{csim}(c_1, c_2/c_4, c_5)$ shown in Figure 4.10 (the S's action values are represented by the sequence of channels $(c_1\, c_2\, c_3/c_4\, c_5\, c_6)$, and $(c_1\, c_2/c_4\, c_5)$ is the sequence of channels of P_{csim} actions).

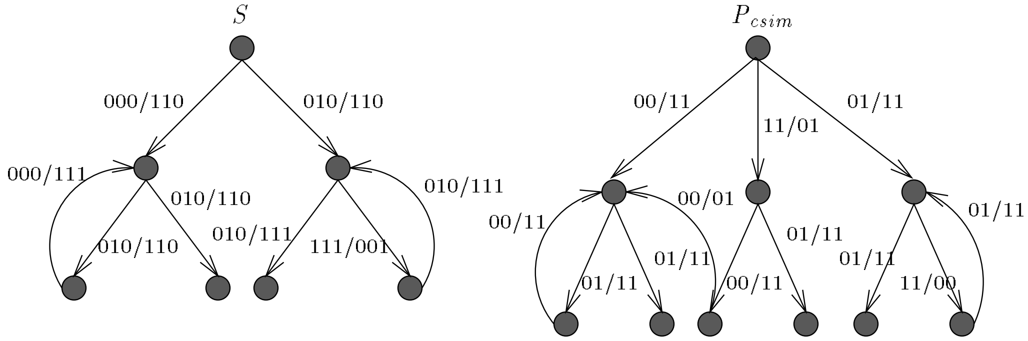

Figure 4.10: Processes $S(c_1\, c_2\, c_3/c_4\, c_5\, c_6)$ and $P_{csim}(c_1\, c_2/c_4\, c_5)$

Processes S and P_{csim} are compared under restriction to the set of channels $C = \{c_1, c_2, c_4, c_5\}$; S and P_{csim} are C restricted similar. Actions 000/110 and 010/110 from S are respectively C restrictedly simulated by 00/11 and 01/11 from P_{csim}. The P_{csim}'s initial action 11/01 is not able to simulate any action of S, it is then a "spare action" that cannot be used to simulate S.

Simulation of channel restricted processes indexed by C is also a preorder of processes defined over supersets of C, where Stop (defined over the universal set of channels) is the *least upper bound* process of this complete partially ordered set.

Lemma 4.3 *Channel restricted simulation as preorder of processes:*

1. $P \restriction_c C \overset{c}{\sqsubseteq} P \quad$ *if* $C \cap \mathsf{Chs}(P) \neq \emptyset$

Every set of channel names indexes a preorder of processes defined over its supersets. Therefore, channel restricted simulation determines a family of preorder classes of processes such that each class is indexed by a non–empty set of channels.

4.2.6 Channel Restricted Simulation under a Total Functional Interface

It is clear that we can now combine the channels restricted similarity relations with the functional interface relations.

Consider, for example, processes $P_{tfcsim}(c_7\, c_8\, c_9/c_{10}\, c_{11}\, c_{12})$ and $S(c_1\, c_2\, c_3/c_4\, c_5\, c_6)$ (Figure 4.11). These are neither related via our notion of channel restricted relations nor even under functional interface relations.

Since S and P_{tfcsim} are defined over distinct sets of channel names, they could not be compared as bisimilar or similar, nor even as channel restricted bisimilar/similar. Also, no functions can be found to map input/output actions of these processes. Alternatively, we may try to find out a function to map

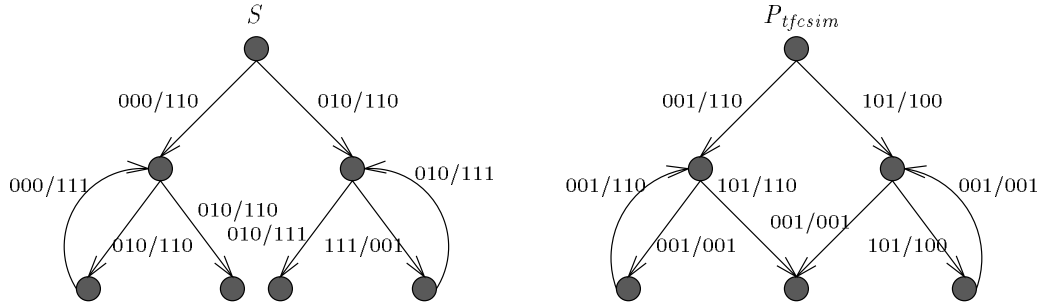

Figure 4.11: Processes $S(c_1\,c_2\,c_3/c_4\,c_5\,c_6)$ and $P_{tfcsim}(c_7\,c_8\,c_9/c_{10}\,c_{11}\,c_{12})$

$f'_{S\text{-}P_{tfcsim}}$	$f''_{S\text{-}P_{tfcsim}}$
$(000, 01/110) \mapsto 11$	$(000, 01/100) \mapsto 11$
$(010, 01/100) \mapsto 11$	$(010, 01/110) \mapsto 11$
$(010, 01/001) \mapsto 11$	$(010, 01/100) \mapsto 11$
$(010, 01/110) \mapsto 11$	$(010, 01/001) \mapsto 11$
$(111, 01/100) \mapsto 00$	$(111, 01/001) \mapsto 00$
	$(000, 01/001) \mapsto 11$

Table 4.3: Interfacing S and P_{tfcsim}

on channel restricted actions of P_{tfcsim} and S (making a combination of abstraction on sets of channels and sorts).

For processes S and P_{tfcsim}, in particular, if only the sets of channels $\{c_1, c_2, c_3, c_4, c_5\}$ from S and $\{c_8, c_9, c_{10}, c_{11}, c_{12}\}$ from P_{tfcsim} are used (the restricted S and P_{tfcsim} ($S'(c_1\,c_2\,c_3/c_4\,c_5)$ and $P'_{tfcsim}(c_8\,c_9/c_{10}\,c_{11}\,c_{12})$) are shown in Figure 4.12), a function that interfaces S with P_{tfcsim} channel restricted actions can be found.

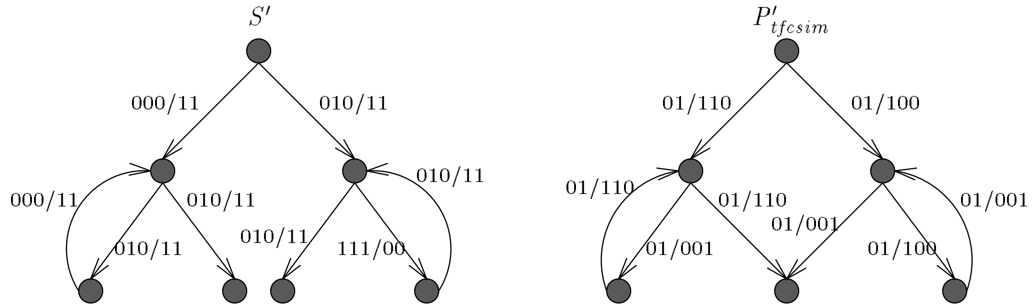

Figure 4.12: Channel restricted S and P_{tfcsim}

Taking advantage of distinct actions becoming identical when restricted to channels (the input actions 001 and 101 from P_{tfcsim}, for example, become identical when restricted to channels c_8 and c_9), the alternative functions shown in Table 4.3 are found[6].

The gain in comparing processes in such a way is that of finding the maximal similarity between P_{tfcsim} and S. Initially, S and P_{tfcsim} appear to have no similarity at all. By making abstraction on

[6] The channel names of actions are dropped in these functions to assume maps like $((c_1\,c_2\,c_3), (c_8\,c_9/c_{10}\,c_{11}\,c_{12})) \mapsto (c_4\,c_5)$.

both processes' sorts and sets of channels, we are able to find a sufficient similarity that enables a single state interface process to shift to a channel restricted bisimilarity.

Let us now formalize the idea of channel restricted simulation under a total functional interface:

Definition 4.7 *Channel Restricted Simulation under a Total Functional Interface* $\mathcal{S}_{ctf}$*, for a non–empty set of channels* $C \subseteq (\mathsf{Chs}(P) \cup \mathsf{Chs}(Q))$ *and* $f\colon \mathsf{Sort}_i(P \upharpoonright_c C) \times \mathsf{Sort}(Q \upharpoonright_c C) \rightarrow \mathsf{Sort}_o(P \upharpoonright_c C)$*, is a binary relation on processes such that if* $(P, Q) \in \mathcal{S}_{ctf}$ *(abbreviated by* $P \overset{cf}{\approx} Q$*), then for every "proper" action* $i_p/o_p, i_q/o_q \in Act$*:*

1. *whenever* $P \xrightarrow{i_p/o_p} P'$ *then* $Q \xrightarrow{i'_q/o'_q} Q'$ *for some* i'_q/o'_q *and* Q' *such that* $f(\textit{res-io-ch}(i_p, C), \textit{res-ch}(i'_q/o'_q, C)) = \textit{res-io-ch}(o_p, C)$ *and* $P' \overset{cf}{\approx} Q'$*,*
2. *whenever* $Q \xrightarrow{i_q/o_q} Q'$ *then* $P \xrightarrow{i'_p/o'_p} P'$ *for some* i'_p/o'_p *and* P' *such that* $f(\textit{res-io-ch}(i'_p, C), \textit{res-ch}(i_q/o_q, C)) = \textit{res-io-ch}(o'_p, C)$ *and* $P' \overset{cf}{\approx} Q'$*.*

This relation is indexed by a set of channels C. The interface function maps on channel restricted actions of both processes instead of the full actions as defined in simulation under a total functional interface. Identification of actions from both processes is obtained via a function f. Since f is total, there exists a corresponding action in the simulated process for each action in the simulation one and vice–versa; f is defined over the full set of simulation actions.

4.2.7 Channel Restricted Simulation under a Functional Interface

Channel restricted simulation under a functional interface resembles the previous relation, except that some actions of the simulation process might simulate no action from the simulated process.

Definition 4.8 *Channel Restricted Simulation under a Functional Interface* $\mathcal{S}_{cf}$*, for a non–empty set of channels* $C \subseteq (\mathsf{Chs}(P) \cup \mathsf{Chs}(Q))$ *and* $f\colon \mathsf{Sort}_i(P \upharpoonright_c C) \times \mathsf{Sort}(Q \upharpoonright_c C) \nrightarrow \mathsf{Sort}_o(P \upharpoonright_c C)$*, is a binary relation on processes such that if* $(P, Q) \in \mathcal{S}_{cf}$ *(abbreviated by* $P \overset{cf}{\sqsubseteq} Q$*), then for every "proper" action* $i_p/o_p, i_q/o_q \in Act$ *and* $i'_p \in \mathsf{Sort}_{1i}(P)$*:*

1. *whenever* $P \xrightarrow{i_p/o_p} P'$ *then* $Q \xrightarrow{i'_q/o'_q} Q'$ *for some* i'_q/o'_q *and* Q' *such that* $f(\textit{res-io-ch}(i_p, C), \textit{res-ch}(i'_q/o'_q, C)) = \textit{res-io-ch}(o_p, C)$ *and* $P' \overset{cf}{\sqsubseteq} Q'$*,*
2. *whenever* $Q \xrightarrow{i_q/o_q} Q'$ *and* $(\textit{res-io-ch}(i'_p, C), \textit{res-ch}(i_q/o_q, C)) \in \mathsf{dom}\, f$ *then* $P \xrightarrow{i'_p/o'_p} P'$ *for some* o'_p *and* P' *such that* $P' \overset{cf}{\sqsubseteq} Q'$ *and* $f(\textit{res-io-ch}(i'_p, C), \textit{res-ch}(i_q/o_q, C)) = \textit{res-io-ch}(o'_p, C)$*.*

Let us illustrate this relation by showing the similarities between processes $P_{fcsim}(c_7 c_8 c_9/c_{10}\, c_{11} c_{12})$ and $S(c_1 c_2 c_3/c_4 c_5 c_6)$. Figure 4.13 shows the action–value transition graphs for both processes where the sequence of actions are passed through channels as indicated in the definition of those processes.

Notice that P_{fcsim} is a simulation of P_{tfcsim} (defined in the previous section). As a result, if S and P_{fcsim} are restricted to the sets of channels as previously defined ($\{c_1, c_2, c_3, c_4, c_5\}$ for S and $\{c_8, c_9, c_{10}, c_{11}, c_{12}\}$ for P_{fcsim}), those interface functions defined there ($f'_{S\text{-}Pfcrbis}, f''_{S\text{-}Pfcrbis}$) can also be used to interface S and P_{fcsim}. By using one of those functions, P_{fcsim} actions that are on the right–hand side of the graph, starting from 110/101, are left behind. Other functions can also interface S and P_{fcsim} when some of those actions of P_{fcsim} not present in P_{tfcsim} are used.

4.3 Similarities between the Traffic Light Controllers

Considering the similarity relations defined in the previous section, the traffic light controllers presented in Chapter 3 can now have their behavioural similarities checked. For all of those examples, we may

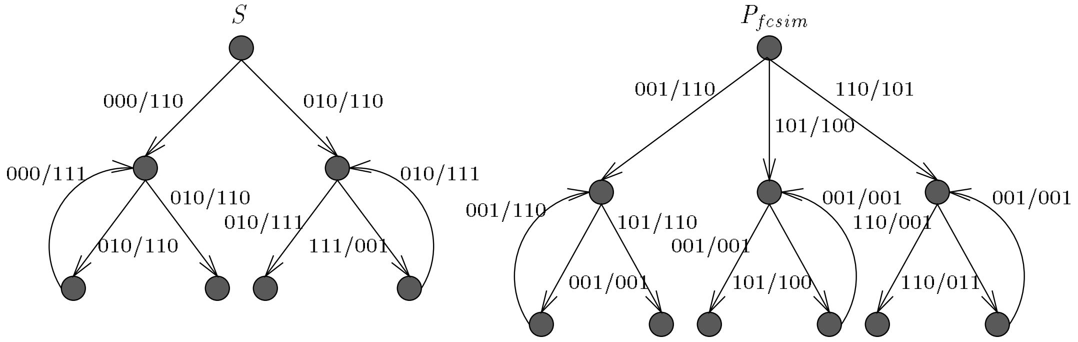

Figure 4.13: Processes $S(c_1\,c_2\,c_3/\,c_4\,c_5\,c_6)$ and $P_{fcsim}(c_7\,c_8\,c_9/\,c_{10}\,c_{11}\,c_{12})$

check how close the processes are in behaviour in order to figure out what still has to be developed to obtain the desired specification.

Regarding the development of the major/minor roads traffic light controller ($TLC_2(s$: *sensor*, t: *intervs*$/l_{mj}, l_{mn}$: *colour*, *setc*: *stcount*$)$) when the major road process ($TLC_{mj}(s$: *sensor*, t: *intervs*$/l_{mj}$: *colour*, *setc*: *stcount*$)$) is a library component, both processes are defined over a subset of identical channels: $\{s, t, l_{mj}, setc\}$. Their input channels coincide but the traffic light controller has an output channel not defined for the major road process.

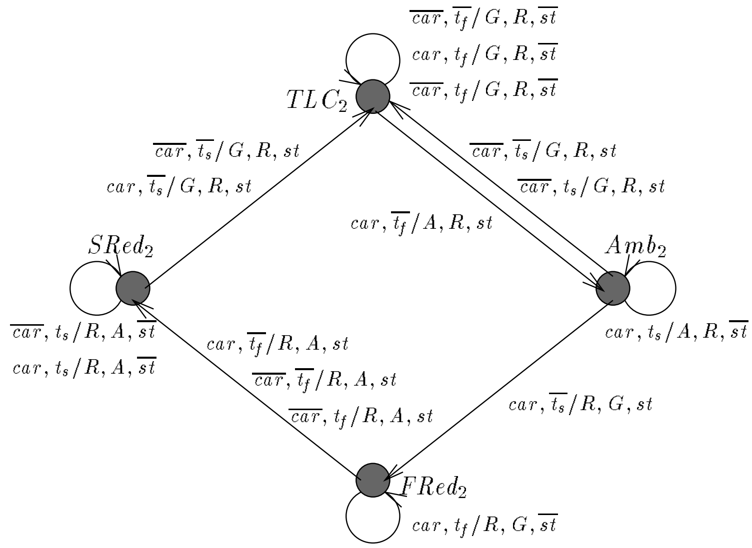

Figure 4.14: Transition Graph for $TLC_2(s, t/l_{mj}, l_{mn}, setc)$

To compare these processes, let us then provide their behavioural transition graphs (Figure 4.14 – TLC_2's actions are represented by $(s, t/l_{mj}, l_{mn}, setc)$ – and Figure 4.15 – TLC_{mj} has actions $(s, t/ l_{mj}, setc)$). No functions can be found to identify actions of these processes, but if both processes are restricted to the set of channels $\{s, t, l_{mj}, setc\}$, we may check they are channel restricted bisimilar. Process TLC_{mj}, for example, can perform all the restricted actions for which TLC_2 is able to move. Transition $TLC_2 \xrightarrow{\overline{t_f},\, car/A,\, R,\, st} Amb_2$, for example, is channel restricted identical to $TLC_{mj} \xrightarrow{\overline{t_f},\, car/A,\, st} Amb_1$. TLC_2 and TLC_{mj} have identical sets of channel restricted initial actions and, besides that, they reach processes that are also channel restricted bisimilar. These processes are then channel restricted bisimilar: $TLC_2 \overset{c}{\sim} TLC_{mj}$, $Amb_2 \overset{c}{\sim} Amb_{mj}$, $FRed_2 \overset{c}{\sim} FRed_{mj}$ and $SRed_2 \overset{c}{\sim} SRed_{mj}$.

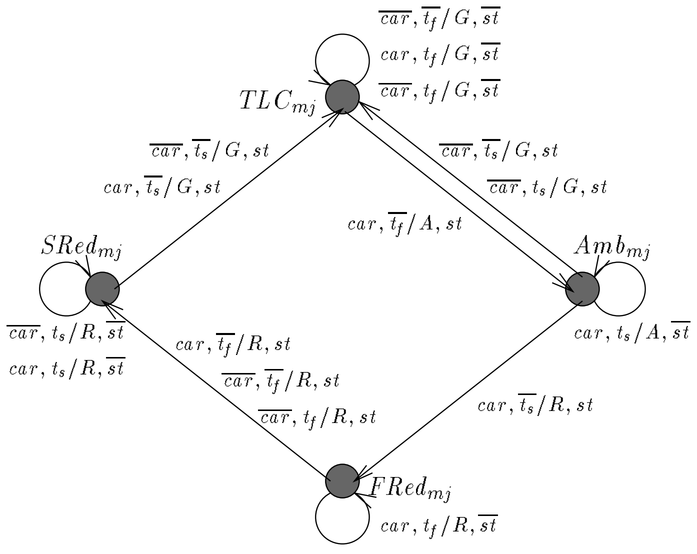

Figure 4.15: Transition Graph for $TLC_{mj}(s, t/l_{mj}, setc)$

We also proposed the development of a traffic light controller in a perfect environment, when the traffic light controller which responds to certain abnormal cases has already been designed. Both processes are defined over identical sets of channels ($TLC_2(s\colon\ sensor, t\colon\ intervs/l_{mj}, l_{mn}\colon\ colour, setc\colon\ stcount)$ and $TLC_{perf}(s\colon\ sensor, t\colon\ intervs/l_{mj}, l_{mn}\colon\ colour, setc\colon\ stcount)$), they can then be compared using the full set of channels – Figure 4.16 shows a transition graph for the traffic light controller embedded in a perfect environment. In doing so, it can be checked that the initial actions of TLC_2 is able to simulate all initial actions of TLC_{perf}. Furthermore, all initial actions of Amb_{perf} can also be performed by Amb_2. However, actions $(\overline{car}, \overline{t_s}/G, R, st)$ and $(\overline{car}, t_s/G, R, st)$ from Amb_2 cannot be performed by Amb_{perf}. TLC_2 is then a simulation for TLC_{perf} because $TLC_{perf} \sqsubseteq TLC_2$, $Amb_{perf} \sqsubseteq Amb_2$, $FRed_{perf} \sqsubseteq FRed_2$ and $SRed_{perf} \sqsubseteq SRed_2$.

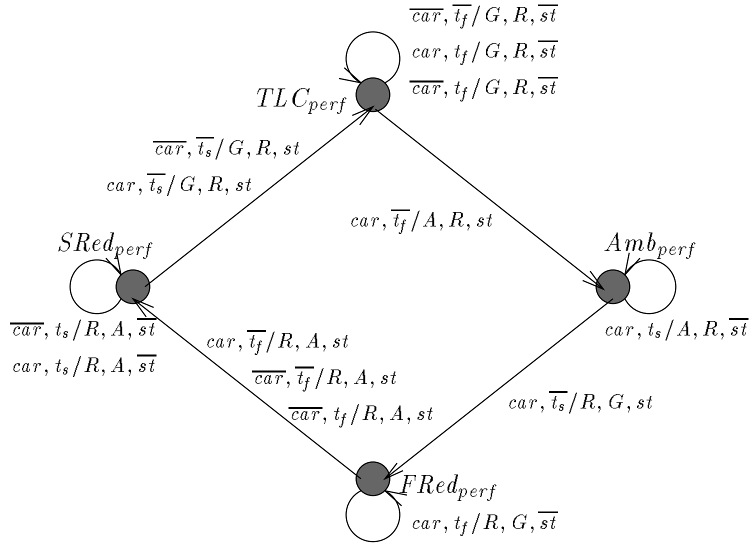

Figure 4.16: Transition Graph for $TLC_{perf}(s, t/l_{mj}, l_{mn}, setc)$

And finally, how similar are the pedestrian/road and the major/minor roads traffic light controllers ($TLC_2(s, t/l_{mj}, l_{mn}, setc)$ is in Figure 4.14, and $TLC_{ped}(b, t/l_r, l_{pd}, setc)$ is in Figure 4.17)? These processes are defined over distinct sorts. So, they can only be compared by making abstraction on their sorts. In doing so, function $f_{TLC_{ped}\text{-}TLC_2}$ (Table 4.4) can be found to identify all actions from TLC_{ped} with actions performed by TLC_2.

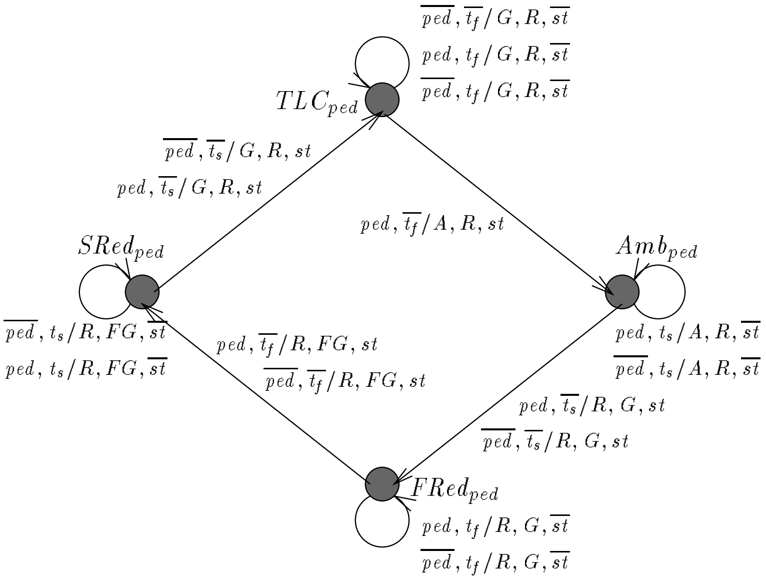

Figure 4.17: Transition Graph for $TLC_{ped}(b, t/l_r, l_{pd}, setc)$

Since such a function can be found, the major/minor roads traffic light controller is a simulation for the pedestrian/road process via application of $f_{TLC_{ped}\text{-}TLC_2}$. Action $(ped, \overline{t_f}/A, R, st)$ from TLC_{ped}, for example, is simulated by $(car, \overline{t_f}/A, R, st)$ from TLC_2 under application of $f_{tlcd\text{-}TLC_{ped}}$. Actions $(\overline{car}, \overline{t_f}/G, R, st)$ and $(\overline{car}, t_s/G, R, st)$ from Amb_2 are unable to simulate any action from Amb_{ped}. Since no total functions can be found to interface these processes, the major/minor roads process is then a simulation under a functional interface of the pedestrian/road process. Other functions could actually be found to interface these processes, but we confine ourselves to showing only this one in order to avoid unnecessary information.

4.4 Classifying Similarity Relations

Bisimulation is a relation necessary and sufficient for substitutivity. For processes related under simulation, only the simulation process is able to behave like the simulated one. The other similarity relations underlying reusability make abstraction over certain elements to realize behavioural similarity of processes. Comparing processes by abstraction of certain elements provides relations weaker than those ones that compare the full range of processes elements. Channel restricted bisimulation, for example, is weaker than bisimulation. In this section we classify the various similarity relations introduced in the current chapter:

$f_{TLC_{ped}\text{-}TLC_2}$		
$((\overline{ped}, \overline{t_f}), (\overline{car}, \overline{t_f}/G, R, \overline{st}))$	$\mapsto$	$(G, R, \overline{st})$
$((ped, t_f), (car, t_f/G, R, \overline{st}))$	$\mapsto$	$(G, R, \overline{st})$
$((\overline{ped}, t_f), (\overline{car}, t_f/G, R, \overline{st}))$	$\mapsto$	$(G, R, \overline{st})$
$((ped, \overline{t_f}), (car, \overline{t_f}/A, R, st))$	$\mapsto$	(A, R, st)
$((ped, t_s), (car, t_s/A, R, \overline{st}))$	$\mapsto$	$(A, R, \overline{st})$
$((\overline{ped}, t_s), (car, t_s/A, R, \overline{st}))$	$\mapsto$	$(A, R, \overline{st})$
$((ped, \overline{t_s}), (car, \overline{t_s}/R, G, st))$	$\mapsto$	(R, G, st)
$((\overline{ped}, \overline{t_s}), (car, \overline{t_s}/R, G, st))$	$\mapsto$	(R, G, st)
$((ped, t_f), (car, t_f/R, G, \overline{st}))$	$\mapsto$	$(R, G, \overline{st})$
$((\overline{ped}, t_f), (car, t_f/R, G, \overline{st}))$	$\mapsto$	$(R, G, \overline{st})$
$((ped, \overline{t_f}), (car, \overline{t_f}/R, A, st))$	$\mapsto$	(R, FG, st)
$((\overline{ped}, \overline{t_f}), (\overline{car}, \overline{t_f}/R, A, st))$	$\mapsto$	(R, FG, st)
$((\overline{ped}, t_s), (\overline{car}, t_s/R, A, \overline{st}))$	$\mapsto$	$(R, FG, \overline{st})$
$((ped, t_s), (car, t_s/R, A, \overline{st}))$	$\mapsto$	$(R, FG, \overline{st})$
$((\overline{ped}, \overline{t_s}), (\overline{car}, \overline{t_s}/G, R, st))$	$\mapsto$	(G, R, st)
$((ped, \overline{t_s}), (car, \overline{t_s}/G, R, st))$	$\mapsto$	(G, R, st)

Table 4.4: Interfacing Pedestrian/Road and Major/Minor Roads Controllers

$\mathcal{B}$	-	Bisimulation
$\mathcal{S}$	-	Simulation
$\mathcal{B}_c$	-	Channel Restricted Bisimulation
$\mathcal{S}_c$	-	Channel Restricted Simulation
$\mathcal{S}_{tf}$	-	Simulation under a Total Functional Interface
$\mathcal{S}_f$	-	Simulation under a Functional Interface
$\mathcal{S}_{ctf}$	-	Channel Restricted Simulation under a Total Functional Interface
$\mathcal{S}_{cf}$	-	Channel Restricted Simulation under a Functional Interface.

We obtain the following diagram (Figure 4.18) in which the strongest relations are at the top and all the other following relations are deduced by the top ones.

Bisimulation is the strongest similarity relation presented here, it hence deduces all the other relations. In other words, if P and Q are bisimilar, they are also similar, channel restricted bisimilar, etc. Conversely, channel restricted simulation under a functional interface is the weakest similarity relation. Thus, all the other relations deduce it.

The following theorem states all the relations shown in the diagram presented above. Looking at the diagram from top to bottom, we can see successive deductions. These deductions are stated as follows:

Theorem 4.1 *$\mathcal{B} \subseteq \mathcal{S}$, $\mathcal{B} \subseteq \mathcal{S}_{tf}$, $\mathcal{B} \subseteq \mathcal{B}_c$, $\mathcal{S} \subseteq \mathcal{S}_f$, $\mathcal{S} \subseteq \mathcal{S}_c$, $\mathcal{S}_{tf} \subseteq \mathcal{S}_f$, $\mathcal{S}_{tf} \subseteq \mathcal{S}_{ctf}$, $\mathcal{B}_c \subseteq \mathcal{S}_c$, $\mathcal{B}_c \subseteq \mathcal{S}_{ctf}$, $\mathcal{S}_{ctf} \subseteq \mathcal{S}_{cf}$, $\mathcal{S}_f \subseteq \mathcal{S}_{cf}$ and $\mathcal{S}_c \subseteq \mathcal{S}_{cf}$.*

A proof for all these relations is trivial from definitions of the similarity relations. A validation for this theorem is shown in Appendix C, and only certain relations are shown here.

- $\mathcal{B} \subseteq \mathcal{S}$: If $P \sqsubseteq Q$, for every "proper" action i_p/o_p such that i_p is in the input sort of P initial actions, both processes perform identical actions. In bisimulation, both processes must behave

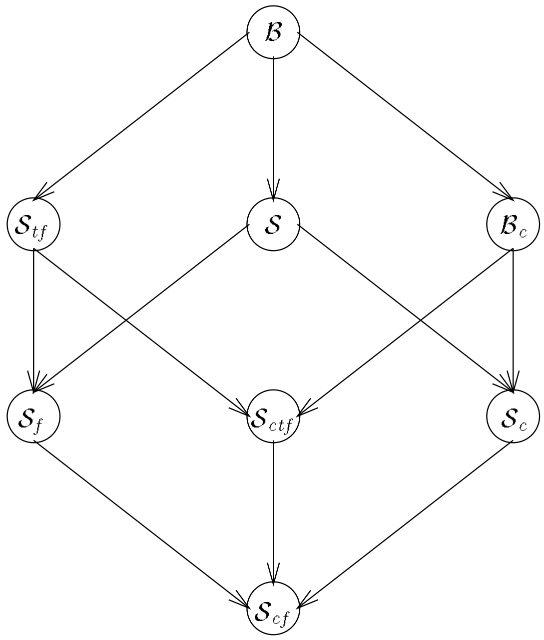

Figure 4.18: Classifying Similarity Relations

equivalently for any proper action, while in simulation only the P actions are taken into account. Bisimulation is obviously an instance of simulation where processes behave equivalently for any proper action rather than P's proper actions. As being an instance of simulation, all bisimilar processes are also similar: $\mathcal{B} \subseteq \mathcal{S}$.

- $\mathcal{B} \subseteq \mathcal{B}_c$: Looking at the definition of $\mathcal{B}$, we can, with no loss of generality, redefine it as a symmetric binary relation such that if $(P, Q) \in \mathcal{B}$ and $\mathsf{Chs}(P) = \mathsf{Chs}(Q) = C$ then for every "proper" action $\alpha \in Act$:

  1. whenever $P \xrightarrow{\alpha} P'$ then $Q \xrightarrow{\alpha'} Q'$ for some α' and Q' such that $\textit{res-ch}(\alpha, C) = \textit{res-ch}(\alpha', C)$ and $P' \sim Q'$,
  2. whenever $Q \xrightarrow{\alpha} Q'$ then $P \xrightarrow{\alpha'} P'$ for some α' and P' such that $\textit{res-ch}(\alpha, C) = \textit{res-ch}(\alpha', C)$ and $P' \sim Q'$.

  Observe that $\textit{res-ch}(\alpha, C) = \textit{res-ch}(\alpha', C)$ is reduced to $\alpha = \alpha'$ because all channel names of actions α and α' are in C. Like bisimulation, channel restricted bisimulation can also be redefined as a symmetric binary relation such that if $(P, Q) \in \mathcal{B}_c$ and $\mathsf{Chs}(P) \cap \mathsf{Chs}(Q) = C$ then for every "proper" action $\alpha \in Act$:

  1. whenever $P \xrightarrow{\alpha} P'$ then $Q \xrightarrow{\alpha'} Q'$ for some α' and Q' such that $\textit{res-ch}(\alpha, C) = \textit{res-ch}(\alpha', C)$ and $P' \sim Q'$,
  2. whenever $Q \xrightarrow{\alpha} Q'$ then $P \xrightarrow{\alpha'} P'$ for some α' and P' such that $\textit{res-ch}(\alpha, C) = \textit{res-ch}(\alpha', C)$ and $P' \sim Q'$.

  This definition resembles that one for bisimulaiton, except that $\mathsf{Chs}(P) \cap \mathsf{Chs}(Q) = C$ appears instead of $\mathsf{Chs}(P) = \mathsf{Chs}(Q) = C$. The constraint $\mathsf{Chs}(P) \cap \mathsf{Chs}(Q) = C$ is less restrictive than that one for bisimulation. Bisimulation is therefore an instance of channel restricted bisimulation such that $\mathsf{Chs}(P) = \mathsf{Chs}(Q)$, and then $\mathcal{B} \subseteq \mathcal{B}_c$.

- $\mathcal{S} \subseteq \mathcal{S}_f$: For processes P and Q related by $\mathcal{S}_f$, the existence of a function f to identify behaviour of these processes is necessary. This means that for every P's initial action i_p/o_p, there exists an action i_q/o_q in Q such that $f(i_p, i_q/o_q) = o_p$. For instance, if f exclusively maps actions of P and Q that are identical, then no interface is necessary to identify actions of P and Q, and Q is indeed a simulation for P. Simulation is a particular case of simulation under a functional interface where f exclusively relates identical actions of P and Q.

- $\mathcal{S}_{ctf} \subseteq \mathcal{S}_{cf}$: Considering that $\mathcal{B} \subseteq \mathcal{S}$, $\mathcal{S} \subseteq \mathcal{S}_f$ and $\mathcal{B} \subseteq \mathcal{S}_{tf}$, we may conclude $\mathcal{S}_{tf} \subseteq \mathcal{S}_f$ – $\mathcal{S}_{tf}$ is an instance of $\mathcal{S}_f$ where the map on actions of both processes is an *onto function*. Also, $\mathcal{B}_c \subseteq \mathcal{S}_c$ is deduced from $\mathcal{B} \subseteq \mathcal{S}$ and $\mathcal{B} \subseteq \mathcal{B}_c$. $\mathcal{S}_{ctf} \subseteq \mathcal{S}_{cf}$ is thus deduced from $\mathcal{B}_c \subseteq \mathcal{S}_c$ and $\mathcal{S}_{tf} \subseteq \mathcal{S}_f$. Roughly speaking, $\mathcal{S}_{ctf}$ is a particular case of $\mathcal{S}_{cf}$ where all restricted actions from Q simulate at least one restricted action of P.

Summarizing, bisimulation deduces all the other similarity relations. By contrast, channel restricted simulation under a functional interface is deduced by disjunction of all those other relations.

4.5 Interchanging Similarity Relations

Any set can be restricted to its subsets by making constraints. The subsets could be defined by an explicit enumeration of all their elements (when they are finite), or even by definition of constraints over the full set. As we have seen in Theorem 4.1, certain similarity relations are subsets of others. Hence, constraints on a "superset" relation can be made to obtain a subset of pairs of processes (a stronger similarity relation).

Reuse of processes requires substitution of the given components for the desired specifications. Bisimulation guarantees processes to be used interchangeably, but some other similarity relations demand an interface process to yield the desired behaviour (interface processes shall be addressed in the forthcoming chapter). However, with the application of certain constraints to processes (related by one of the relations above), a stronger relation on the constrained processes can be found. Channel restricted bisimilar processes, for example, achieve bisimulation when restricted to the set of their coincident channels (vide definition of channel restricted bisimulation). This means that the constrained processes can substitute for each other.

The constraints applied to processes to achieve bisimilarity are twofold: they show which part of the given component is not going to be reused, and also the behaviour of the desired specification which cannot be simulated by the component. To some extent, processes can be constrained gradually until bisimulation of the constrained processes is found, and consequently substitution of the constrained processes is feasible. Likewise, under certain constraints, the simulation process can substitute for the simulated one[7]. So, we are particularly interested in constraining all the other relations to yield simulation or bisimulation.

In the sense of process algebras, constraints emerge with application of certain combinators. Once we have logically related the similarity relations (in the previous section), processes can now be restricted to become more similar (pass to a stronger similarity relation) by application of certain combinators. Several cases require application of combinators to both processes. In simpler cases, however, constraints applied to the component process is sufficient to achieve a stronger relation. Unfortunately, not every relation can be passed to the immediately stronger one by application of combinators. For instance, no combinator can be applied to processes related by simulation to turn them bisimilar. But, certain constraints can be defined to relate similar and bisimilar processes.

In this section we investigate which combinators, or constraints, are necessary to be applied to certain similar processes in order to turn the restricted processes similar or bisimilar. We first present several theorems to achieve simulation and bisimulation, and then show the constraints necessary to achieve bisimulation from processes related by simulation.

[7] Substitutivity of similar processes will be discussed in Chapter 5.

4.5.1 Achieving Bisimulation

From Simulation under a Total Functional Interface

Whenever processes are related by simulation under a total functional interface, there exists a function f, and consequently a corresponding process P_f (construction of P_f from f has been shown in Definition 4.3), which interfaces P with Q. Composition of the simulation process with the single state process (P_f) is indeed bisimilar to the simulated process.

Theorem 4.2 *$P \overset{f}{\approx} Q$ iff there exists a total function f such that $f\colon \mathsf{Sort}_i(P) \times \mathsf{Sort}(Q) \longrightarrow \mathsf{Sort}_o(P)$ and $P \sim (Q \mid P_f) \upharpoonright_c \mathsf{Chs}(P)$.*

Proof. P_f implements f's behaviour and, consequently, interfaces P with Q. Since the EPA composition combinator $(|)$ exposes all output actions, a simple composition of Q with P_f would expose all their output events. P_f is aimed at interfacing P with Q, then, connections between P_f and Q must be internalized (by application of the restriction to channels operator).

For the forward implication (if), the existence of f is proved by definition of $\mathcal{S}_{tf}$. It also states that for each action i_p/o_p from P there exists an action i_q/o_q in the Q's sort such that $f(i_p, i_q/o_q) = o_q$. But, for each pair of corresponding actions, i_p/o_p and i_q/o_q, there exists an action in P_f (denoted as $((i_p \oplus o_q) \ominus o_q/(i_q \oplus o_p) \ominus i_p))$. Assuming that input and output channels of EPA processes do not overlap, composition of Q action with the corresponding P_f action is given by:

$$
\begin{aligned}
(i_q/o_q) \star ((i_p \oplus o_q) \ominus o_q/(i_q \oplus o_p) \ominus i_p)&\\
&= ((i_q \oplus (i_p \oplus o_q) \ominus o_q) \ominus (o_q \oplus (i_q \oplus o_p) \ominus i_p))/(o_q \oplus (i_q \oplus o_p) \ominus i_p)\\
&= i_p/(o_q \oplus i_q \oplus o_p \ominus i_p).
\end{aligned}
$$

But, the compound actions are restricted to the set of P channels, and then

$$\textit{res-ch}(i_p/(o_q \oplus i_q \oplus o_p \ominus i_p), \mathsf{Chs}(P)) = (i_p/o_p).$$

Thus, extending this argument throughout the set of P actions, we find composition of Q with P_f bisimilar to P when all actions are restricted to the set of P's channels.

The backward implication (only if) is proved by using similar arguments: the existence of f with such a signature is proved by definition, and due to bisimilarity and properties of synchronous composition, each action in the compound process equals a P's action. The equality of these actions is given as above. P and Q are then similar under a total functional interface.

$\square$

From Channel Restricted Bisimulation

The relation between bisimilarity and channel restricted bisimilarity is given as follows:

Theorem 4.3 *$P \overset{c}{\sim} Q$ iff there exists a non-empty C such that $C = (\mathsf{Chs}(P) \cap \mathsf{Chs}(Q))$ and $P \upharpoonright_c C \sim Q \upharpoonright_c C$.*

This theorem is trivially proved by the definition of channel restricted bisimulation.

From Channel Restricted Simulation under a Total Functional Interface

Three other similarity relations are stronger than $\mathcal{S}_{ctf}$: $\mathcal{S}_{tf}$, $\mathcal{B}_c$ and $\mathcal{B}$. In order to state a property for relating channel restricted similar processes under a total functional interface with the corresponding bisimilar ones, we must first analyse the relation with those immediately stronger relations[8]: $\mathcal{S}_{tf}$ and $\mathcal{B}_c$.

[8] A validation for the following propositions has been included in Appendix C.

Proposition 4.1 *$P \overset{cf}{\approx} Q$ iff there exists a non-empty C such that $C \subseteq (\mathsf{Chs}(P) \cup \mathsf{Chs}(Q))$ and $P \upharpoonright_c C \overset{f}{\approx} Q \upharpoonright_c C$.*

If processes are related by $\mathcal{S}_{ctf}$, each channel restricted action from P is identified with a channel restricted action from Q, through f. So, if both processes are restricted to this set of channels, the existence of a function to identify these processes is still ensured by definition of $\mathcal{S}_{ctf}$, and the channel restricted processes are then related by $\mathcal{S}_{tf}$.

When processes P and Q are related by $\mathcal{S}_{tf}$, P_f is able to interface P with Q. For P and Q related by $\mathcal{S}_{ctf}$, however, P_f is defined over a restricted set of channels of both processes, and then its composition with Q is only able to simulate a channel restricted P:

Proposition 4.2 *$P \overset{cf}{\approx} Q$ iff there exists a non-empty C and a total function f such that $C \subseteq (\mathsf{Chs}(P) \cup \mathsf{Chs}(Q))$, $f \colon \mathsf{Sort}_i(P \upharpoonright_c C) \times \mathsf{Sort}(Q \upharpoonright_c C) \rightarrow \mathsf{Sort}_o(P \upharpoonright_c C)$ and $P \overset{c}{\sim} (Q \mid P_f)$.*

Due to the fact that f is applied to channel restricted actions of both processes, bisimilarity can only be found for channel restricted processes, and subjected to a composition with a process that implements such a function. So, if $P \overset{cf}{\approx} Q$, certain operators can be applied to Q to obtain a process bisimilar to a channel restricted P.

Theorem 4.4 *$P \overset{cf}{\approx} Q$ iff there exists a non-empty C and a total function f such that $C \subseteq (\mathsf{Chs}(P) \cup \mathsf{Chs}(Q))$, $f \colon \mathsf{Sort}_i(P \upharpoonright_c C) \times \mathsf{Sort}(Q \upharpoonright_c C) \rightarrow \mathsf{Sort}_o(P \upharpoonright_c C)$ and $P \upharpoonright_c C \sim (Q \mid P_f) \upharpoonright_c (C \cap \mathsf{Chs}(P))$.*

Proof. By Proposition 4.2, if $P \overset{cf}{\approx} Q$ then $P \overset{c}{\sim} (Q \mid P_f)$. From Theorem 4.3, if processes are channel restricted bisimilar, their constrained processes are bisimilar if restricted to the set of their identical channels ($P \upharpoonright_c C \sim Q \upharpoonright_c C$). Notice that the set of channel names identical in processes P and $(Q \mid P_f)$ is $(C \cap \mathsf{Chs}(P))$ because P_f is defined over C. So, the above expression deduces

$$P \upharpoonright_c (C \cap \mathsf{Chs}(P)) \sim (Q \mid P_f) \upharpoonright_c (C \cap \mathsf{Chs}(P))$$

and then,

$$P \upharpoonright_c C \sim (Q \mid P_f) \upharpoonright_c (C \cap \mathsf{Chs}(P))$$

can be concluded because $P \upharpoonright_c C \sim P \upharpoonright_c (C \cap \mathsf{Chs}(P))$ holds.

□

4.5.2 Achieving Simulation

Here we present how processes related by $\mathcal{S}_f$, $\mathcal{S}_c$ and $\mathcal{S}_{cf}$ can be constrained to $\mathcal{S}$.

From Simulation under a Functional Interface

Again, the construction of P_f is straightforward from processes related by simulation under a functional interface (Definition 4.3). Composition of the simulation process under a functional interface with a single state process P_f is a simulation for the simulated process.

Theorem 4.5 *$P \overset{f}{\sqsubseteq} Q$ iff there exists a partial function f such that $f \colon \mathsf{Sort}_i(P) \times \mathsf{Sort}(Q) \nrightarrow \mathsf{Sort}_o(P)$ and $P \sqsubseteq (Q \mid P_f) \upharpoonright_c \mathsf{Chs}(P)$.*

Proof. By the definition of $\mathcal{S}_f$, for each action i_p/o_p of P, there exists a corresponding i_q/o_q from Q such that $f(i_p, i_q/o_q) = o_p$ (the existence of f is proved by definition). For this particular map, the corresponding action is represented in P_f by $(i_p \oplus o_q) \ominus o_p/(i_q \oplus o_q) \ominus i_p$.

Notice that P_f actions depend on P as much as Q's actions. Then, for each action i_p/o_p from P, there is an observable action from $(Q \mid P_f) \upharpoonright_c \mathsf{Chs}(P)$ given by:

$$res\text{-}ch((i_q/o_q) \star ((i_p \oplus o_q) \ominus o_p/(i_q \oplus o_q) \ominus i_p), \mathsf{Chs}(P))$$

which equals i_p/o_p[9]. This equality is achieved for all the P's actions, and then composition of Q with P_f is a simulation for P when restricted to P's channels.

$\square$

At first glance, composition of Q with P_f (restricted to P's channels) looks bisimilar to P because P_f can only move by undertaking actions that are related to P's. However, Q can do all moves that P does or even more. These "spare actions" appearing in Q could be combined with actions of P_f to perform actions not present in P. For instance, suppose P and Q can move for actions

$$P \xrightarrow{i_p/o_p} P', \quad P' \xrightarrow{i_{p'}/o_{p'}} P'',$$
$$Q \xrightarrow{i_q/o_q} Q', \quad Q' \xrightarrow{i_{q'}/o_{q'}} Q''' \text{ and } \quad Q \xrightarrow{i_{q'}/o_{q'}} Q'',$$

and $P \overset{f}{\sqsubseteq} Q$ (and $P' \overset{f}{\sqsubseteq} Q'$) where $f = \{(i_p, i_q/o_q) \mapsto o_q, (i_{p'}, i_{q'}/o_{q'}) \mapsto o_{p'}\}$. Composition of Q with P_f, restricted to P channels, can make a move not performed by P:

$$(Q \mid P_f) \restriction_c \mathsf{Chs}(P) \xrightarrow{i_p/o_p} (Q' \mid P_f) \restriction_c \mathsf{Chs}(P),$$
$$(Q \mid P_f) \restriction_c \mathsf{Chs}(P) \xrightarrow{i_{p'}/o_{p'}} (Q'' \mid P_f) \restriction_c \mathsf{Chs}(P)$$
$$(Q' \mid P_f) \restriction_c \mathsf{Chs}(P) \xrightarrow{i_{p'}/o_{p'}} (Q''' \mid P_f) \restriction_c \mathsf{Chs}(P).$$

The "spare action" $Q \xrightarrow{i_{q'}/o_{q'}} Q''$ can be combined with an action from P_f and, as a result, this compound process has also $i_{p'}/o_{p'}$ as initial action (instead of only performing i_p/o_p like P). Therefore, the compound process is indeed a simulation of P rather than a bisimulation.

From Channel Restricted Simulation

Processes related by $\mathcal{S}_c$ can be constrained in such a way that the constrained processes achieve simulation:

Theorem 4.6 *$P \overset{c}{\sqsubseteq} Q$ iff there exists a non-empty C such that $C = (\mathsf{Chs}(P) \cap \mathsf{Chs}(Q))$ and $P \restriction_c C \sqsubseteq Q \restriction_c C$.*

This theorem is a straightforward result from definition of channel restricted simulation.

From Channel Restricted Simulation under a Functional Interface

This is the weakest similarity relation. It can thus be constrained to achieve any other similarity relation. Propositions can be defined to show how processes related by $\mathcal{S}_{cf}$ can be constrained to the immediately stronger relations: $\mathcal{S}_f$ and $\mathcal{S}_c$. These propositions are given to elucidate how channel restricted similar processes under a functional interface are constrained to achieve simulation (the validations of the following propositions are included in Appendix C).

If processes P and Q are related by $\mathcal{S}_{cf}$, then composition of Q with P_f (a process which implements the interface function f) is a channel restricted simulation of P. Also, the channel restricted P and Q are similar under a functional interface:

Proposition 4.3 *$P \overset{cf}{\sqsubseteq} Q$ iff there exists a non-empty C and a partial function f such that $C \subseteq (\mathsf{Chs}(P) \cup \mathsf{Chs}(Q))$, $f\colon \mathsf{Sort}_i(P \restriction_c C) \times \mathsf{Sort}(Q \restriction_c C) \nrightarrow \mathsf{Sort}_o(P \restriction_c C)$ and $P \overset{c}{\sqsubseteq} (Q \mid P_f)$.*

[9] This result has already been shown in Theorem 4.2.

Proposition 4.4 *$P \overset{cf}{\sqsubseteq} Q$ iff there exists a non-empty C such that $C \subseteq (\mathsf{Chs}(P) \cup \mathsf{Chs}(Q))$ and $P \upharpoonright_c C \overset{f}{\sqsubseteq} Q \upharpoonright_c C$.*

Apart from constraining channel restricted similar processes under a functional interface to achieve $\mathcal{S}_f$ and $\mathcal{S}_c$, these processes can also be constrained to accomplish simulation:

Theorem 4.7 *$P \overset{cf}{\sqsubseteq} Q$ iff there exists a non-empty C and a partial function f such that $C \subseteq (\mathsf{Chs}(P) \cup \mathsf{Chs}(Q))$, $f\colon \mathsf{Sort}_i(P \upharpoonright_c C) \times \mathsf{Sort}(Q \upharpoonright_c C) \nrightarrow \mathsf{Sort}_o(P \upharpoonright_c C)$ and $P \upharpoonright_c C \sqsubseteq (Q \mid P_f) \upharpoonright_c (C \cap \mathsf{Chs}(P))$.*

Proof. By Proposition 4.4, if $P \overset{f}{\sqsubseteq} Q$ then $P \upharpoonright_c C \overset{f}{\sqsubseteq} Q \upharpoonright_c C$. Using Theorem 4.6, we can rewrite the above expression as $P \upharpoonright_c C \sqsubseteq ((Q \upharpoonright_c C) \mid P_f) \upharpoonright_c \mathsf{Chs}(P)$, and then obtain the expression

$$P \upharpoonright_c C \sqsubseteq (Q \mid P_f) \upharpoonright_c (C \cap \mathsf{Chs}(P))$$

from laws defined for $\upharpoonright_c$ (Section 2.5, page 30).

□

4.5.3 From Simulation to Bisimulation

We have shown so far that processes related by any of the presented similarity relations can be constrained to achieve simulation or bisimulation. However, no relation has been stated to turn similar processes into bisimilar ones. The interest of having such a property relies on the fact that bisimulation is the only relation which ensures unrestricted substitutivity.

If $P \sqsubseteq Q$, Q can do everything P does, or even more. So, how could Q be restricted to become bisimilar to P? This is one of the cases where application of combinators is insufficient. But, a constraint can be found to turn similar processes into bisimilar ones:

Proposition 4.5 *If $P \sqsubseteq Q$ and whenever $P \xrightarrow{\alpha} P'$ and $Q \xrightarrow{\alpha} Q'$ such that $P' \sim Q'$ then if $Q_{\mathsf{Sort}_1(P)}$ represents Q having its set of initial actions restricted to P initial actions, $P \sim Q_{\mathsf{Sort}_1(P)}$.*

This property is straightforward from definitions of simulation and bisimulation. If all corresponding initial actions from P and Q reach bisimilar processes, then P will be bisimilar to Q when Q initial actions are restricted to the set of P initial actions ($Q_{\mathsf{Sort}_1(P)}$).

In spite of being very restrictive, this property drives to a simple algorithm to check similarity from algorithms defined for bisimulation. Since various algorithms for checking bisimulation have been proved correct, algorithms for deciding simulation can be derived from bisimulation algorithms and the proposition above.

4.6 Algorithms for Deciding Similarities

Several algorithms have been presented in the literature of process algebras for deciding bisimulation of finite state processes, see [PT87, Fer90, KS90, FM91, GM92] for example. For deciding the similarity relations presented in Section 4.2, however, a few algorithms must be presented to show their decidability.

Since bisimulation is stronger than simulation, algorithms for deciding simulation are suitable for checking bisimulation: if $P \sqsubseteq Q$ and $Q \sqsubseteq P$, then $P \sim Q$[10]. Likewise, channel restricted bisimulation can be checked by algorithms for deciding channel restricted simulation, $\mathcal{S}_{tf}$ can be decided from algorithms for $\mathcal{S}_c$, and $\mathcal{S}_{ctf}$ can be checked by $\mathcal{S}_f$'s algorithm. So, the algorithms presented here are all defined for the weaker versions of the similarity relations.

Our algorithm to decide simulation of finite state processes is based on the idea of coarsest partition [KS90] given by Paige and Tarjan in [PT87] and further adapted to bisimulation problem by Fernandez

[10] Considering our simulation relation, processes are bisimilar if they simulate each other (this property does not hold for most simulation relations).

[Fer90]. The algorithms to decide the other similarity relations are based upon the simulation algorithm as well as upon results from Section 4.5.

Firstly, the idea of coarsest partition is presented; it is then carried over to an abstract algorithm to decide, respectively, simulation, channel restricted simulation, simulation under a functional interface, and channel restricted simulation under a functional interface.

4.6.1 Coarsest Partitions and Bisimulation

Given a finite state process P represented as a labelled transition system, $TS(P) = (\mathsf{Sts}_P, \mathcal{A}_P, T^P, s_0^P)$, an equivalence relation $\underline{R}$ over Sts_P and an initial partition ρ (on Sts_P), the relational coarsest partition problem is that of finding the coarsest refinement[11] of ρ which is compatible with the binary relation $\underline{R}$. If ρ' is the coarsest refinement of ρ under $\underline{R}$, then ρ' is unique[12] and any other partition of Sts_P under $\underline{R}$ and compatible with ρ is a refinement of ρ'. In other words, ρ' is the partition of Sts_P which contains the fewest number of blocks, and each block B_i^P is contained in an equivalence class of Sts_P under $\underline{R}$ ($\forall s_1, s_2 \in B_i^P \cdot s_1 \underline{R} s_2$).

Suppose a process P has its behaviour represented by a labelled transition system containing the set of states $\mathsf{Sts}_P = \{s_1, s_2, s_3, s_4\}$. A binary relation $R = \{(s_1, s_2), (s_1, s_4), (s_2, s_4)\}$, and an initial partition $\rho = \{\{s_1, s_2, s_3\}, \{s_4\}\}$ are given over the set of P states. The coarsest partition of Sts_P under R and compatible with ρ is given by $\rho' = \{\{s_1, s_2\}, \{s_3\}, \{s_4\}\}$. As s_3 is not related with s_1 and s_2 by R, it cannot form an equivalence class with those other processes. So, it must be placed in a separate block. s_4 is related with s_1 and s_2 but the initial partition ρ does not have any block containing all these states. To be compatible with ρ, s_4 must then be kept in a separate block. ρ' is therefore the coarsest refinement of ρ under R.

If processes P and Q are bisimilar, for each equivalence class of Sts_P under bisimulation, B_i^P, there exists an equivalence class B_j^Q of Sts_Q which is bisimilar to B_i^P and vice–versa. Consequently, each element of the coarsest partition of P under bisimulation, $\{B_1^P, ..., B_n^P\}$, has an equivalent element in the coarsest partition of Q under bisimulation and vice–versa.

Given a partition $\rho = \{B_i^P \mid i \in 1, ..n\}$ on P states, and a binary relation T_α^P upon pairs of states that are ,respectively, outcoming and incoming states of transitions of P labelled by action α ($T_\alpha^P = \{(s, s') \mid s, s' \in \mathsf{Sts}_P \wedge (s \xrightarrow{\alpha} s') \in T^P\}$),

- $T_\alpha^{P^{-1}}[X]$ is defined as $\{s \mid s' \in X \wedge (s, s') \in T_\alpha^P\}$;
- partition ρ is *compatible* with $T_\alpha^{P^{-1}}$ if and only if $\forall B_i^P, B_j^P \in \rho \cdot B_j^P \subseteq T_\alpha^{P^{-1}}[B_i^P] \vee B_j^P \cap T_\alpha^{P^{-1}}[B_i^P] = \emptyset$;
- in particular, ρ is compatible with T^P if and only if it is compatible with $T_\alpha^{P^{-1}}$, for each $\alpha \in \mathcal{A}_P$;
- given a partition ρ on Sts_P, ρ is compatible with T^P if and only if it is a partition of Sts_P under bisimulation.

$T_\alpha^{P^{-1}}[X]$ gives the emerging states of transitions labelled by α that come into states in set X. The refinement of ρ under compatibility with T^P is a complete lattice and, as a result, a unique coarsest partition exists. This coarsest partition coincides with the equivalence classes of Sts_P under bisimulation[13].

The coarsest partition of processes under bisimulation, having the trivial partition (partition which has all states in a unique block) as the initial one, is indeed the minimal process under bisimulation. Bisimilarity between distinct processes can, therefore, be checked by matching the equivalence classes of both processes.

The algorithm defined by Fernandez, in [Fer90], decides bisimilarity between processes defined over identical sets of states. Here, we shall adapt such an algorithm for similarity relations which differ from

[11] The elements of a partition are called *blocks*. ρ' is a refinement of ρ if they are defined over identical sets and every block of ρ' is contained in a block of ρ.

[12] As ρ' is the stable partition of P under $\underline{R}$, it holds the property of being unique [Fer90].

[13] The reader is referred to [Fer90] for more details and proof of these properties.

bisimulation. We thus redefine the Fernandez's algorithm to consider processes defined over distinct sets of states. Consider processes P and Q represented by the transition systems $TS(P) = (\mathsf{Sts}_P, \mathcal{A}_P, T^P, s_0^P)$ and $TS(Q) = (\mathsf{Sts}_Q, \mathcal{A}_Q, T^Q, s_0^Q)$. Our algorithm for deciding bisimulation is as follows:

- The refinement of a pair of blocks B_i^P and B_j^Q with respect to a particular action α is given by[14]:

$$\begin{aligned}\Phi_{\alpha,(B^P,B^Q)}(\rho) = {} & \{(X \cap T_\alpha^{P^{-1}}[B^P], Y \cap T_\alpha^{Q^{-1}}[B^Q]) \mid (X, Y) \in \rho\} \cup \\ & \{(X - T_\alpha^{P^{-1}}[B^P], Y - T_\alpha^{P^{-1}}[B^Q]) \mid (X, Y) \in \rho\}\end{aligned}$$

- considering that actions of both processes range over the set $\{i_1/o_1, ..., i_n/o_n\} = (\mathcal{A}_P \cup \mathcal{A}_Q)$, $\Phi_{(B^P,B^Q)}(\rho)$ partition both state blocks with respect to all actions defined for both processes:

$$\Phi_{(B^P,B^Q)}(\rho) = \Phi_{i_1/o_1,(B^P,B^Q)} \circ \cdots \circ \Phi_{i_n/o_n,(B^P,B^Q)}(\rho)$$

- refinements of partitions of both processes are simultaneously performed by successively refining each pair of equivalence blocks. Given a partition of both processes by pairs of equivalence blocks, $\rho = \{(B_1^P, B_1^Q), ..., (B_n^P, B_n^Q)\}$, the refinement of this partition with respect to all equivalence classes is given by:

$$\Phi_\rho(\rho) = \Phi_{(B_1^P,B_1^Q)} \circ \cdots \circ \Phi_{(B_n^P,B_n^Q)}(\rho)$$

- let the pair of P and Q partitions be initially defined as $\rho = \{(\mathsf{Sts}_P, \mathsf{Sts}_Q)\}$. The coarsest stable partition of both processes is calculated by $\Phi(\rho)$. P and Q are bisimilar if the stable refinement of this initial partition contains no empty blocks.

$$\begin{aligned}\Phi(\rho) = {} & \mathsf{if}\ Stable(\Phi_\rho(\rho)) \\ & \mathsf{then}\ \Phi_\rho(\rho) \\ & \mathsf{else}\ \Phi(\Phi_\rho(\rho))\end{aligned}$$

Since the initial partition is given by a pair of the trivial partitions of both processes, the successive refinement with respect to all actions finds the stable coarsest partitions of both processes. Notice that for each action, both blocks have their stability checked at the same time; they are bisimilar if both of them can perform all actions identically. Moreover, the refinement of ρ with respect to each pair of blocks, finds a stable partition considering all actions[15].

In the algorithm by Fernandez, partitions of processes are not given as pairs of blocks, but in two separate partitions. Since both processes are defined over identical sets of states there, blocks of both partitions can be identified by states names. This is the essential difference between the Fernandez's and our algorithms.

Suppose processes P and Q have their behaviour represented by the transition systems[16]:

$$\begin{aligned}& TS(P) = (\mathsf{Sts}_P, \mathcal{A}_P, T^P, s_0^P) \\ & TS(Q) = (\mathsf{Sts}_Q, \mathcal{A}_Q, T^Q, s_0^Q) \\ & \mathsf{Sts}_P = \{s_0^P, s_1^P, s_2^P, s_3^P, s_4^P, s_5^P\} \\ & \mathsf{Sts}_Q = \{s_0^Q, s_1^Q, s_2^Q, s_3^Q\} \\ & \mathcal{A}_P = \mathcal{A}_Q = \{a/x, b/y, c/z\}\end{aligned}$$

[14] $T^{P^{-1}}$ and $T^{Q^{-1}}$ represent T^{-1} relation of processes P and Q respectively
[15] This algorithm has been proved to decide bisimulation in [Fer90].
[16] This example is a variant of an example from [Fer90].

$T^P_{a/x} = \{(s^P_0, s^P_1), (s^P_1, s^P_2), (s^P_2, s^P_1)\}$
$T^P_{b/y} = \{(s^P_0, s^P_3), (s^P_1, s^P_3), (s^P_1, s^P_4), (s^P_2, s^P_4)\}$
$T^P_{c/z} = \{(s^P_3, s^P_5), (s^P_4, s^P_5)\}$
$T^Q_{a/x} = \{(s^Q_0, s^Q_1), (s^Q_1, s^Q_1)\}$
$T^Q_{b/y} = \{(s^Q_0, s^Q_2), (s^Q_1, s^Q_2)\}$
$T^Q_{c/z} = \{(s^Q_2, s^Q_3)\}$

A simultaneous refinement of the trivial partitions of Sts_P and Sts_Q ($B^P_0 = \{s^P_0, s^P_1, s^P_2, s^P_3, s^P_4, s^P_5\}$ and $B^Q_0 = \{s^Q_0, s^Q_1, s^Q_2, s^Q_3\}$), where $\rho = \{(B^P_0, B^Q_0)\}$), is given by $\Phi(\rho)$. Such a refinement is shown in Figure 4.19 and described as follows:

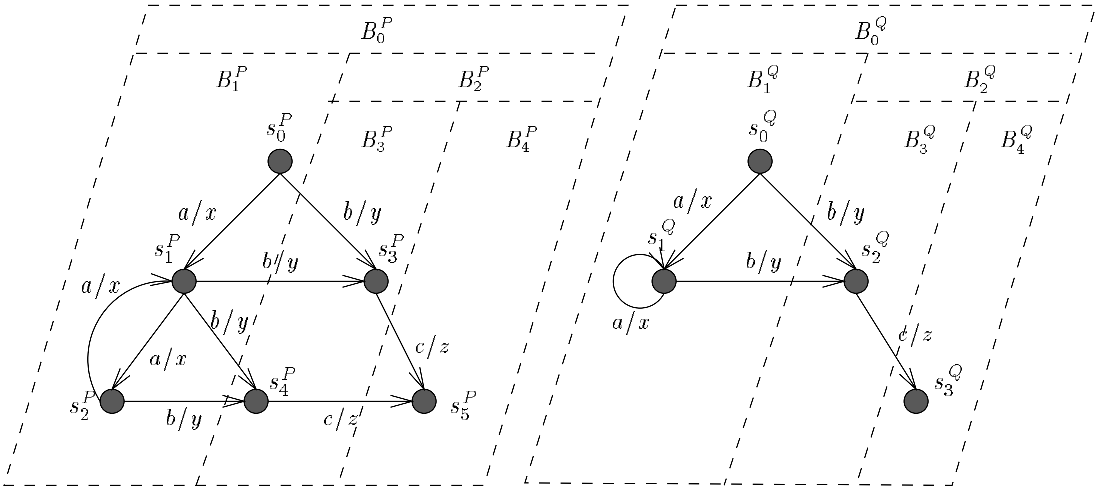

Figure 4.19: Coarsest Partitions of Sts_P and Sts_Q

1. Refinement with respect to (B^P_0, B^Q_0):

   - action a/x: the refinement of ρ in relation to $T^P_{a/x}$ and $T^Q_{a/x}$ splits B^P_0 into $B^P_1 = \{s^P_0, s^P_1, s^P_2\}$ and $B^P_2 = \{s^P_3, s^P_4, s^P_5\}$, and B^Q_0 into $B^Q_1 = \{s^Q_0, s^Q_1\}$ and $B^Q_2 = \{s^Q_2, s^Q_3\}$. The refined ρ is now $\rho' = \{(B^P_1, B^Q_1), (B^P_2, B^Q_2)\}$.
   - action b/y: since successive refinements must be performed, the partition considered at this stage is ρ' instead of ρ. The refinement of ρ' with respect to $T^P_{b/y}$ and $T^Q_{b/y}$ ends up with ρ' because $T^{P^{-1}}_{b/y} = B^P_1$ and $T^{Q^{-1}}_{b/y} = B^Q_1$.
   - action c/z: the refinement of ρ' with respect to c/z splits B^P_2 into $B^P_3 = \{s^P_3, s^P_4\}$ and $B^P_4 = \{s^P_5\}$, and B^Q_2 into $B^Q_3 = \{s^Q_2\}$ and $B^Q_4 = \{s^Q_3\}$. As a result, a new partition is created: $\rho'' = \{(B^P_1, B^Q_1), (B^P_3, B^Q_3), (B^P_4, B^Q_4)\}$.

2. Since partition ρ'' differs from ρ, ρ is said an unstable partition and stability of ρ'' must now be checked. The refinement of ρ'' with respect to the pairs of blocks $(B^P_1, B^Q_1), (B^P_3, B^Q_3)$ and (B^P_4, B^Q_4) is as follows:

   - action a/x: ρ'' is already stable with respect to a/x and then ρ'' remains as its own refinement.
   - action b/y: for action b/y, ρ'' is also an stable partition.

- action c/z: finally, ρ'' remains as the coarsest refinement of itself because c/z does not split ρ'' into further sub–blocks.

As ρ'' remains as a stable partition with respect to all actions executed by P and Q, it is the coarsest partition of both processes states. Moreover, these processes are bisimilar because for each state block of Sts_P exists an equivalent element in Sts_Q, and vice–versa.

4.6.2 Simulation

Despite the existence of algorithms for deciding bisimulation, we are particularly interested in checking a particular simulation between processes. Suppose P is the process to be simulated, while Q is the process to be reused. A process can simulate another if for all the initial actions of the desired process, the simulation process behaves bisimilarly. Furthermore, all the following corresponding processes are similar.

The transition systems for both processes are as in the previous section, $TS(P)$ and $TS(Q)$[17], $\mathsf{IncidAct}_i(B^P)$ gives all input actions coming to processes in block B^P ($\mathsf{IncidAct}_i(B^P) = \{i \mid (P', P'') \in T^P_{i/o}$ and $P'' \in B^P\}$), and T^{P^*} gives all sequences of transitions performed by P.

The properties necessary and sufficient to relate processes by simulation are given as follows[18]:

1. $\mathcal{A}_P \subseteq \mathcal{A}_Q$

2. $$reachable(B^Q, B^P, \mathcal{A}_P) \triangleq \forall s_i^Q \in B^Q \cdot \exists s_j^P \in B^P, [i_1/o_1, ..., i_n/o_n] \in \mathcal{A}_P^* \cdot$$
$$(s_0^Q \xrightarrow{i_1/o_1} \cdots \xrightarrow{i_n/o_n} s_i^Q) \in T^{Q^*}$$
$$\Rightarrow (s_0^P \xrightarrow{i_1/o_1} \cdots \xrightarrow{i_n/o_n} s_j^P) \in T^{P^*}$$

3. $$\pi_{\alpha,(B^P,B^Q)}(\rho_P, \rho_Q) \triangleq \forall X \in \rho_P, Y \in \rho_Q \cdot$$
$$(X \cap T_\alpha^{P^{-1}}[B^P] = \emptyset \wedge Y \cap T_\alpha^{Q^{-1}}[B^Q] = \emptyset) \vee$$
$$(X \subseteq T_\alpha^{P^{-1}}[B^P] \wedge Y \subseteq T_\alpha^{Q^{-1}}[B^Q])$$

4. $$\pi_{(B^P,B^Q)}(\rho_P, \rho_Q) \triangleq \forall i/o \in Act \cdot$$
$$i \in \mathsf{IncidAct}_i(B^P) \wedge \pi_{i/o,(B^P,B^Q)}(\rho_P, \rho_Q) \wedge$$
$$reachable(B^Q, B^P, \mathcal{A}_P)$$

5. $\pi(\rho_P, \rho_Q) \triangleq \forall B^P \in \rho_P \cdot \exists B^Q \in \rho_Q \cdot \pi_{(B^P,B^Q)}(\rho_P, \rho_Q)$

Firstly, we state that the set of actions of process P must be contained in the Q's set of actions, otherwise P would never be simulated by Q. Property $\pi_{\alpha,(B^P,B^Q)}(\rho_P, \rho_Q)$ defines compatibility of B^P and B^Q with respect to $T_\alpha^{P^{-1}}$ and $T_\alpha^{Q^{-1}}$ respectively. Notice that all states in B^P and B^Q must be respectively compatible with $T_\alpha^{P^{-1}}$ and $T_\alpha^{Q^{-1}}$ at the same time. $\pi_{(B^P,B^Q)}(\rho_P, \rho_Q)$ must hold for all actions coming to states in B^P – actions that reach B^Q but do not simulate actions reaching B^P are all ignored. In this sense, all actions that reach states in B^P ($\mathsf{IncidAct}(B^P)$) behave bisimilarly to those actions that reach states in B^Q. Block B^Q is a simulation for B^P if whenever an input action can be performed by P to reach a P state in B^P, an identical action can be performed by Q to reach a state in B^Q. For such a property to hold, each action defined over the input sort $\mathsf{IncidAct}_i(B^P)$ has an action coming to B^Q which behaves bisimilarly to that one coming to B^P.

Simulation of P by Q is finally checked by property $\pi(\rho_P, \rho_Q)$. Each partition block of ρ_P must be simulated by a partition block in ρ_Q. Since Q is more defined than P, certain Q actions might be unused to simulate P. Some of the Sts_Q might therefore be unreachable if only P sequences of actions

[17] $TS(P) = (\mathsf{Sts}_P, \mathcal{A}_P, T^P, s_0^P)$ and $TS(Q) = (\mathsf{Sts}_Q, \mathcal{A}_Q, T^Q, s_0^Q)$

[18] The properties given here are based on those given by Fernadez [FM91] for bisimulation. Similar arguments can be used to prove they satisfy "simulation".

are permitted; there is, obviously, no blocks in ρ_P that are simulated by these blocks of "unreachable states".

The algorithm to refine partitions of states of similar processes is directly derived from properties defined above:

- unreachable states of Sts_Q, using the sequence of actions of P, are discarded in:

$$reachable(B, B^P, \mathcal{A}_P) = \{B' \mid B' \subseteq B \wedge s^Q \in B' \wedge s_0^Q \overset{w}{\Longrightarrow} s^Q \wedge s_0^P \overset{w}{\Longrightarrow} s^P\}$$

- property $\pi_{\alpha,(B^P,B^Q)}(\rho_P, \rho_Q)$ is constructively defined as:

$$\begin{aligned}\Upsilon_{\alpha,(B^P,B^Q)}(\rho) = \{&(X \cap T_\alpha^{P^{-1}}[B^P], Y \cap T_\alpha^{Q^{-1}}[B^Q]) \mid (X, Y) \in \rho\} \cup \\ \{&(X - T_\alpha^{P^{-1}}[B^P], Y - T_\alpha^{Q^{-1}}[B^Q]) \mid (X, Y) \in \rho\}\end{aligned}$$

- considering that actions of both processes range over $\{i_1/o_1, ..., i_n/o_n\} \subseteq (\mathcal{A}_P \cup \mathcal{A}_Q)$, such that $i_j \in \mathsf{IncidAct}_i(B^P)$ and $B^{Q'} = reachable(B^Q, B^P, \mathcal{A}_P)$, $\pi_{(B^P,B^Q)}(\rho_P, \rho_Q)$ is algorithmically defined by:

$$\Upsilon_{(B^P,B^Q)}(\rho) = (\Upsilon_{i_1/o_1,(B^P,B^{Q'})} \circ \cdots \circ \Upsilon_{i_n/o_n,(B^P,B^{Q'})})(\rho)$$

- $\pi(\rho_P, \rho_Q)$ is constructively defined by finding a stable partition of both processes from their trivial partitions. $\Upsilon(\rho)$ refines partition ρ with respect to pairs of equivalence blocks of both processes. $\Upsilon(\rho)$, however, finds (simultaneously) the stable partitions of both processes with respect to actions for which P can move. ρ is firstly defined as the trivial partition of both transition systems: $\rho = \{(\mathsf{Sts}_P, \mathsf{Sts}_Q)\}$ and then it is successively refined until the stable partition of both processes is found. Q is a simulation of P if there is no empty block in the stable partition of both processes.

$$\begin{aligned}\Upsilon(\rho) = \ &\mathsf{if}\ Stable(\Upsilon_\rho(\rho)) \\ &\mathsf{then}\ \Upsilon_\rho(\rho) \\ &\mathsf{else}\ \Upsilon(\Upsilon_\rho(\rho))\end{aligned}$$

$$\Upsilon_\rho(\rho) = \Upsilon_{(B_1^P,B_1^Q)} \circ \cdots \circ \Upsilon_{(B_n^P,B_n^Q)}(\rho)$$

Function *reachable* eliminates unreachable states of $TS(Q)$ under P sequences of actions. $\Upsilon_{\alpha,(B^P,B^Q)}(\rho)$ gives a constructive form of property $\pi_{\alpha,(B^P,B^Q)}(\rho_P, \rho_Q)$ as well as $\pi_{(B^P,B^Q)}(\rho)$ and $\Upsilon(\rho)$ are respectively constructive forms for properties $\pi_{(B^P,B^Q)}(\rho_P, \rho_Q)$ and $\pi(\rho_P, \rho_Q)$.

Partitions ρ_P and ρ_Q of πs properties are represented by pairs of partitions blocks in Υ's functions. Such an adaptation aims at representing the conjunction operator found in $\pi_{\alpha,(B^P,B^Q)}(\rho_P, \rho_Q)$ and the existential quantifier of $\pi(\rho_P, \rho_Q)$ in a more effective and concise way. For each block in ρ_P, there exists a simulation block in Q; this fact guarantees the existence of pairs of simulation blocks. Moreover, if the simulation blocks are given as pairs, all comparison and operations are simultaneously performed in both elements. At the end of this procedure, both processes are similar if there is no pair of blocks with an empty element.

Note the similarity between this algorithm and the algorithm for deciding bisimulation. In fact, the simulation relation defined in the present work looks for bisimilarity of the simulated process. So, the algorithm for deciding simulation checks bisimilarity between P and Q restricted to actions performed by P. This algorithm is concerned with Proposition 4.5 which shows bisimilarity between the simulated process and a restricted simulation one.

Partitions of both processes states are initially given as trivial partitions containing all states. The algorithm splits such partitions into classes for simulation of the specification process; this is a recursive procedure which stops when the coarsest stable partition of the specification process is found. Alternatively, the coarsest partition of both processes could be initially given, and then check their similarity. However, this procedure is far more expensive than the method presented above because it might find the corresponding simulation classes of both processes instead of constructing them "on–the–fly".

The similarity between this algorithm and that one for bisimulation is now illustrated using the process P previously defined and a new process R. Their transition systems are defined as follows:

$$
\begin{aligned}
&TS(P) = (\mathsf{Sts}_P, \mathcal{A}_P, T^P, s_0^P)\\
&TS(R) = (\mathsf{Sts}_R, \mathcal{A}_R, T^R, s_0^R)\\
&\mathsf{Sts}_P = \{s_0^P, s_1^P, s_2^P, s_3^P, s_4^P, s_5^P\}\\
&\mathsf{Sts}_R = \{s_0^R, s_1^R, s_2^R, s_3^R, s_4^R\}\\
&\mathcal{A}_P = \mathcal{A}_R = \{a/x, b/y, c/z\}\\
&T^P_{a/x} = \{(s_0^P, s_1^P), (s_1^P, s_2^P), (s_2^P, s_1^P)\}\\
&T^P_{b/y} = \{(s_0^P, s_3^P), (s_1^P, s_3^P), (s_1^P, s_4^P), (s_2^P, s_4^P)\}\\
&T^P_{c/z} = \{(s_3^P, s_5^P), (s_4^P, s_5^P)\}\\
&T^R_{a/x} = \{(s_0^R, s_1^R), (s_1^R, s_1^R)\}\\
&T^R_{b/y} = \{(s_0^R, s_2^R), (s_1^R, s_2^R)\}\\
&T^R_{c/z} = \{(s_2^R, s_3^R), (s_1^R, s_4^R)\}
\end{aligned}
$$

The algorithm $\Upsilon(\rho)$ is applied to the trivial partitions of Sts_P and Sts_R ($B_0^P = \{s_0^P, s_1^P, s_2^P, s_3^P, s_4^P, s_5^P\}$ and $B_0^R = \{s_0^R, s_1^R, s_2^R, s_3^R, s_4^R\}$), where $\rho = \{(B_0^P, B_0^R)\}$), to find the coarsest partitions of both processes that simulate P. Such a refinement is shown in Figure 4.20 and described as follows:

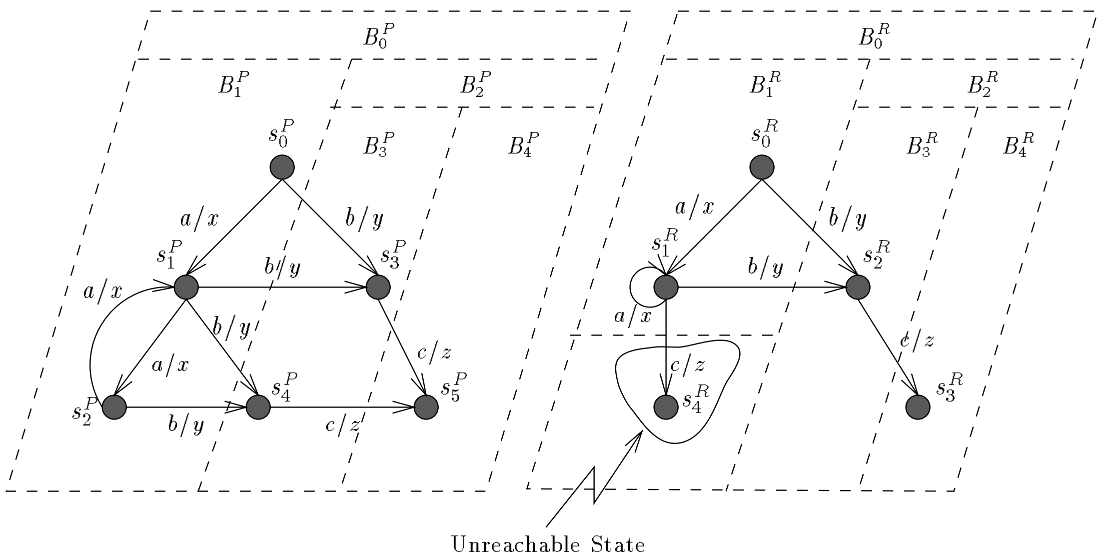

Figure 4.20: Coarsest Partitions of Sts_P and Sts_R

1. Refinement with respect to (B_0^P, B_0^R):

   - action a/x: the refinement of ρ in relation to $T^P_{a/x}$ splits B_0^P into $B_1^P = \{s_0^P, s_1^P, s_2^P\}$ and $B_2^P = \{s_3^P, s_4^P, s_5^P\}$. Since the sequence of actions $[a/x, c/z]$ is not performed by P, s_4^R is an unreachable state and the block of reachable states of Q is given by $B_0^{R'} = \{s_0^R, s_1^R, s_2^R, s_3^R\}$. The refinement of $B_0^{R'}$ in relation to $T^R_{a/x}$ splits it into $B_1^R = \{s_0^R, s_1^R\}$ and $B_2^R = \{s_2^R, s_3^R\}$. The refined ρ is now $\rho' = \{(B_1^P, B_1^R), (B_2^P, B_2^R)\}$.
   - action b/y: the refinement of ρ' with respect to $T^P_{b/y}$ and $T^R_{b/y}$ ends up with ρ'.

- action c/z: the refinement of ρ' with respect to c/z splits B_2^P into $B_3^P = \{s_3^P, s_4^P\}$ and $B_4^P = \{s_5^P\}$, and B_2^R into $B_3^R = \{s_2^R\}$ and $B_4^R = \{s_3^R\}$. A new partition is then created: $\rho'' = \{(B_1^P, B_1^R), (B_3^P, B_3^R), (B_4^P, B_4^R)\}$.

2. Since partition ρ'' differs from ρ, the stability of ρ'' must now be checked. In fact, ρ'' is stable and given as result.

Since no empty blocks are found on the coarsest partitions of those processes, R is a simulation of P and s_4^R is unreachable when R is used to simulate P.

4.6.3 Channel Restricted Simulation

The algorithm for checking channel restricted simulation is derived from the simulation algorithm. As it has already been stated in Theorem 4.6 (page 68), certain constraints can be applied to channel restricted similar processes in order to obtain similar ones. If both processes are restricted to the set of their coincident channels, the simulation algorithm can then be applied to decide whether those restricted processes are similar. If they are so, then the unrestricted processes are channel restricted similar.

Suppose C is the intersection of P and Q channels. The transition system for P restricted to C ($P \upharpoonright_c C$) is defined as $TS(P') = (\mathsf{Sts}'_P, \mathcal{A}'_P, T^{P'}, s_0^P)$, and $Q \upharpoonright_c C$ is represented by $TS(Q') = (\mathsf{Sts}'_Q, \mathcal{A}'_Q, T^{Q'}, s_0^Q)$. The algorithm for checking channel restricted similarity is as follows:

$$\Upsilon^C(\{(\mathsf{Sts}_P, \mathsf{Sts}_Q)\}) = \Upsilon(\{(\mathsf{Sts}'_P, \mathsf{Sts}'_Q)\})$$

4.6.4 Simulation under a Functional Interface

Processes that are similar under a functional interface can be restricted to achieve simulation – Theorem 4.5, Section 4.5.2:

$$\exists f : \mathsf{Sort}_i(P) \times \mathsf{Sort}(Q) \longrightarrow \mathsf{Sort}_o(P) \cdot P \sqsubseteq ((Q \mid P_f) \upharpoonright_c \mathsf{Chs}(P)).$$

The algorithm for deciding simulation under a functional interface is then based on the simulation algorithm in such a way that composition of Q with the process which implements function f is tested as a simulation for P (whenever a function f is given, we assume the existence of the corresponding process P_f without showing its construction – it has been defined in Definition 4.3, page 54):

$$\begin{aligned}
&\Upsilon^f(\{(\mathsf{Sts}_P, \mathsf{Sts}_Q)\}) = \\
&\qquad \{(P_f, (\Upsilon(\{(\mathsf{Sts}_P, \mathsf{Sts}'_Q)\}))) \mid i_p/o_p \in \mathcal{A}_P \wedge i_q/o_q \in \mathcal{A}_Q \wedge \\
&\qquad\qquad ((i_p, i_q/o_q) \mapsto o_p) \in f \wedge \\
&\qquad\qquad TS((Q \mid P_f) \upharpoonright_c \mathsf{Chs}(P)) = (\mathsf{Sts}'_Q, \mathcal{A}'_Q, T^{Q'}, s_0^Q) \wedge \\
&\qquad\qquad \mathit{non\text{–}empty}(\Upsilon(\{(\mathsf{Sts}_P, \mathsf{Sts}'_Q)\}))\}
\end{aligned}$$

We deliberately use EPA combinators inside this algorithm without implementing them. Algorithmic definition of combinators of process algebras are trivially defined as functions. The element used in this algorithm without being previously defined is the function *non–empty*. It checks non–emptiness of pairs of sets; every block of Sts_P must have a corresponding non–empty block of Sts_Q and vice–versa.

P_f implements a function (f). So, the number of P_f processes is as many as functions can be found over sorts of both processes. The complexity of this algorithm is that of testing simulation between P and all composite Q with P_f processes such that f is an interface function over P and Q sorts. The result of this algorithm is both a family of single state processes that can be composed with Q to behave similarly to P, and the coarsest partitions of both processes under simulation.

Note that several functions can actually be found to interface P and Q sorts. More efficient algorithms might find these functions "on–the–fly", but it still requires a huge computation; finding various functions and checking simulation between both processes under such an interface.

4.6.5 Channel Restricted Simulation under a Functional Interface

This relation can also be checked by using an algorithm based on the simulation one, since certain constraints applied to channel restricted similar processes under a functional interface make them similar (as stated in Theorem 4.7, page 69). In fact, the algorithm presented for checking such a relation is based on that one for checking simulation under a functional interface. It is a straightforward result from Proposition 4.4, page 69.

$$\begin{aligned}\Upsilon^{fC}(\{(\mathsf{Sts}_P, \mathsf{Sts}_Q)\}) = \{(C, (\Upsilon^f(\{(\mathsf{Sts}'_P, \mathsf{Sts}'_Q)\}))) \mid\ & C \subseteq (\mathsf{Chs}(P) \cup \mathsf{Chs}(Q)) \wedge \\ & TS(P \restriction_c C) = (\mathsf{Sts}'_P, \mathcal{A}'_P, T^{P'}, s_0^P) \wedge \\ & TS(Q \restriction_c C) = (\mathsf{Sts}'_Q, \mathcal{A}'_Q, T^{Q'}, s_0^Q) \wedge \\ & \mathit{non\text{-}empty}(\Upsilon^f(\{(\mathsf{Sts}'_P, \mathsf{Sts}'_Q)\}))\}\end{aligned}$$

In principle, this algorithm differs from the previous one (Section 4.6.4) in definition of P_f. Now, P_f is defined over channel restricted sorts of both processes instead of using the full set of channels. Also, the comparison is performed on channel restricted processes since only a channel restricted P is similar to Qunder a functional interface.

This algorithm gives as result all possible sets of channels C and the corresponding functions f under which P and Q are similar. A family of functions f can then be given with each set of channels C. The complexity of this algorithm is calculated as application of the algorithm for checking similarity under a functional interface for each subset of actions.

An improved algorithm should take only maximal sets of channels and find the P_fs "on–the–fly". But it still remains a non–trivial computational problem.

4.7 Discussion

The relations on processes presented in this chapter are concerned with: looking for behavioural similarities between a specification and a given component, considering that the existing component has its behaviour "transformed" by an interface process to yield the desired specification. Since the existing and interface processes run synchronously, similarities between the specification and the component only exists if the existing component is able to simulate all actions performed by the desired specification via composition with an interface process.

From one side, reuse underlying synchronous composition is restricted to cases where the existing process can "simulate" all actions performed by the desired specification. Processes which simulate only a subset of actions of the specification process cannot be reused under synchronous composition. From the other side, synchronous composition allows for matching processes defined over distinct sorts. In the channel restricted simulation, for example, only a subset of synchronous actions are compared because the subactions not compared are provided by the interface process. The other basic similarity relation relies on the fact that, with synchronous composition, the sort of the existing component can be transformed to the specified one by composition with a single state process.

Our account relies on four basic concepts: bisimulation, simulation, abstraction on sorts and abstraction on synchronous actions. All the relations presented in this work only relate processes where one of them is somehow able to simulate all moves of the other process. Hence, these similarity relations are useful for reuse using a synchronous interface approach.

The algorithms presented here are aimed at showing decidability of the similarity relations rather than studying efficient algorithms (they have been prototyped in SML [Pau92]). The algorithm for deciding bisimulation, by Fernandez, can be implemented in $\mathcal{O}(mn)$ time where n represents the number of states and m is the number of transitions. The algorithms for deciding simulation and channel restricted simulation can be implemented in identical time complexity, but the algorithms which involve comparison using a a functional interface have their time complexity increased by the number of functions that can interface the sorts of both processes. We believe that algorithms which find the possible functions "on–the–fly" are far more efficient, but such algorithms have not been studied so far.

Most relations shown in this chapter requires another process running in parallel with the existing one in order to accomplish the whole desired behaviour. The forthcoming chapter (Chapter 5) shows that once the similarity between processes (desired and existing processes) is found, the interface process can be constructed.

Chapter 5

A Solution for the Interface Equation

5.1 Introduction

As we have already discussed in the foregoing chapters, a solution for the interface equation involving synchronous processes, i.e. finding a process X that satisfies

$$(P \mid X) \restriction_c \mathsf{Chs}(S) \;\sim\; S, \tag{5.1}$$

depends on how the given component P can partially simulate the desired specification S. Also, the similarities between the desired and given processes defined by relations presented in Chapter 4 indicate the partial simulation of one process by the other and, therefore, the dissimilarities between them can be inferred straightforwardly. Once the similarities between processes have been recognized, various solutions can be given to the interface equation. This chapter is devoted to presenting a particular solution for the interface equation, by creating a combinator for EPA which enables the interface process to be constructed.

Due to the nature of the EPA composition operator, multiple solutions can be found for the interface equation. However, to find an algebraic combinator to solve such a problem, a generic interface process must be constructed. A discussion on the interface process necessary to solve the equation for processes related by any of the presented similarity relations will be further conducted.

Once the interface elements to solve the equation are defined for the various similarity relations, the generic interface process can be found using a "decomposition operator". This operator is aimed at playing the same role as the "division" operator for numbers. For example, consider having to find a solution for equation

$$x \star 2 = 10. \tag{5.2}$$

How can this equation be solved? One approach is to use the algebra defined for the division and multiplication operators, for $m, n, t \in \mathbb{Z}$ and $n \neq 0$:

$$m \star n = t \text{ if and only if } m = t \div n,$$

and thus find the following solution:

$$x = 10 \div 2.$$

To obtain a similar algebraic treatment for processes, the "decomposition operator" is defined, and then several laws to provide algebraic manipulation of reusable processes are finally presented.

5.2 Multiple Solutions for the Interface Equation

We are particularly interested in solving the interface equation by providing an algebraic combinator. Such a solution requires the creation of a unique interface process. We must then check the theoretical limitations of reusing components undertaking the synchronous interface approach, and whether a unique solution can be found.

5.2.1 Constraints

The interface equation requires bisimilarity between the specification and the compound process which has an existing process and an interface process as components; all actions from the specification must be matched to actions from the compound process and vice–versa. For a synchronous approach, in particular, each specification action (action from the specification component) is simulated by synchronous execution of an existing action with an interface action that are conflict free. So, for each specification action, there exists an action in the existing component that is going to simulate it. Also, such an existing action must be conflict free only with actions from the interface process whose composition is bisimilar with the required specification action.

Conflict freedom of actions can be rewritten as follows:

Definition 5.1 *Suppose action "α" is defined over the set of channels C_1, "β" is defined over C_2 and $C = C_1 \cap C_2$. Then, "α" is conflict free with "β" iff res-ch$(\alpha, C) =$ res-ch(β, C).*

To hold uniqueness on combination of actions, the component processes must hold the following property:

Proposition 5.1 *Suppose $C = \mathsf{Chs}(P) \cap \mathsf{Chs}(Q)$. Process P has its initial actions uniquely combined with actions from Q only if for all action $\alpha \in \mathsf{Sort}_1(P)$, there exists a unique $\beta \in \mathsf{Sort}_1(Q)$ such that res-ch$(\alpha, C) =$ res-ch(β, C).*

This property defines what is necessary for P to achieve uniqueness of conflict freedom of its initial actions with Q's initial actions. In order to have all P's actions uniquely combined with Q's actions, this property must hold for all the compound processes reachable from $(P \mid Q)$.

In looking for processes that hold uniqueness on composition of actions, we first address the case of composition of a process with itself $(P \mid P)$. To satisfy Proposition 5.1, each action of P must be combined only with itself, and as a result, $(P \mid P) \sim P$. In general, composition of an arbitrary process P with its copy does not ensure bisimilarity with itself. Suppose $P \xrightarrow{\alpha} P'$, $P \xrightarrow{\alpha} P''$ such that $P' \not\sim P''$. The compound process $(P \mid P)$ has the transitions: $(P \mid P) \xrightarrow{\alpha} (P' \mid P')$, $(P \mid P) \xrightarrow{\alpha} (P'' \mid P'')$ and $(P \mid P) \xrightarrow{\alpha} (P' \mid P'')$. The two first moves are bisimilar to those moves of P, but the third one reaches a process non–bisimilar to any of the P's reachable processes. Thus, $(P \mid P) \not\sim P$. To achieve uniqueness of conflict free actions for all the P's actions, this process must be deterministic for all its input/output actions ($\mathcal{D}!(P)$).

Lemma 5.1 $(P \mid P) \sim P$ *if and only if* $\mathcal{D}!(P)$.

Considering that each P input/output action is unique, it can only be combined with its own copy. So, the above property is sufficient to ensure bisimilarity of P with the compound process $(P \mid P)$. On the other hand, if $(P \mid P) \sim P$, determinism of input/output actions is ensured because uniqueness of conflict free actions holds. Thus, the above property is sufficient and necessary to ensure bisimilarity between a process and its self–composition.

For a general case, consider processes P and Q, and the set of channels $C = \mathsf{Chs}(P) \cap \mathsf{Chs}(Q)$. To obtain each action of P uniquely combined with actions from Q, each P action must be conflict free with a unique Q action (for each $\alpha \in \mathsf{Sort}_1(P \restriction_c C)$, there exists a unique action $\beta \in (Q \restriction_c C)$ such that $\alpha = \beta$, Proposition 5.1). So, if $(Q \restriction_c C)$ is not deterministically defined for its input/output actions that are conflict free with P, then uniqueness of combination of P with Q actions cannot be achieved since some of the Q's actions could be combined with more than one action from P. Determinism of processes for its set of actions that are conflict free with another process is defined as follows:

Definition 5.2 *Suppose* $C = \mathsf{Chs}(P) \cap \mathsf{Chs}(Q)$. *$Q$ is deterministically defined for its input/output actions that are conflict free with P (denoted as $\mathcal{D}!_P(Q)$), if for every "proper actions" $\alpha, \beta \in Act$, whenever $Q \xrightarrow{\alpha} Q'$ and $P \xrightarrow{\beta} P'$ such that α CF β, then $\mathcal{D}!_\alpha(Q)$ and $\mathcal{D}!_{P'}(Q')$.*

Then, the necessary condition to have P actions combined with exactly one action from Q is as follows:

Proposition 5.2 *Suppose* $C = \mathsf{Chs}(P) \cap \mathsf{Chs}(Q)$. *$P$ actions are uniquely combined with Q actions only if $\mathcal{D}!_P(Q \upharpoonright_c C)$.*

As a consequence, a solution for the interface equation can only be found if the property defined in Proposition 5.2 holds for the existing and the interface processes.

5.2.2 Multiple Solutions

The fact that the EPA composition operator hides the input actions that coincide with output actions (between the component processes), there might exist various solutions for the interface equation.

Suppose, for example, processes $P(c_1, c_2/c_3)$, $Q(c_1, c_4/c_5)$ and $R(c_3, c_4/c_5)$ perform the following transitions[1]:

$$
\begin{aligned}
P &\xrightarrow{a/x} P' \\
Q &\xrightarrow{b/y} Q' \\
R &\xrightarrow{c/z} R'
\end{aligned}
$$

such that

$$
\begin{aligned}
a/x &= (c_1(0) \oplus c_2(1))/(c_3(0)) \\
b/y &= (c_1(0) \oplus c_4(0))/(c_5(0)) \\
c/z &= (c_3(0) \oplus c_4(0))/(c_5(0)).
\end{aligned}
$$

Processes $(P \mid Q)$ and $(P \mid R)$ perform the following transitions:

$$
\begin{aligned}
(P \mid Q) &\xrightarrow{a/x \star b/y} (P' \mid Q') \\
(P \mid R) &\xrightarrow{a/x \star c/z} (P' \mid R')
\end{aligned}
$$

where $(a/x \star b/y)$ and $(a/x \star c/z)$ are "proper" actions because a/x is conflict free with b/y and c/z. These compound actions are identical because the input actions that coincide with the output's are hidden in composition of actions $(i_m/o_m \star i_n/o_n = ((i_n \oplus i_m) \ominus (o_n \oplus o_m))/(o_n \oplus o_m))$:

$$
\begin{aligned}
(a/x \star b/y) &= (c_1(0) \oplus c_2(1) \oplus c_4(0))/(c_3(0) \oplus c_5(0)) \\
(a/x \star c/z) &= (c_1(0) \oplus c_2(1) \oplus c_4(0))/(c_3(0) \oplus c_5(0)).
\end{aligned}
$$

Therefore, composition of distinct actions can result in identical compound actions. Scaling cases of distinct single actions generating identical compound actions up to processes, it can be checked that composition of distinct processes could result in identical behaviour. For instance, if P' is bisimilar to P, $Q \sim Q'$, $R \sim R'$, and these processes are only defined for the range of actions presented above, then $(P \mid Q)$ is bisimilar to $(P \mid R)$. This means that if $S \trianglelefteq (P \mid Q)$, then the interface equation

$$(P \mid X) \upharpoonright_c \mathsf{Chs}(S) \sim S$$

[1] We omit the channel names of processes in the forthcoming definitions to simplify notation. The processes are assumed to have channels as previously defined.

where X is the incognito process, has Q or R as possible solutions.

The fact that identical input channels are shared by both processes, and also certain input channels are hidden when they coincide with output channels (internal connections), makes composition an odd combinator (with no complementary combinator). Although various solutions can be found for the interface equation, we are particularly interested in all minimal solutions. Moreover, to allow a combinator to be created to solve the interface equation, an interface process comprising all the minimal solutions must be provided.

5.3 Similarity Relations vs. Interface Processes

Apart from bisimulation, all the similarity relations presented here require an interface process to yield bisimilarity with the specification. In this section, the information necessary to be included in the generic interface process will be highlighted for cases where the specification and the existing process are related by each of the presented similarity relations.

To facilitate further discussions, we assume S as the desired specification and P as the given component. X is the interface process to be found.

5.3.1 Simulation

Processes related by simulation cannot achieve bisimilarity under application of combinators. Composition of a process S with P, which is a simulation of S, can however achieve bisimilarity with S if it is deterministic on its input/output actions that simulate S ($\mathcal{D}!_S(P)$).

Proposition 5.3 *If $S \sqsubseteq P$ and $\mathcal{D}!_S(P)$, then $(S \mid P) \sim S$.*

This a straightforward result from Lemma 5.1, Proposition 4.5, and the fact that only conflict free actions are synchronously executed to provide compound actions. Whether $S \sqsubseteq P$, it can be inferred that S and P are defined over identical sets of channels, and then actions of S and P are conflict free only if they are identical. Because of that, the compound process $(S \mid P)$ behaves bisimilarly to S.

Therefore, to solve the interface equation when the given component is a simulation of the desired specification, a process bisimilar to the specification must be provided as an interface process. Under such circumstances, there is no improvement in reusing processes that are simulations of the specification because the specification process is necessary to be constructed as interface (except when it can be minimized as an interacting element – Chapter 6). For effective (and predictable) reuse of processes that are higher ordered than specifications (from a definedness point of view), a new ordering relation stronger than simulation must be defined: *strong simulation.*

Strong Simulation

A simulation process is able to move undertaking actions that do not simulate any action from the simulated process, the "spare actions". Some spare actions, however, could eventually be an element in the set of actions of the simulated process. This is basically the reason which makes similar processes impossible to be constrained to bisimilar ones under application of combinators. A new preorder relation can then be defined under which the simulation process performs all actions present in the simulated one, or even more, but whenever "spare actions" exist, they must be ones never performed by the simulated process. This *Strong Simulation* is defined as follows:

Definition 5.3 *Strong Simulation $\mathcal{S}_s$ is a binary relation on processes such that if $(P, Q) \in \mathcal{S}_s$ (abbreviated by $P \sqsubseteq_s Q$), then for every "proper" action $i_1/o_1 \in Act$ such that $i_1 \in \mathsf{Sort}_1(P)$ (initial actions of P):*

1. $\mathsf{Sort}_1(Q) \cap \mathsf{Sort}(P) = \mathsf{Sort}_1(P)$,
2. *whenever $S \xrightarrow{i_1/o_1} S'$ then $P \xrightarrow{i_1/o_1} P'$ for some P' such that $S' \sqsubseteq_s P'$,*
3. *whenever $P \xrightarrow{i_1/o_1} P'$ then $S \xrightarrow{i_1/o_1} S'$ for some S' such that $S' \sqsubseteq_s P'$.*

This definition is very similar to that for simulation. It is however stronger than simulation since only actions not defined in the simulated process can be given as "spare actions" of the simulation process. This restriction is fixed by the first constraint above.

Considering such a constraint, if the "spare actions" are removed from the simulation process, then this restricted simulation process is bisimilar to the simulated one:

Theorem 5.1 *If* $P \sqsubseteq_s Q$*, then* $P \sim (Q \upharpoonright_a \mathsf{Sort}(P))$.

Proof. Whenever Q performs an action that is defined in the sort of actions of P, P is able to perform the same action. For each action performed by P, Q is able to perform an identical action. So, if Q initial actions are restricted to actions in P's sort, all initial actions of this Q restricted process are also performed by P (the first constraint in Definition 5.3), and then they reach strongly similar processes. The restriction to actions combinator restricts initial actions, as well as all the other actions, to the set of actions actually defined with the combinator. The restricted Q performs initial actions bisimilarly to P and then reaches processes with this property and so on. This restricted Q is indeed bisimilar to P. □

Strong simulation is restrictive enough to guarantee that a simple restriction to the sort of simulated actions can turn these processes bisimilar. But, how could the restriction to actions combinator be interpreted in terms of the interface equation?

Regarding the interface approach, processes must be reused without redesigning. So, the Q restricted process must be achieved by composition with an interface process. Considering synchronism of composition operator, restriction to actions can be implemented by composition of the process with a single state process that only performs actions embedded in the restricted set. A single state process that only performs actions in a given set is defined as follows:

Definition 5.4 *A single state process defined over sort* $\mathcal{A}_1$*,* $P_{\mathcal{A}_1}$*, is defined by:*

1. *for all action* α *defined in* $\mathcal{A}_1$*,* $P_{\mathcal{A}_1} \xrightarrow{\alpha} P_{\mathcal{A}_1}$*, and*
2. $\mathsf{Sort}(P_{\mathcal{A}_1}) = \mathcal{A}_1$.

Restriction to actions and this single state process are related as follows:

Proposition 5.4 $(P \upharpoonright_a \mathcal{A}_1) \sim (P \mid P_{\mathcal{A}_1})$.

With this proposition and the fact that bisimulation is a congruence, the following corollary emerges:

Corollary 5.1 *If* $P \sqsubseteq_s Q$*, then* $P \sim (Q \mid P_{\mathcal{A}_P})$.

The interface for processes related by strong simulation are illustrated by the case of reusing the traffic light controller to implement the other one embedded in a perfect environment (already presented in Section 3.2). The traffic light process is a strong simulation of the one in the perfect environment because beyond performing all actions of the perfect process, it is also able to respond to actions $(\overline{car}, \overline{t_s}/G, R, st)$ and $(\overline{car}, t_s/G, R, st)$ from subprocess $Amb_2(s\colon sensor, t\colon intervs/l_{mj}, l_{mn}\colon colour, setc\colon stcount)$ (the transitional graphs for both processes are found in Appendix D). Also, these "spare actions" are not element in the sort of actions of the major/minor roads process in the perfect environment. As a result, the single state process shown in Figure 5.1 represents a solution for the interface equation

$$(TLC_2(s, t/l_{mj}, l_{mn}, setc) \mid X) \upharpoonright_c \mathsf{Chs}(TLC_{perf}) \sim TLC_{perf}(s, t/l_{mj}, l_{mn}, setc).$$

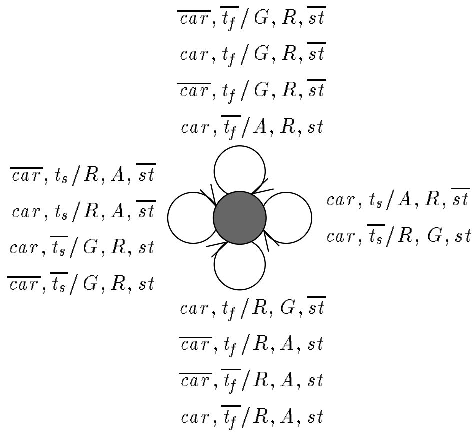

Figure 5.1: An Interface Process for Implementing TLC_{perf} using TLC_2

5.3.2 "Functional Interface" Relations

If process P is similar to S under a total functional interface, then only a single state process is required as the interface process. This property has been stated in Theorem 4.2.

The procedure to compare similar processes under a total functional interface ends up with a map (or maps) over io–sorts of both processes. So, single state processes can then be created from any of the obtained functions, as defined in Section 4.2.2 (Definition 4.3), and they are indeed interface processes.

For processes that are similar under a functional interface, however, composition of the simulation process with a single state process ends up with a similar (Theorem 4.5) instead of a bisimilar process. This particular relation faces the same problem found in simulation, and the desired process is required as solution for the interface equation.

To amend such a problem, we have again to define a new relation which is neither too strong, as simulation under a total functional interface, nor too weak, as simulation under a functional interface, but which can be reduced to bisimulation under application of combinators. This new relation is based on strong simulation instead of simulation:

Definition 5.5 *Strong Simulation under a Functional Interface $\mathcal{S}_{sf}$, for $f{:}\,\mathsf{Sort}_i(P) \times \mathsf{Sort}(Q) \nrightarrow \mathsf{Sort}_o(P)$, is a binary relation on processes such that if $(P, Q) \in \mathcal{S}_{sf}$ (abbreviated by $P \overset{f}{\sqsubseteq}_s Q$), then for every "proper" action i_q/o_q, $i_p/o_p \in Act$ and $i'_p \in \mathsf{Sort}_i(P)$:*

1. *whenever $f(i_p, i_q/o_q) = o_p$ and $i_q/o_q \in \mathsf{Sort}_1(Q)$, then $i_p/o_p \in \mathsf{Sort}_1(P)$,*
2. *whenever $P \xrightarrow{i_p/o_p} P'$ then $Q \xrightarrow{i'_q/o'_q} Q'$ for some i'_q/o'_q and Q' such that $f(i_p, i'_q/o'_q) = o_p$ and $P' \overset{f}{\sqsubseteq}_s Q'$,*
3. *whenever $Q \xrightarrow{i_q/o_q} Q'$ and $(i'_p, i_q/o_q) \in \mathsf{dom}\, f$ then $P \xrightarrow{i'_p/o'_p} P'$ for some o'_p and P' such that $f(i'_p, i_q/o_q) = o'_p$ and $P' \overset{f}{\sqsubseteq}_s Q'$.*

This definition is quite similar to that one for simulation under a functional interface (the first property is implicitly defined in the third one; we only maintain it in the definition to make analogy with the strong simulation). Whenever Q actions are related to any of the P actions by function f, there actually exists a P action of which the Q's action is a simulation. This means that the initial "spare actions" performed by Q cannot be a simulation for any action in P's sort. There exists no Q initial action related (by f) to any other action in the P's sort apart from the P initial actions.

Since all Q actions defined in f are a simulation of P actions, composition of Q with a process which implements f is able to perform only actions that are actually a simulation for P actions (due to synchronism of the EPA composition operator). As a consequence, such a compound process is bisimilar to P. This property is stated by the following theorem:

Theorem 5.2 *$P \overset{f}{\sqsubseteq}_s Q$ iff exists a function f such that*
$f\colon \mathsf{Sort}_i(P) \times \mathsf{Sort}(Q) \rightarrow \mathsf{Sort}_o(P)$ and
$P \sim (Q \mid P_f) \restriction_c \mathsf{Chs}(P)$

Proof. By definition of $\mathcal{S}_{sf}$, for each action i_p/o_p performed by P, there exists a corresponding action i_q/o_q from Q such that $f(i_p, i_q/o_q) = o_p$ and the corresponding action in P_f is given by $P_f \xrightarrow{(i_p \oplus o_q) \ominus o_p/(i_q \oplus o_p) \ominus i_p} P_f$.

Since P_f actions depend on P as much as Q actions, each Q's action can only be combined with actions of P_f to perform a compound action identical to the P's one (as shown in the proof of Theorem 4.2). Considering that the set of initial actions of this compound process is identical to the sort of P's initial actions, the Q's "spare actions" are not executed when Q is composed with P_f; these actions cannot be combined with any P_f's action. By scaling this equality of sets of initial actions up to all processes reachable from P and the compound process, bisimilarity between P and the compound process is concluded.

The backward implication is also trivial.

□

From the theorem above, a single state process is sufficient to solve the interface equation whether the desired and given components are related by strong simulation under a functional interface. In fact, the number of minimal solutions to the interface equation is as many as functions that can be found to interface the specification with the given process.

Let us use the example of the major/minor roads process being reused to implement the pedestrian traffic light controller, to illustrate the relation above. We have shown, in Section 4.3, that the major/minor roads process is similar under a functional interface to the pedestrian/road process. But, since all the "spare actions" of major/minor roads process are elements not defined in the domain of $f_{TLC_{ped}\text{-}TLC_2}$ (vide their transition graphs in Appendix D), these processes are indeed strongly similar under a functional interface. A single state process which implements $f_{TLC_{ped}\text{-}TLC_2}$, $P_{f_{TLC_{ped}\text{-}TLC_2}}(b, t, l_{mj}, l_{mn}/s, l_r, l_{pd}, setc)$ (Figure 5.2), represents therefore a solution for the equation:

$$(TLC_2(b, t/l_r, l_{pd}, setc) \mid X) \restriction_c \mathsf{Chs}(TLC_{ped}) \sim TLC_{ped}(s, t/l_{mj}, l_{mn}, setc).$$

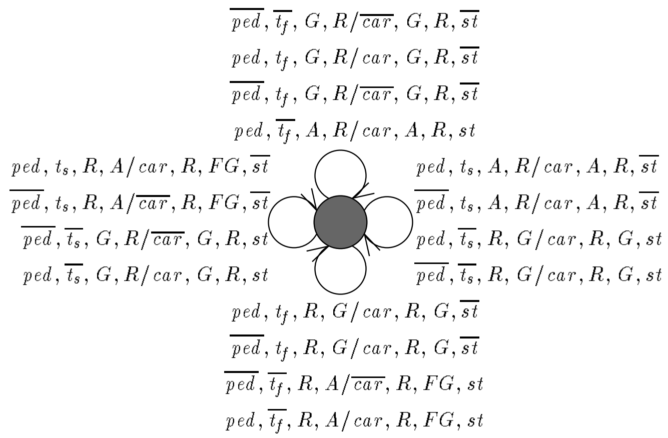

Figure 5.2: An Interface Process for Implementing TLC_{ped} using TLC_2

5.3.3 "Channel Restricted" Relations

Channel restricted bisimulation and simulation abstract on a set of channels to compare processes. As a result, channel restricted bisimilar/similar processes coincide only in part of their behaviour. Theorems 4.3 and 4.6 stated which part of those processes are bisimilar or similar. Conversely, the set of channels abstracted in such a comparison represents the unmatched behaviour. This means that neither the whole behaviour of S can be substituted, nor the entire behaviour of P can be reused. The interface process must then provide the unmatched behaviour and, at the same time, enable P to be run synchronously.

Once we have stated which part of S can be replaced by P and consequently which part of P has actually been used to substitute for S, the interface process can then be analysed. To be channel restricted bisimilar or similar, S and P are related in such a way that each action from S has a corresponding action from P. Then, to solve the interface equation, each S action must be simulated by composition of the corresponding P's action with an interface action (an action from the interface process X[2]).

Assuming that each S action is simulated by the corresponding channel restricted bisimilar/similar action from P composed with an X action, the analysis on the information to be included in X's actions must then be performed. To facilitate this investigation, we split this problem in two other more primitive problems: the input and the output behaviours that must be provided by the interface process.

To make this analysis still easier, we first concentrate on the initial actions necessary for the interface process to simulate the S's initial actions, and defer the analysis of the entire interface process to Section 5.4, where our new decomposition operator is presented. The forthcoming sections are devoted to checking the input and output behaviours of the initial actions of X as interfacing S and P.

Output Behaviour

The coincident output channels of P and S denote which output behaviour of P can substitute for the S's output behaviour. But the output behaviour of S not matched to P must be supplied by the interface process. Hence, to provide the entire output behaviour of S, only the output channels of S not coinciding with P need to be fixed: $\mathsf{Chs}_o(S) - \mathsf{Chs}_o(P)$.

On the other hand, the external environment is only prepared to generate/accept events that interact with S – it is only able to communicate through S's input and output channels. But, to reuse P, all of its input channels must be provided in order to make it run, in synchrony, with the interface process. Those input channels of P which coincide with S's channels can be obtained directly from the environment. Those other non–coincident input channels, however, must be provided by the interface process. Thus, the output channels of the interface process must comprise at least the following elements:

$$\mathsf{Chs}_o(X) = (\mathsf{Chs}_o(S) - \mathsf{Chs}_o(P)) \cup (\mathsf{Chs}_i(P) - \mathsf{Chs}_i(S))$$

For instance, if $S \xrightarrow{i_s/o_s} S'$ and $P \xrightarrow{i_p/o_p} P'$ such that $S \overset{c}{\sqsubseteq} P$, $S' \overset{c}{\sqsubseteq} P'$, and i_s/o_s and i_p/o_p are channel restricted identical, then the X's action (denoted as i_x/o_x) which interfaces the above S and P actions must comprise the output information:

$$o_x = (o_s \ominus o_p) \oplus (i_p \ominus i_s).$$

To satisfy the interface equation, composition of i_p/o_p with i_x/o_x, restricted to S's channels must be identical to the i_s/o_s:

$$\textit{res-ch}(i_x/o_x \star i_p/o_p, \mathsf{Chs}(S)) = i_s/o_s. \tag{5.3}$$

By definition, composition of input actions is given by $(i_x \oplus i_p) \ominus (o_x \oplus o_p)$ while composition of output actions is given by $o_x \oplus o_p$. Also, $\textit{res-ch}(i/o, C) = \textit{res-io-ch}(i, C)/\textit{res-io-ch}(o, C)$. Then, Equation 5.3 can be redefined by two other equations:

[2] As an abuse of notation, we use the variable X to represent any process that satisfies Equation 5.1.

$$
\begin{aligned}
\mathit{res\text{-}io\text{-}ch}((i_x \oplus i_p) \ominus (o_x \oplus o_p), \mathsf{Chs}(S)) &= i_s && (5.4)\\
\mathit{res\text{-}io\text{-}ch}(o_x \oplus o_p, \mathsf{Chs}(S)) &= o_s. && (5.5)
\end{aligned}
$$

Once we have defined the necessary output channels for the interface process, we can now check that the output behaviour of the compound action is identical to the output of S's action using definition of o_x as above:

$$
\begin{aligned}
\mathit{res\text{-}io\text{-}ch}(o_x \oplus o_p, \mathsf{Chs}(S)) &= o_s\\
\mathit{res\text{-}io\text{-}ch}((o_s \ominus o_p) \oplus (i_p \ominus i_s) \oplus o_p, \mathsf{Chs}(S)) &= o_s.
\end{aligned}
$$

A proof for the equality above has been included in Appendix C. So, the minimal set of output channels defined for X is sufficient to ensure that the output compound action is identical to the S's output action.

Assuming that S is channel restricted similar to P, and consequently there is no coincident S input and P output channels, the minimal set of output channels of the interface process is shown in Figure 5.3.

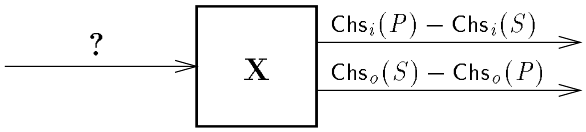

Figure 5.3: X Output Channels

Input Behaviour

Now, what must be provided as input action of the interface process to complete the identification of the compound action with the corresponding S actions? The obvious answer might be "all the non–coincident input channels of S", because this is the behaviour not provided by P ($\mathsf{Chs}_i(S) - \mathsf{Chs}_i(P)$).

This is indeed the minimal set of input channels that must be provided as interface behaviour of S's input actions in contrast to P's. But, to solve the interface equation, composition of X with P (restricted to the set of S channels) must behave bisimilarly to S. X must then be constructed in such a way to interface each S's action with its corresponding channel restricted action from P. So, for each pair of corresponding S and P actions, there must exist an X action to interface them. In particular, for each S's action, there exists a corresponding action from X which is going to simulate it.

To achieve bisimilarity with S however, each X action must comprise sufficient information to ensure that its combination with the range of P's actions exclusively provides the corresponding S action. In other words, each X action must be uniquely combined with the P's action to provide the behaviour of the corresponding S action: considering $C = \mathsf{Chs}(P) \cap \mathsf{Chs}(X)$, then property $\mathcal{D}!_X(P \restriction_c C)$ must hold. This required property has already been stated as necessary to achieve reuse of processes using the interface approach (Proposition 5.2, Section 5.2).

Summarizing, to solve the interface equation, composition of X with a P's action must be identical to the corresponding S's action ($\mathcal{D}!_X(P \restriction_c C)$ holds). If the set of X's input channels exclusively comprises non–coincident input channels ($\mathsf{Chs}_i(S) - \mathsf{Chs}_i(P)$) as suggested above, the only coincident channels of X and P is given by the set ($\mathsf{Chs}_i(P) - \mathsf{Chs}_i(S)$) (defined as output channels of X). Is this input and output behaviour of X sufficient to ensure bisimilarity of the compound process with S?

Whether only non–coincident input channels are given as input channels of X, P is available for reuse only if $\mathcal{D}!_X(P \restriction_c (\mathsf{Chs}_i(P) - \mathsf{Chs}_i(S)))$ because $\mathcal{D}!_X(P \restriction_c (\mathsf{Chs}(P) \cap \mathsf{Chs}(X)))$ is an essential property (as stated above). Notice that the more S and P coincide in channels, the more restricted is the set of non–coincident channels; $P \restriction_c (\mathsf{Chs}_i(P) - \mathsf{Chs}_i(S))$ has fewer channels as P and S are closer in behaviour.

In some cases, these channels are sufficient to have the process $P \restriction_c (\mathsf{Chs}_i(P) - \mathsf{Chs}_i(S))$ deterministically defined for its actions which does not conflict with X, but several other cases require additional channels to satisfy such a property.

In general, to make P available for reuse, some of its channels which coincide with S's channels must be added to X. There are basically two different approaches to include these channels: adding coincident channels one by one until P restricted to the coincident channels is deterministically defined for its conflict free actions ($\mathcal{D}!_X(P \restriction_c (\mathsf{Chs}(P) \cap \mathsf{Chs}(X)))$); or including all the coincident channels and then eliminating unnecessary channels at the end.

In the first approach, various alternative processes can be constructed with a minimal number of connections necessary to ensure the above mentioned properties. To reach such minimal processes, two distinct procedures can be adopted: various alternative processes are built simultaneously and then the choice of best solutions is deferred to the end, or a specific alternative is chosen to be constructed at the beginning without looking for best solutions. By contrast, the second approach "overconnects" X[3] and a further minimization is necessary to give the minimal set of communications of such a process. It is indeed the most generic process that can be minimized into many other processes; the best solutions are given or chosen after minimization. For reasons of creating a combinator to solve the interface equation, we adopted the second approach where the generic process is built up and then minimal processes can be found afterwards.

Concerning with the generic process approach, X's input channels must comprise all the S's input channels as much as all the P's output channels:

$$\mathsf{Chs}_i(X) = \mathsf{Chs}_i(S) \cup \mathsf{Chs}_o(P).$$

So, the set of X's channels comprises all the S and P channels:

$$\mathsf{Chs}(X) = (\mathsf{Chs}_o(S) - \mathsf{Chs}_o(P)) \cup (\mathsf{Chs}_i(P) - \mathsf{Chs}_i(S)) \cup (\mathsf{Chs}_i(S) \cup \mathsf{Chs}_o(P)).$$

As a result, the constraint $\mathcal{D}!_X(P \restriction_c (\mathsf{Chs}(P) \cap \mathsf{Chs}(X)))$ can be redefined as $\mathcal{D}!_X(P)$ because $(\mathsf{Chs}(P) \cap \mathsf{Chs}(X)) = \mathsf{Chs}(P)$. Considering that each X's action is constructed from information of S and P corresponding actions, the P's actions that are conflict free with X's actions are also conflict free with the corresponding S's action. So, the constraint $\mathcal{D}!_X(P)$ is reduced to $\mathcal{D}!_S(P)$. The property of uniqueness of combination of actions between X and P required to reuse P is redefined as:

Lemma 5.2 *Suppose $S \overset{c}{\sqsubseteq} P$, then P can be reused to simulate S only if $\mathcal{D}!_S(P)$.*

Considering $S \xrightarrow{i_s/o_s} S'$ and $P \xrightarrow{i_p/o_p} P'$ such that $S \overset{c}{\sqsubseteq} P$, $S' \overset{c}{\sqsubseteq} P'$, and i_s/o_s and i_p/o_p are channel restricted identical, the corresponding X's action which interfaces the above S and P actions is as follows:

$$i_x/o_x = (i_s \oplus o_p)/(o_s \ominus o_p) \oplus (i_p \ominus i_s).$$

Once the input behaviour of i_x/o_x has been entirely defined, it can now be checked if composition of these actions behaves like i_s/o_s. Equality of such composition with the S action (i_s/o_s) has been defined by a conjunction of equalities of input and output actions (Equations 5.4 and 5.5). We have already verified that composition of i_x/o_x with i_p/o_p, restricted to S's channels achieve o_s's behaviour. To complete equality of the entire action, we must now check equality of the input actions:

$$\begin{aligned} \mathit{res\text{-}io\text{-}ch}((i_x \oplus i_p) \ominus (o_x \oplus o_p), \mathsf{Chs}(S)) &= i_s \\ \mathit{res\text{-}io\text{-}ch}((i_s \oplus o_p \oplus i_p) \ominus (((o_s \ominus o_p) \oplus (i_p \ominus i_s)) \oplus o_p), \mathsf{Chs}(S)) &= i_s. \end{aligned}$$

In fact, this equality holds and a proof for that has been included in Appendix C. Thus, each pair of conflict free actions from P and X behaves bisimilarly to a corresponding S's action.

[3] "Overconnected" processes are those which have unnecessary ports of communication; channels that can be eliminated without affecting the behaviour of the compound process.

Corollary 5.2 *Suppose i_s/o_s, i_p/o_p and $i_x/o_x = (i_s \oplus o_p)/(o_s \ominus o_p) \oplus (i_p \ominus i_s)$ are io–actions such that chs-of$(i_s) \cap$chs-of$(o_p) = \emptyset$ and chs-of$(o_s) \cap$chs-of$(i_p) = \emptyset$, then res-ch$(i_p/o_p \star i_x/o_x, \mathsf{Chs}(i_s/o_s)) = i_s/o_s$.*

Now, Figure 5.3 can be completed with the input channels of process X. Figure 5.4 shows all necessary input and output channels for the interface process.

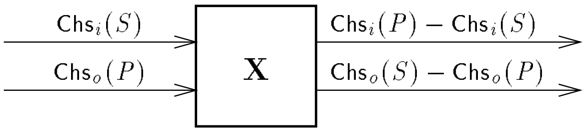

Figure 5.4: X Input and Output Channels

Bisimilarity of Initial Actions

Once we have checked that a single action from S and the compound process are identical when X actions are defined as above, the set of initial actions of S and the compound process can also be checked as identical.

Firstly, the X's initial actions are defined as suggested above.

Definition 5.6 *Assume $S \stackrel{c}{\sqsubseteq} P$ and $\mathcal{D}!_S(P)$. Whenever $S \xrightarrow{i_s/o_s} S'$ and $P \xrightarrow{i_p/o_p} P'$ such that $S' \stackrel{c}{\sqsubseteq} P'$ and res-ch$(i_s/o_s, C) =$ res-ch$(i_p/o_p, C)$, then X has the transition $X \xrightarrow{i_x/o_x} X'$ for some X' such that $i_x/o_x = (i_s \oplus o_p)/(o_s \ominus o_p) \oplus (i_p \ominus i_s)$.*

As a consequence of property $\mathcal{D}!_S(P)$, if X's initial actions are defined as above, then each of these actions can be combined only with the action from P which was used to define it:

Proposition 5.5 *Suppose $S \stackrel{c}{\sqsubseteq} P$, $\mathcal{D}!_S(P)$ and X's initial actions are built up as in Definition 5.6. Then, for any action $i_x/o_x \in \mathsf{Sort}_1(X)$, there exists only one action $i_p/o_p \in \mathsf{Sort}_1(P)$ such that i_x/o_x CF i_p/o_p.*

Proof. For this proposition, we must prove the existence of a unique corresponding i_p/o_p action and conflict freedom of i_p/o_p with i_x/o_x.

Since $S \stackrel{c}{\sqsubseteq} P$, whenever $S \xrightarrow{i_s/o_s} S'$ then $P \xrightarrow{i_p/o_p} P'$ for some P' such that *res-ch*$(i_s/o_s, (\mathsf{Chs}(S) \cap \mathsf{Chs}(P))) =$ *res-ch*$(i_p/o_p, (\mathsf{Chs}(S) \cap \mathsf{Chs}(P)))$ (it implies that i_s/o_s CF i_p/o_p). By definition of X initial actions, there is exactly one action i_x/o_x in X for each pair of actions i_s/o_s and i_p/o_p. For instance, there exists an action i_p/o_p conflict free with i_x/o_x. Properties $\mathcal{D}!_S(P)$ and $\mathsf{Chs}(P) \subseteq \mathsf{Chs}(X)$ ensure that i_p/o_p is unique.

Moreover, i_x/o_x is exclusively defined by "summation and subtractions" of actions i_s/o_s and i_p/o_p. Since i_s/o_s CF i_p/o_p and summation and subtraction preserve conflict freedom of actions, i_x/o_x CF i_p/o_p. □

Thus, equality of sets of initial actions of S and the compound process is defined as follows:

Theorem 5.3 *If $S \stackrel{c}{\sqsubseteq} P$, $\mathcal{D}!_S(P)$ and X's prefixing actions are built up as in Definition 5.6, then $\mathsf{Sort}_1((P \mid X) \restriction_c \mathsf{Chs}(S)) = \mathsf{Sort}_1(S)$.*

Proof. To prove such a theorem two assertions must be satisfied: there is a compound action identical to each S action, and the compound process only performs S actions.

By definition of the X initial actions (Definition 5.6), there exists an action i_x/o_x for each channel restricted identical actions from S (i_s/o_s) and P (i_p/o_p). At the same time, each X action, i_x/o_x, is conflict free with only one P action (i_p/o_p) as stated in Proposition 5.5. By Corollary 5.2, the compound action is identical to the S's corresponding action. Then, composition of all X initial actions with the conflict free actions from P can only perform S actions. □

It is important to notice that even when P performs actions that have no corresponding channel restricted action from S, composition of X with P only performs S actions. Since composition is a synchronous combinator, a compound action only exists if both processes can run concurrently. But, X actions are conflict free only with P actions that are channel restricted identical to S's. So, the P's actions not related to S's might never be stimulated under composition with X.

Proposition 5.6 *Suppose $S \stackrel{c}{\sqsubseteq} P$, $\mathcal{D}!_S(P)$ and X's prefixing actions are built up as in Definition 5.6. If $P \xrightarrow{\beta} P'$ but there is no action $\alpha \in \mathsf{Sort}_1(S)$ such that $\alpha \ \mathsf{CF}\ \beta$, then there does not exist any action γ from X such that $\gamma \ \mathsf{CF}\ \beta$.*

We have proved so far that the sets of initial actions of the compound process and S are identical, but we have not analysed bisimilarity between these processes yet. Bisimilarity of S with the compound process depends on equality of initial actions as much as bisimilarity between their reachable processes. Obviously, this analysis depends on the definition of X's reachable processes. We defer the creation of X's reachable processes to Section 5.4 where a decomposition combinator is defined.

5.3.4 "Channel Restricted Simulation under a Functional Interface" Relations

Like $\mathcal{B}_c$ and $\mathcal{S}_c$, processes related by channel restricted simulation under (total) a functional interface have only a subset of channels compared. Thus, only a channel restricted S is bisimilar or similar to a channel restricted compound process, as stated in Theorems 4.4 and 4.7. A solution for the interface equation for processes related by $\mathcal{S}_{ctf}$ or $\mathcal{S}_{cf}$ also requires creation of an interface process.

Considering $S \stackrel{cf}{\sqsubseteq} P$ such that the set of channels C and function f satisfy such a relation, a single state process P_f can then be constructed from f (as stated in Definition 4.3). Since a channel restricted behaviour of S is interfaced with a channel restricted behaviour of P (through f), channel restricted S's input and output behaviour is not directly simulated by P. But composition of P with P_f makes such a simulation available (Proposition 4.2 states: $S \stackrel{c}{\sqsubseteq} (P \mid P_f)$).

So, $P_I \triangleq (P \mid P_f)$[4] can be considered as an existing process which is channel restricted similar to S. Considering that, the problem of finding an interface process for reusing channel restricted similar processes under a functional interface is reduced to the problem of finding the interface process for channel restricted similar processes, except that the existing process is P_I instead of P.

Constraints

Considering P_I as the existing process, the constraints necessary to reuse it in implementing S must now be redefined based on Lemma 5.2:

Corollary 5.3 *Suppose $S \stackrel{cf}{\sqsubseteq} P$ such that C and f are, respectively, a set of channels and a function which satisfy this relation, and $P_I \triangleq (P \mid Pf)$. Then, P can be reused to implement S only if $\mathcal{D}!_S(P_I)$.*

Determinacy of P_I with respect to conflict free actions with S cannot be analysed over P because it involves the product of P with P_f, where some channels are hidden by the end. So, construction of P_I is required to analyse the conditions in which P can be reused

Input and Output Behaviour

Definition of input and output actions of the interface process to reuse channel restricted similar processes depends on the input and output actions of S and P. At this point, S's behaviour and its corresponding channels are known, as well as the channels and behaviour of P_I. The input and output channels of the interface process can thus be defined as a case of channel restricted similar processes. Figure 5.5 shows P_I being composed with the corresponding interface process X'.

[4] P_I is a P interfaced process: P_f interfaces a subset of S and P channels.

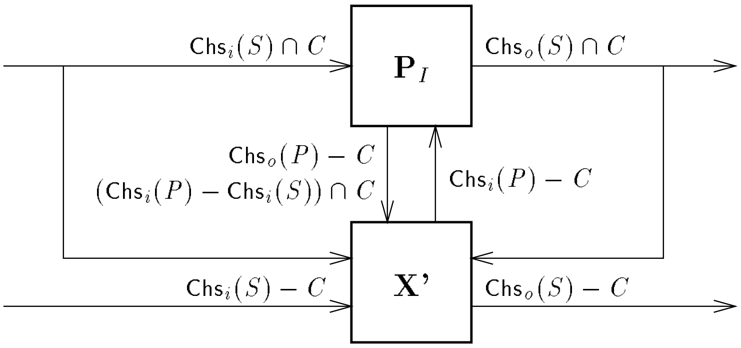

Figure 5.5: Composition of P_I with its Interface Process X'

Notice that P channels that interact with P_f to simulate the channel restricted S are unobservable from the outside of P_I. X' interacts with P through those P channels which do not interface the channel restricted S. Also, interaction between X' and P_f is denoted by those S input and output channels that are simulated by P through P_f.

Bisimilarity of Initial Actions

Assuming that P_I exists whenever $S \overset{cf}{\sqsubseteq} P$, bisimilarity of initial actions S and the compound process $(P_I \mid X')$ can be inferred from Theorem 5.3 and from the fact that P_I is a channel restricted simulation of S. In fact, the existence of P_I is inferred from the existence of P_f and the relation between S and P_I is inferred from Proposition 4.3:

Corollary 5.4 *If $S \overset{cf}{\sqsubseteq} P$ such that C and f are, respectively, a set of channels and a function which satisfy such relation, and $P_I \underline{\triangle} (P \mid P_f)$, then $S \overset{c}{\sqsubseteq} P_I$.*

Thus, assuming P_I as an existing process, initial actions of X' can be defined and its composition with P_I behaves like S's initial actions.

Theorem 5.4 *Suppose $S \overset{cf}{\sqsubseteq} P$ such that C and f are, respectively, a set of channels and a function which satisfy this relation, $P_I \underline{\triangle} (P \mid P_f)$ and $\mathcal{D}!_S(P_I)$. If P_I is considered as the existing process and X''s initial actions are built up using Definition 5.6, then $\mathsf{Sort}_1(P_I \mid X') \upharpoonright_c \mathsf{Chs}(S) = \mathsf{Sort}_1(S)$.*

Proof. By Corollary 5.4, $S \overset{c}{\sqsubseteq} P_I$. If P_I is assumed as an existing process, then X' can be defined by Definition 5.6 (page 89). Also Theorem 5.3 shows that the set of initial actions of $(P \mid X) \upharpoonright_c \mathsf{Chs}(S)$ is identical to the S's initial actions set. So, $\mathsf{Sort}_1(P_I \mid X') \upharpoonright_c \mathsf{Chs}(S) = \mathsf{Sort}_1(S)$ holds.

□

We must emphasize that X' is not a straightforward solution for the interface equation when considering P as the existing process, but a solution for such an equation when the compound process $(P \mid P_f)$ is assumed as the existing one. To solve Equation 5.1, process $(P_f \mid X')$ has to be found. Since P_f is a known process, only the remaining process X' needs to be created.

5.4 A Solution for the Interface Equation

In cases where the existing process is bisimilar (similar, strong similar, similar and strong similar under a functional interface) to the specification, as shown in the previous sections, none or a single state interface process is required to yield the desired behaviour. Moreover, we have shown how to find such a single state process for each of those cases. Since channel restricted simulation under a functional interface is the weakest similarity relation on processes which are suitable for reuse, a solution for the interface equation considering processes related by such a relation provides a general solution that can be carried over the other relations.

The discussion in Section 5.3.4 showed that if processes are related by $\mathcal{S}_{cf}$, then the whole process necessary to interface the existing one can be split into a single state process P_f composed with another interface process. Suppose S and P are related by $\mathcal{S}_{cf}$ (they could actually be related by a stronger similarity relation) and P_f is a known process constructed from f which satisfies $\mathcal{S}_{cf}$, Equation 5.1 is rewritten as[5]:

$$(P_I \mid X) \restriction_c \mathsf{Chs}(S) \sim S. \tag{5.6}$$

Considering that $S \overset{c}{\sqsubseteq} P_I$, the problem of finding a solution for Equation 5.6 when S is channel restricted similar to P under a functional interface is reduced to finding a solution for Equation 5.1 when the involved processes related by channel restricted simulation.

In the following sections we state the necessary and sufficient conditions to solve Equation 5.6, and show that the application of a decomposition operator based on channel restricted similar processes is a solution for such an equation.

5.4.1 Necessary and Sufficient Conditions

Sections 5.2 and 5.3 showed that a process P_I, even when it is similar to S, can only be reused if it satisfies certain constraints (Corollary 5.3). In fact, if the specification and the given components satisfy these conditions, then a solution for the interface equation can be found:

Corollary 5.5 *Suppose $S \overset{cf}{\sqsubseteq} P$ such that C and f are, respectively, a set of channels and a function which satisfy this relation, $P_I \trianglelefteq (P \mid P_f)$ and $\mathcal{D}!_S(P_I)$. Then, $(P_I \mid X) \restriction_c \mathsf{Chs}(S) \sim S$ (Equation 5.6) can be solved.*

Since these constraints are necessary to allow reuse, we assume the given and the desired processes satisfy them whenever an interface process is required.

Bisimilarity between processes depends on the ability of both processes performing identical initial actions and then reaching bisimilar processes. Therefore, a solution for equation 5.6 depends upon equality of the initial actions of S and the compound process, as well as upon solutions for equations defined over the related reachable processes. Now, instead of naming X as any solution (as we did in the previous sections), let us assume Q as a solution for this equation. The following proposition gives the necessary conditions to have Q as a solution for such an equation:

Proposition 5.7 *Assume $S \overset{cf}{\sqsubseteq} P$ such that C and f are, respectively, a set of channels and a function which satisfy this relation, $P_I \trianglelefteq (P \mid P_f)$ and $\mathcal{D}!_S(P_I)$. Suppose i_s/o_s, i_p/o_p and i_q/o_q are proper actions; $Q \xrightarrow{i_q/o_q} Q'$ and $P_I \xrightarrow{i_p/o_p} P'_I$ such that $res\text{-}ch(i_p/o_p \star i_q/o_q, \mathsf{Chs}(S)) = i_s/o_s$ and $S \xrightarrow{i_s/o_s} S'$. Then, Q is a solution for $(P_I \mid X) \restriction_c \mathsf{Chs}(S) \sim S$ only if it satisfies the following properties:*

1. $\mathsf{Sort}_1((P_I \mid Q) \restriction_c \mathsf{Chs}(S)) = \mathsf{Sort}_1(S)$, *and*
2. *Q' is a solution for $(P'_I \mid X) \restriction_c \mathsf{Chs}(S) \sim S'$.*

The first condition requires all initial actions of the compound process to be simulated by S and vice–versa – for every move from either S or $(P_I \mid Q) \restriction_c \mathsf{Chs}(S)$, there must exist an identical move in the other process. The second condition is also necessary in order to guarantee that both processes reach bisimilar processes whenever performing identical moves.

If both properties hold, for each move from S there exists an identical move from the compound process and vice–versa (by first condition). Also, both processes reach bisimilar processes after performing

[5] A discussion on S's channels being simulated by composition of P with P_f can be found in Section 5.3.4

identical moves, as fixed by the second condition. Notice that these conditions are indeed sufficient to guarantee bisimilarity of processes. Therefore, beyond being necessary, the conditions in Proposition 5.7 are *sufficient* to achieve a solution for Equation 5.6.

Theorem 5.5 *Assume $S \overset{cf}{\sqsubseteq} P$ such that C and f are, respectively, a set of channels and a function which satisfy this relation, $P_I \trianglelefteq (P \mid P_f)$ and $\mathcal{D}!_S(P_I)$. Suppose i_s/o_s, i_p/o_p and i_q/o_q are proper actions. If*

1. $\mathsf{Sort}_1((P_I \mid Q) \restriction_c \mathsf{Chs}(S)) = \mathsf{Sort}_1(S)$, *and*
2. *Q' is a solution for $(P'_I \mid X) \restriction_c \mathsf{Chs}(S) \sim S'$*

whenever $Q \xrightarrow{i_q/o_q} Q'$, $S \xrightarrow{i_s/o_s} S'$ and $P_I \xrightarrow{i_p/o_p} P'_I$ such that $\mathit{res\text{-}ch}(i_p/o_p \star i_q/o_q, \mathsf{Chs}(S)) = i_s/o_s$, then Q is a solution for $(P_I \mid X) \restriction_c \mathsf{Chs}(S) \sim S$.

Proof. The first property ensures that for any initial action of S, there exists an identical initial action in the compound process and vice–versa because their sets of initial actions are identical. Then, it satisfies the property of bisimulation which says that

(a) whenever $((P_I \mid Q) \restriction_c \mathsf{Chs}(S)) \xrightarrow{i_s/o_s} ((P'_I \mid Q') \restriction_c \mathsf{Chs}(S))$ then $S \xrightarrow{i_s/o_s} S'$ and

(b) whenever $S \xrightarrow{i_s/o_s} S'$ then $((P_I \mid Q) \restriction_c \mathsf{Chs}(S)) \xrightarrow{i_s/o_s} ((P'_I \mid Q') \restriction_c \mathsf{Chs}(S))$.

because $\mathit{res\text{-}ch}(i_p/o_p \star i_q/o_q, \mathsf{Chs}(S)) = i_s/o_s$.

Also, by the second property, the compound process and S reach bisimilar processes when performing identical moves. So, it completes the definition above as:

(a) whenever $((P_I \mid Q) \restriction_c \mathsf{Chs}(S)) \xrightarrow{i_s/o_s} ((P'_I \mid Q') \restriction_c \mathsf{Chs}(S))$ then $S \xrightarrow{i_s/o_s} S'$ for some S' such that $((P'_I \mid Q') \restriction_c \mathsf{Chs}(S)) \sim S'$ and

(b) whenever $S \xrightarrow{i_s/o_s} S'$ then $((P_I \mid Q) \restriction_c \mathsf{Chs}(S)) \xrightarrow{i_s/o_s} ((P'_I \mid Q') \restriction_c \mathsf{Chs}(S))$ for some Q' and P'_I such that $((P'_I \mid Q') \restriction_c \mathsf{Chs}(S)) \sim S'$.

Thus, the second condition completes the definition of bisimulation. These conditions are then sufficient to ensure that Q is a solution for Equation 5.6.

□

5.4.2 A Decomposition Operator

Creation of a combinator for process algebras can only be accomplished if its application is able to find a unique result. In this section, we present a decomposition operator and claim that its use yields a particular solution for the interface equation in which minimal solutions are included.

The information to be provided as input and output actions of the interface process has been discussed in Section 5.3. Actions of the interface process are created by taking each action from the desired process and "dividing" it by the similar action from the existing one, as suggested in Definition 5.6. Interface actions are therefore created for each pair of similar actions from S and P_I actions. Suppose i_s/o_s and i_p/o_p are similar actions from S and P_I respectively, then the interface process must move through actions that are "divisions" of S by P_I actions:

$$
\begin{aligned}
&\div : Act^2 \rightarrow Act \\
&\forall\ i_p/o_p, i_q/o_q \in Act \bullet \\
&\qquad (i_p/o_p) \div (i_q/o_q) = (i_p \oplus o_q)/(o_p \ominus o_q) \oplus (i_q \ominus i_p)
\end{aligned}
$$

Since the creation of the interface actions depends on similarity between S and P_I actions, as previously defined, the decomposition operator must only permit division of similar actions. To do so, decomposition of process S by P_I must embed conditions which guarantee that only similar actions are going to be divided. Using Definition 5.6 as basis, $(S \ / \ P_I) \xrightarrow{i_s/o_s \div i_p/o_p} (S' \ / \ P'_I)$ only if $S \xrightarrow{i_s/o_s} S'$, $P_I \xrightarrow{i_p/o_p} P'_I$, $\mathit{res\text{-}ch}(i_s/o_s, \mathsf{Chs}(P_I) \cap \mathsf{Chs}(S)) = \mathit{res\text{-}ch}(i_p/o_p, \mathsf{Chs}(P_I) \cap \mathsf{Chs}(S))$ and $S' \overset{c}{\sqsubseteq} P'_I$.

By the second property required to solve the interface equation (Theorem 5.5), the Q's following processes (Q') must solve equations like $(P'_I \mid X) \lceil_c \mathsf{Chs}(S) \sim S'$. But it is reduced to the problem of finding a decomposed process for S' with respect to P'_I. So, the process reached by Q when performing that division action must be a decomposition of S' with respect to P'_I.

Based on these definitions of initial actions and reachable processes, a transitional semantics for the decomposition operator is as follows:

$$\boxed{\mathsf{Div}}\ \frac{S \xrightarrow{\alpha} S',\quad P_I \xrightarrow{\beta} P'_I,\quad \{(S, P_I), (S', P'_I)\} \subseteq \mathcal{S}_c,\qquad \textit{res-ch}(\alpha, \mathsf{Chs}(P_I) \cap \mathsf{Chs}(S)) = \textit{res-ch}(\beta, \mathsf{Chs}(P_I) \cap \mathsf{Chs}(S))}{S \ / \ P_I \xrightarrow{\alpha \div \beta} S' \ / \ P'_I}$$

This operator constructs "overconnected" processes which can be minimized. In principle, the new process is created on the size of pairs of S and P_I that are related by channel restricted simulation. So, a further minimization must be performed based on minimal interactive processes. Also minimization of connections must be performed to remove unnecessary inputs. Minimization of decomposed processes is deferred to Chapter 6.

Due to the fact that the decomposed processes satisfies all properties of Theorem 5.5, decomposition provides then a solution for Equation 5.6.

Theorem 5.6 *Suppose* $S \overset{cf}{\sqsubseteq} P$ *such that* C *and* f *are, respectively, a set of channels and a function which satisfy this relation,* $P_I \trianglelefteq (P \mid P_f)$ *and* $\mathcal{D}!_S(P_I)$. *Then, process* $S \ / \ P_I$ *is a solution for* $(P_I \mid X) \lceil_c \mathsf{Chs}(S) \sim S$ *(Equation 5.6).*

5.4.3 Soundness of the Decomposition Operator

Satisfiability of decomposed processes as a general solution for the interface equation has been stated in Theorem 5.6. A proof for that can be given by satisfiability of the two necessary and sufficient properties to solve the interface equation (stated in Theorem 5.5):

1. $\mathsf{Sort}_1((P_I \mid Q) \lceil_c \mathsf{Chs}(S)) = \mathsf{Sort}_1(S)$ and
2. Q' is a solution for $(P'_I \mid X) \lceil_c \mathsf{Chs}(S) \sim S'$.

This section is devoted to proving that application of the presented decomposition operator is a solution for the interface equation. In doing so, the essential properties defined in Theorem 5.5 are proved in the two following sections.

Equality of Initial Actions

As a consequence of being based on Definition 5.6, application of the decomposition operator results in a process with identical properties of processes created from such a definition. Therefore, each $(S \ / \ P_I)$ action is conflict free with at least one action from P_I. Moreover, if P_I is deterministically defined for its actions that do not conflict with the S's ones, each $(S \ / \ P_I)$ action is conflict free with exactly one P_I action.

Corollary 5.6 *Assume* $S \overset{cf}{\sqsubseteq} P$ *such that* C *and* f *are, respectively, a set of channels and a function which satisfy this relation,* $P_I \trianglelefteq (P \mid P_f)$ *and* $\mathcal{D}!_S(P_I)$. *If* $i_q/o_q \in \mathsf{Sort}_1(S \ / \ P_I)$ *then there exists a unique action* $i_p/o_p \in \mathsf{Sort}_1(P_I)$ *such that* i_q/o_q CF i_p/o_p.

Theorem 5.3, in Section 5.3.4, stated that if S is channel restricted similar to P_I and initial actions of the interface process are defined as in Definition 5.6, then the set of initial actions of S equals the set of initial actions of the compound process (the property on equality of initial actions is satisfied). Uniqueness of conflict free actions of $(S\ /\ P_I)$ and P_I makes composition of these processes to have a set of initial actions identical to S's initial actions. The following corollary is a result from Theorem 5.3 and Corollary 5.6:

Corollary 5.7 *Suppose* $S \overset{cf}{\sqsubseteq} P$ *such that* C *and* f *are, respectively, a set of channels and a function which satisfy this relation,* $P_I \trianglelefteq (P \mid P_f)$ *and* $\mathcal{D}!_S(P_I)$*. Then,* $\mathsf{Sort}_1((P_I \mid (S\ /\ P_I)) \upharpoonright_c \mathsf{Chs}(S)) = \mathsf{Sort}_1(S)$*.*

Solutions for the Dependent Equations

Here we outline a proof of the assertion: all the equations on which the main interface equation depend are solved by application of the decomposition operator. To accomplish this, we show that all the dependent equations can be solved and they are indeed solved by application of the decomposition operator.

Given the constraints to apply the decomposition operator, for each action from S, there exists a corresponding channel restricted action from P_I, and consequently an action from Q $(Q \trianglelefteq (S\ /\ P_I))$ which interfaces those actions. Uniqueness of conflict free actions from P_I and Q ensures that each S action is exclusively simulated by the corresponding channel restricted action from P_I composed with Q. Consequently, for each move of S, P_I undertakes the corresponding action, and by the time S reaches a process S', P_I reaches a process P'_I which is channel restricted similar to S'. This is a result of Corollary 5.6.

Corollary 5.8 *Assume* $S \overset{cf}{\sqsubseteq} P$ *such that* C *and* f *are, respectively, a set of channels and a function which satisfy this relation,* $P_I \trianglelefteq (P \mid P_f)$*,* $\mathcal{D}!_S(P_I)$ *and* $Q \trianglelefteq S\ /\ P_I$*. Suppose* i_s/o_s*,* i_p/o_p *and* i_q/o_q *are proper actions;* $S \xrightarrow{i_s/o_s} S'$*,* $P_I \xrightarrow{i_p/o_p} P'_I$ *and* $Q \xrightarrow{i_q/o_q} Q'$ *such that* $res\text{-}ch(i_p/o_p, \mathsf{Chs}(S)) = res\text{-}ch(i_s/o_s, \mathsf{Chs}(S))$ *and* i_p/o_p CF i_q/o_q*. Then* $S' \overset{c}{\sqsubseteq} P'_I$ *and* $(P'_I \mid X) \upharpoonright_c \mathsf{Chs}(S) \sim S'$ *can be solved.*

The fact that S is channel restricted similar to P_I implies they reach channel restricted similar processes when making identical channel restricted moves. As a result, the dependent equations to be further solved relate channel restricted similar processes ($(P'_I \mid X) \upharpoonright_c \mathsf{Chs}(S) \sim S'$ and so on) and can then be solved.

In fact, all these equations are solved by application of the decomposition operator to their respective specification and given components. To outline the proof of this assertion, we first consider the case where all the reachable processes are bisimilar to either Stop or the process being addressed, and then show the general case.

Suppose $S \overset{c}{\sqsubseteq} P_I$, $\mathcal{D}!_S(P_I)$, each dependent equation is represented by $(P'_I \mid X) \upharpoonright_c \mathsf{Chs}(S) \sim S'$, and Q' is a solution for this dependent equation.

Basis Cases: The basis cases are given when the interface equation has already been solved by only providing the initial actions of Q; no dependent equations exist.

1. $S' \sim \mathsf{Stop}$: if the reachable processes are the process Stop, then Q' has no moves because S' has no moves to be simulated. So, $(S'\ /\ P'_I)$ is a solution for $(P'_I \mid X) \upharpoonright_c \mathsf{Chs}(S) \sim S'$ because $(S'\ /\ P') \sim \mathsf{Stop}$.
2. $S' \sim S$: in this case, a solution of the interface equation for S and P_I depends only on the existence of initial actions because the same process is reached. So, $(S\ /\ P_I)$ is also proved to satisfy the dependent interface equation by Corollary 5.7.

General: All the dependent equations can be solved because $S' \overset{c}{\sqsubseteq} P'_I$. Considering that only finite state processes are taken into account, a dependent equation may only depend on a finite number of equations. In fact, each equation depends on a tree of dependent equations and its solution is subjected to a solution for each "leaf equation". The leaves of such a tree can be:

1. an equation involving bisimilarity with Stop: the equation is then solved because the application of the decomposition operator gives a process Stop.
2. an equation which is in the path of such a dependent tree (a cycle in the dependent tree): since the application of the decomposition operator determines the initial actions of the interface process and it is recursively applied to find the reachable processes, this case is reduced to the second basis case above.

5.4.4 Sample: A Traffic Light Interface Process

Having established a decomposition operator to solve the interface equation, we can now create the interface process to be composed with the major road process in order to yield bisimilarity with the major/minor roads process. Once these processes have been proved to be channel restricted bisimilar (in Section 4.3) and property $\mathcal{D}!_{TLC_2}(TLC_{mj})$ holds, the decomposition operator can be applied to these processes (TLC_2 / TLC_{mj}) to find the interface process.

As checked in Section 4.3,

$$\begin{aligned}
&\{(TLC_2(s, t/l_{mj}, l_{mn}, setc), TLC_{mj}(s, t/l_{mj}, setc)),\\
&(Amb_2(s, t/l_{mj}, l_{mn}, setc), Amb_{mj}(s, t/l_{mj}, setc)),\\
&(FRed_2(s, t/l_{mj}, l_{mn}, setc), FRed_{mj}(s, t/l_{mj}, setc)),\\
&(SRed_2(s, t/l_{mj}, l_{mn}, setc), SRed_{mj}(s, t/l_{mj}, setc))\} \subseteq \mathcal{B}_c.
\end{aligned}$$

So, for each action from $TLC_2(s, t/l_{mj}, l_{mn}, setc)$ there exists an action from $TLC_{mj}(s, t/l_{mj}, setc)$ which partially simulates that $TLC_2(s, t/l_{mj}, l_{mn}, setc)$ action. For instance, action $(s(car), t(\overline{t_f})/l_{mj}(A), l_{mn}(R), setc(st))$ from $TLC_2(s, t/l_{mj}, l_{mn}, setc)$ is channel restricted identical to action $(s(car), t(\overline{t_f})/ l_{mj}(A), setc(st))$, from $TLC_{mj}(s, t/l_{mj}, setc)$, and they reach channel restricted bisimilar processes (vide the transition graphs for these processes in Appendix D). The interface process, therefore, comprises an action which interfaces these actions. The interface action for the actions above is as follows:

$$\begin{aligned}
&(s(car), t(\overline{t_f})/l_{mj}(A), l_{mn}(R), setc(st)) \div (s(car), t(\overline{t_f})/l_{mj}(A), setc(st)) =\\
&(s(car), t(\overline{t_f}), l_{mj}(A), setc(st)/l_{mn}(R))
\end{aligned}$$

The entire process $TLC_2(s, t/l_{mj}, l_{mn}, setc)$ / $TLC_{mj}(s, t/l_{mj}, setc)$ is given as follows (its transition graph is included in Appendix D):

$$\begin{aligned}
&\mathsf{process}\ (TLC_2(s\colon\ sensor, t\colon\ intervs/l_{mj}, l_{mn}\colon\ colour, setc\colon\ stcount)\ /\\
&\quad TLC_{mj}(s\colon\ sensor, t\colon\ intervs/l_{mj}\colon\ colour, setc\colon\ stcount)) \underline{\triangle}\\
&\quad\ (s(car), t(\overline{t_f}), l_{mj}(A), setc(st)/l_{mn}(R))\\
&\quad\ \ ::\ (Amb_2(s, t/l_{mj}, l_{mn}, setc)\ /\ Amb_{mj}(s, t/l_{mj}, setc)) +\\
&\quad\ (s(car), t(t_f), l_{mj}(G), setc(\overline{st})/l_{mn}(R))\\
&\quad\ \ ::\ (TLC_2(s, t/l_{mj}, l_{mn}, setc)\ /\ TLC_{mj}(s, t/l_{mj}, setc)) +\\
&\quad\ (s(\overline{car}), t(\overline{t_f}), l_{mj}(G), setc(\overline{st})/l_{mn}(R))\\
&\quad\ \ ::\ (TLC_2(s, t/l_{mj}, l_{mn}, setc)\ /\ TLC_{mj}(s, t/l_{mj}, setc)) +\\
&\quad\ (s(\overline{car}), t(t_f), l_{mj}(G), setc(\overline{st})/l_{mn}(R))\\
&\quad\ \ ::\ (TLC_2(s, t/l_{mj}, l_{mn}, setc)\ /\ TLC_{mj}(s, t/l_{mj}, setc))
\end{aligned}$$

$$
\begin{array}{l}
\textsf{process}\ (Amb_2(s\colon\ sensor, t\colon\ intervs/l_{mj}, l_{mn}\colon\ colour, setc\colon\ stcount)\ /\\
\qquad Amb_{mj}(s\colon\ sensor, t\colon\ intervs/l_{mj}\colon\ colour, setc\colon\ stcount)) \triangleq\\
\qquad\quad (s(car), t(\overline{t_s}), l_{mj}(R), setc(st)/l_{mn}(G))\\
\qquad\quad\ ::\ FRed_2(s, t/l_{mj}, lmn, setc)\ /\ FRed_{mj}(s, t/l_{mj}, setc)) +\\
\qquad\quad (s(car), t(t_s), l_{mj}(A), setc(\overline{st})/l_{mn}(R))\\
\qquad\quad\ ::\ Amb_2(s, t/l_{mj}, l_{mn}, setc)\ /\ Amb_{mj}(s, t/l_{mj}, setc)) +\\
\qquad\quad (s(\overline{car}), t(\overline{t_s}), l_{mj}(G), setc(st)/l_{mn}(R))\\
\qquad\quad\ ::\ TLC_2(s, t/l_{mj}, l_{mn}, setc)\ \ /\ TLC_{mj}(s, t/l_{mj}, setc))+\\
\qquad\quad (s(\overline{car}), t(t_s), l_{mj}(G), setc(st)/l_{mn}(R))\\
\qquad\quad\ ::\ TLC_2(s, t/l_{mj}, l_{mn}, setc)\ /\ TLC_{mj}(s, t/l_{mj}, setc))
\end{array}
$$

$$
\begin{array}{l}
\textsf{process}\ (FRed_2(s\colon\ sensor, t\colon\ intervs/l_{mj}, l_{mn}\colon\ colour, setc\colon\ stcount)\ /\\
\qquad FRed_{mj}(s\colon\ sensor, t\colon\ intervs/l_{mj}\colon\ colour, setc\colon\ stcount)) \triangleq\\
\qquad\quad (s(car), t(t_f), l_{mj}(R), setc(\overline{st})/l_{mn}(G))\\
\qquad\quad\ ::\ FRed_2(s, t/l_{mj}, l_{mn}, setc)\ /\ FRed_{mj}(s, t/l_{mj}, setc)) +\\
\qquad\quad (s(car), t(\overline{t_f}), l_{mj}(R), setc(st)/l_{mn}(A))\\
\qquad\quad\ ::\ SRed_2(s, t/l_{mj}, l_{mn}, setc)\ /\ SRed_{mj}(s, t/l_{mj}, setc)) +\\
\qquad\quad (s(\overline{car}), t(\overline{t_f}), l_{mj}(R), setc(st)/l_{mn}(A))\\
\qquad\quad\ ::\ SRed_2(s, t/l_{mj}, l_{mn}, setc)\ /\ SRed_{mj}(s, t/l_{mj}, setc)) +\\
\qquad\quad (s(\overline{car}), t(t_f), l_{mj}(R), setc(st)/l_{mn}(A))\\
\qquad\quad\ ::\ SRed_2(s, t/l_{mj}, l_{mn}, setc)\ /\ SRed_{mj}(s, t/l_{mj}, setc))
\end{array}
$$

$$
\begin{array}{l}
\textsf{process}\ (SRed_2(s\colon\ sensor, t\colon\ intervs/l_{mj}, l_{mn}\colon\ colour, setc\colon\ stcount)\ /\\
\qquad TLC_{mj}(s\colon\ sensor, t\colon\ intervs/l_{mj}\colon\ colour, setc\colon\ stcount)) \triangleq\\
\qquad\quad (s(car), t(\overline{t_s}), l_{mj}(G), setc(st)/l_{mn}(R))\\
\qquad\quad\ ::\ TLC_2(s, t/l_{mj}, l_{mn}, setc)\ /\ TLC_{mj}(s, t/l_{mj}, setc)) +\\
\qquad\quad (s(\overline{car}), t(\overline{t_s}), l_{mj}(G), setc(st)/l_{mn}(R))\\
\qquad\quad\ ::\ TLC_2(s, t/l_{mj}, l_{mn}, setc)\ /\ TLC_{mj}(s, t/l_{mj}, setc)) +\\
\qquad\quad (s(car), t(t_s), l_{mj}(R), setc(\overline{st})/l_{mn}(A))\\
\qquad\quad\ ::\ SRed_2(s, t/l_{mj}, l_{mn}, setc)\ /\ SRed_{mj}(s, t/l_{mj}, setc)) +\\
\qquad\quad (s(\overline{car}), t(t_s), l_{mj}(R), setc(\overline{st})/l_{mn}(A))\\
\qquad\quad\ ::\ SRed_2(s, t/l_{mj}, l_{mn}, setc)\ /\ SRed_{mj}(s, t/l_{mj}, setc))
\end{array}
$$

5.5 Algebraic Manipulation of Interface Processes

We present, in this section, certain laws involving the decomposition and the other EPA combinators. Our purpose is to show how interface processes can be algebraically manipulated rather than presenting a complete set of laws for interface processes.

The constraints to simulate a process by reusing an existing one are as follows:

$$S \overset{cf}{\sqsubseteq} P \ \text{ and } \ \mathcal{D}!_S(P_I).$$

In order to simplify notation and make discussion shorter, without emphasizing that P_I can be created, etc, we assume $S \overset{c}{\sqsubseteq} P$ and $\mathcal{D}!_S(P)$, even knowing that the reusable process could be a channel restricted simulation of S under a functional interface. We assume the constraints above are satisfied whenever the decomposition operator is applied, and only explicitly define them either when they are not clear from the context or when they need to be emphasized.

5.5.1 Decomposition Laws

Interface processes can be combined in different ways to provide bisimilar process terms. The following laws are defined in blocks in which another particular combinator is applied to interface processes, or a special process is treated.

Firstly, decomposition operator is analysed for process terms in which the process Stop appears:

Proposition 5.8 Stop *laws*

1. Stop / Stop $\sim$ Stop
2. Stop / $P \sim$ Stop

These laws are trivial. By Lemma 4.1, Stop is the least upper bound process defined over any sort (Stop $\sqsubseteq P$). As a result, Stop $\overset{c}{\sqsubseteq} P$ for any P. So, in the first law, decomposition operator can be applied because both processes satisfy the required constraints and Stop is clearly the interface between two Stop processes. The second law generalizes the first one in the sense that decomposition of Stop can only result in itself.

In practice, bisimilar processes demand no interface process to be run together with it to simulate the desired process. But, if an interface behaviour is required to run in parallel with this existing bisimilar process and still achieve the desired behaviour, is should have no transformational effect on the existing process. The process that only performs an empty action and evolves to itself (*divergent process*) is indeed a natural interface behaviour necessary to run with any process in order to achieve its own behaviour. In spite of that, considering the definition of the decomposition operator, even when the desired and existing processes are bisimilar (or similar), the interface process constructed from application of such a combinator is one that comprises all the behaviour of the desired process as input behaviour. Let us define a special kind of process that makes all output channels to be input channels, and has exclusively empty actions as output behaviour:

$$In(S) \;\triangleq\; \{In(S) \xrightarrow{i_s \oplus o_s/\epsilon} In(S') \mid S \xrightarrow{i_s/o_s} S'\}$$

Notice that $In(S)$ composed with S can only achieve S's behaviour if $\mathcal{D}!_S(S)$ (this is one of the constraints to decompose processes), while any process S can simulate itself if running concurrently with a divergent process. It is clear that the divergent process is the natural and desired synchronous interface for bisimilar processes. But as a matter of generalization of the decomposition operator, $In(S)$ is achieved as a result.

Suppose $S \sqsubseteq P$. The input process laws are as follows:

Proposition 5.9 $In(S)$ *laws:*

1. $S \;/\; P \sim In(S)$
2. $S \;/\; In(S) \sim S$

These laws are easily checked by definition of the decomposition operator. The second law is obvious since S is the only process that can achieve its own behaviour when running together with $In(S)$. This process has no transformational effect over compound processes because it does not respond to the environment. So, S's behaviour is achieved only if it is run together with its input process in order to make all of its communication with the environment available.

By the first law, decomposition of S with respect to a similar process results in $In(S)$ – it can be checked by application of the decomposition operator. If the desired and the existing processes are similar ($S \sqsubseteq P$), $In(S)$ running together with P has the effect of inhibiting all the P's actions that do not simulate S's: $(P \mid In(S)) \sim S$. So, $In(S)$ is necessary and sufficient to achieve bisimilarity with S.

A few other laws show interface processes having a form of restriction applied, or making combinations with another process by summation, composition or decomposition. We first present laws for channel restriction combinators and then, show several other laws for the other combinators.

Hiding Channels

Restriction to channels and hiding laws are as follows:

Proposition 5.10 *Restriction to Channels and Hiding laws:*

1. $(S \;/\; P) \upharpoonright_c C \sim (S \upharpoonright_c C) \;/\; (P \upharpoonright_c C) \quad$ *if* $\quad C \cap \mathsf{Chs}(S) \cap \mathsf{Chs}(P) \neq \emptyset$
2. $(S \;/\; P) \setminus C \sim (S \setminus C) \;/\; (P \setminus C) \quad$ *if* $\quad (\mathsf{Chs}(S) \cap \mathsf{Chs}(P)) \not\subseteq C$

Consider the first law. Since C comprises coincident channels of S and P and $S \overset{c}{\sqsubseteq} P$, then $(S \upharpoonright_c C) \overset{c}{\sqsubseteq} (P \upharpoonright_c C)$. Assuming these properties, for each action from S, there exists a corresponding action from P: $S \xrightarrow{i_s/o_s} S'$, then $P \xrightarrow{i_p/o_p} P'$ such that $\mathit{res\text{-}ch}(i_s/o_s, \mathsf{Chs}(P) \cap \mathsf{Chs}(S)) = \mathit{res\text{-}ch}(i_p/o_p, \mathsf{Chs}(P) \cap \mathsf{Chs}(S))$ and $S' \overset{c}{\sqsubseteq} P'$.

Decomposition of S with respect to P comprises only actions that interface these corresponding actions:

$$(S \;/\; P) \xrightarrow{i_s/o_s \,\div\, i_p/o_p} (S' \;/\; P').$$

As a consequence, the interface process restricted to C channels performs all $(S \;/\; P)$ actions restricted to C:

$$(S \;/\; P) \upharpoonright_c C \xrightarrow{\mathit{res\text{-}ch}(i_s/o_s \,\div\, i_p/o_p,\, C)} (S' \;/\; P') \upharpoonright_c C.$$

Now, if processes S and P are restricted to channels C, each action from $(S \upharpoonright_c C)$ is related with an action from $(P \upharpoonright_c C)$: $(S \upharpoonright_c C) \xrightarrow{\mathit{res\text{-}ch}(i_s/o_s,\, C)} (S' \upharpoonright_c C)$, then $(P \upharpoonright_c C) \xrightarrow{\mathit{res\text{-}ch}(i_p/o_p,\, C)} (P' \upharpoonright_c C)$ such that $\mathit{res\text{-}ch}(\mathit{res\text{-}ch}(i_s/o_s, C), C) = \mathit{res\text{-}ch}(\mathit{res\text{-}ch}(i_p/o_p, C), C)$ and $(S' \upharpoonright_c C) \overset{c}{\sqsubseteq} (P' \upharpoonright_c C)$. Then, decomposition of these restricted processes has the corresponding action:

$$((S \upharpoonright_c C) \;/\; (P \upharpoonright_c C)) \xrightarrow{\mathit{res\text{-}ch}(i_s/o_s,\, C) \,\div\, \mathit{res\text{-}ch}(i_p/o_p,\, C)} ((S' \upharpoonright_c C) \;/\; (P' \upharpoonright_c C)).$$

This means that for each action $(S \;/\; P) \upharpoonright_c C \xrightarrow{\mathit{res\text{-}ch}(i_s/o_s \,\div\, i_p/o_p,\, C)} (S' \;/\; P') \upharpoonright_c C$, there exists a corresponding action $((S \upharpoonright_c C) \;/\; (P \upharpoonright_c C)) \xrightarrow{\mathit{res\text{-}ch}(i_s/o_s,\, C) \,\div\, \mathit{res\text{-}ch}(i_p/o_p,\, C)} ((S' \upharpoonright_c C) \;/\; (P' \upharpoonright_c C))$ and vice–versa. Since these actions are originated from the same pair of actions from S and P, they must be identical in order to achieve bisimilarity between both processes:

$$\mathit{res\text{-}ch}(i_s/o_s \div i_p/o_p, C) = \mathit{res\text{-}ch}(i_s/o_s, C) \div \mathit{res\text{-}ch}(i_p/o_p, C).$$

These actions are identical indeed, but we defer the proof of this equality to Appendix C in order to avoid unnecessary details in the present discussion.

The second proposition of 5.10 is established in a similar way to the above. It results with the case that for each action

$$(S \;/\; P) \setminus C \xrightarrow{\mathit{hide\text{-}act}(i_s/o_s \,\div\, i_p/o_p,\, C)} (S' \;/\; P') \setminus C,$$

there exists a corresponding action

$$((S \setminus C) \;/\; (P \setminus C)) \xrightarrow{\mathit{hide\text{-}act}(i_s/o_s,\, C) \,\div\, \mathit{hide\text{-}act}(i_p/o_p,\, C)} ((S' \setminus C) \;/\; (P' \setminus C))$$

and vice–versa. The equality of these actions is also presented in Appendix C:

$$hide\text{-}act(i_s/o_s \div i_p/o_p, C) = hide\text{-}act(i_s/o_s, C) \div hide\text{-}act(i_p/o_p, C).$$

The reader should note that the constraints defined over the set of channels C are implicitly defined by the properties these processes must hold to have the decomposition operator applied to both hand sides. So, there is no need to explicitly define these side–conditions.

Summation

Decomposition of a summation process is bisimilar to summation of its decomposed individuals, if they are channel restricted similar to the divisor process:

Proposition 5.11 *Summation laws:*

1. $(S_1 + S_2) \;/\; P \sim (S_1 \;/\; P) + (S_2 \;/\; P) \quad if \quad S_1 \overset{c}{\sqsubseteq} P,\ S_2 \overset{c}{\sqsubseteq} P \ and \ (S_1 + S_2) \overset{c}{\sqsubseteq} P$

Suppose $S_1 \xrightarrow{i_{s1}/o_{s1}} S_1'$, $S_2 \xrightarrow{i_{s2}/o_{s2}} S_2'$, $P \xrightarrow{i_{p1}/o_{p1}} P'$ and $P \xrightarrow{i_{p2}/o_{p2}} P''$ such that $C = (\mathsf{Chs}(S_1) \cap P) = (\mathsf{Chs}(S_2) \cap P)$. Since summation basically works over initial actions, a proof of this law is based on equality of initial actions of both process terms. Initial actions of these summand processes are related to the interface initial actions by one of the following cases:

Case 1 $res\text{-}ch(i_{s1}/o_{s1}, C) = res\text{-}ch(i_{p1}/o_{p1}, C)$, $res\text{-}ch(i_{s2}/o_{s2}, C) = res\text{-}ch(i_{p2}/o_{p2}, C)$, $S_1' \overset{c}{\sqsubseteq} P'$ and $S_2' \overset{c}{\sqsubseteq} P''$. Then, $(S_1 + S_2) \xrightarrow{i_{s1}/o_{s1}} S_1'$ and $(S_1 + S_2) \xrightarrow{i_{s2}/o_{s2}} S_2'$. As a result, the interface process performs the following actions:

$$(S_1 \;/\; P) \xrightarrow{i_{s1}/o_{s1} \div i_{p1}/o_{p1}} (S_1' \;/\; P'), \qquad (S_2 \;/\; P) \xrightarrow{i_{s2}/o_{s2} \div i_{p2}/o_{p2}} (S_2' \;/\; P''), \text{ and}$$
$$(S_1 + S_2) \;/\; P \xrightarrow{i_{s1}/o_{s1} \div i_{p1}/o_{p1}} (S_1' \;/\; P'), \quad (S_1 + S_2) \;/\; P \xrightarrow{i_{s2}/o_{s2} \div i_{p2}/o_{p2}} (S_2' \;/\; P'').$$

Processes $(S_1 + P_2) \;/\; P$ and $(S_1 \;/\; P) + (S_2 \;/\; P)$ perform identical initial actions and reach bisimilar processes as required.

Case 2 $res\text{-}ch(i_{s1}/o_{s1}, C) = res\text{-}ch(i_{p1}/o_{p1}, C) = res\text{-}ch(i_{s2}/o_{s2}, C)$, $S_1' \overset{c}{\sqsubseteq} P'$ and $S_2' \overset{c}{\sqsubseteq} P'$. Then, $(S_1 + S_2) \xrightarrow{i_{s1}/o_{s1}} S_1'$ and $(S_1 + S_2) \xrightarrow{i_{s2}/o_{s2}} S_2'$. As a result, the interface processes perform:

$$(S_1 \;/\; P) \xrightarrow{i_{s1}/o_{s1} \div i_{p1}/o_{p1}} (S_1' \;/\; P'), \qquad (S_2 \;/\; P) \xrightarrow{i_{s2}/o_{s2} \div i_{p1}/o_{p1}} (S_2' \;/\; P'), \text{ and}$$
$$(S_1 + S_2) \;/\; P \xrightarrow{i_{s1}/o_{s1} \div i_{p1}/o_{p1}} (S_1' \;/\; P'), \quad (S_1 + S_2) \;/\; P \xrightarrow{i_{s2}/o_{s2} \div i_{p1}/o_{p1}} (S_2' \;/\; P').$$

In this case, both processes also perform identical sets of initial actions as required.

In both cases, both processes perform identical initial actions and then reach bisimilar processes. These processes are thus bisimilar.

Composition

Composition allows processes to communicate with each other in order to provide more complex elements. On the other hand, decomposition operator aims at "breaking" a process into smaller elements. The decomposition operator is indeed an attempt to make the reverse process of composition. So, the laws involving composition and decomposition operators are very attractive to establish cases where these combinators can enjoy an inverse functionality:

Proposition 5.12 *Composition laws:*

1. $(S \;/\; (P_1 \mid P_2)) \sim ((S \;/\; P_1) \;/\; P_2)$ *if* $\mathsf{Chs}_i(P_1) \cap \mathsf{Chs}_o(P_2) = \emptyset$ *and* $\mathsf{Chs}_i(P_2) \cap \mathsf{Chs}_o(P_1) = \emptyset$
2. $((S_1 \mid S_2) \;/\; P) \mid P \sim (S_1 \;/\; P) \mid S_2 \mid P$ *if* $\mathsf{Chs}_i(S_1) \cap \mathsf{Chs}_i(P) = \mathsf{Chs}_i(S_2) \cap \mathsf{Chs}_i(P)$
3. $((S_1 \mid S_2) \;/\; (P_1 \mid P_2)) \mid (P_1 \mid P_2) \sim$ $(S_1 \;/\; P_1) \mid (S_2 \;/\; P_2) \mid (P_1 \mid P_2)$ *if* $\mathsf{Chs}_i(P_1) \cap \mathsf{Chs}_o(P_2) = \emptyset$ $\mathsf{Chs}_i(P_2) \cap \mathsf{Chs}_o(P_1) = \emptyset$ *and* $\mathsf{Chs}_i(S_1) \cap \mathsf{Chs}_i(P) = \mathsf{Chs}_i(S_2) \cap \mathsf{Chs}_i(P)$

Notice that in some of these laws, bisimulation is only achieved under composition with the corresponding divisor process. This is a straightforward consequence of having some input channels of interface processes that are only created to provide uniqueness of conflict free actions with the divisor process. And then, one way of internalizing these "spare" connections is by composition with the corresponding divisor process. In other words, the decomposition operator does not provide a natural decomposition of processes, but it is very much concerned with the composition operator.

The first law gives the idea of having two processes that can be composed in order to simulate S. By the required constraints, synchronous composition of these two processes simulates a channel restricted S.

Suppose $S \xrightarrow{i_s/o_s} S'$, $P_1 \xrightarrow{i_{p1}/o_{p1}} P_1'$ and $P_2 \xrightarrow{i_{p2}/o_{p2}} P_2'$ such that $S \overset{c}{\sqsubseteq} (P_1 \mid P_2)$, $S \overset{c}{\sqsubseteq} P_1$, $(S \;/\; P_1) \overset{c}{\sqsubseteq} P_2$, $\mathit{res\text{-}ch}(i_s/o_s, (\mathsf{Chs}(S) \cap (\mathsf{Chs}(P_1) \cup \mathsf{Chs}(P_2)))) = \mathit{res\text{-}ch}(i_{p1}/o_{p1}, (\mathsf{Chs}(S) \cap (\mathsf{Chs}(P_1) \cup \mathsf{Chs}(P_2))))$ and $S' \overset{c}{\sqsubseteq} (P_1' \mid P_2')$. As a result, i_{p1}/o_{p1} CF i_{p2}/o_{p2} and for each interface action in $S \;/\; (P_1 \mid P_2)$, there exists a corresponding action in $(S \;/\; P_1) \;/\; P_2$:

$$
\begin{aligned}
&(P_1 \mid P_2) \xrightarrow{i_{p1}/o_{p1} \star i_{p2}/o_{p2}} (P_1' \mid P_2'),\\
&S \;/\; (P_1 \mid P_2) \xrightarrow{i_{s1}/o_{s1} \div (i_{p1}/o_{p1} \star i_{p2}/o_{p2})} S' \;/\; (P_1' \mid P_2')\\
&(S \;/\; P_1) \xrightarrow{i_{s1}/o_{s1} \div i_{p1}/o_{p1}} (S' \;/\; P_1'),\\
&(S \;/\; P_1) \;/\; P_2 \xrightarrow{(i_{s1}/o_{s1} \div i_{p1}/o_{p1}) \div i_{p2}/o_{p2}} (S' \;/\; P_1') \;/\; P_2'
\end{aligned}
$$

To have these processes bisimilar, these corresponding actions must be identical. In fact, they are identical, as proved in Appendix C, because

$$(i_{s1}/o_{s1} \div (i_{p1}/o_{p1} \star i_{p2}/o_{p2})) = ((i_{s1}/o_{s1} \div i_{p1}/o_{p1}) \div i_{p2}/o_{p2}).$$

The sets of initial actions of both processes are identical and they reach processes that have identical initial actions and so on. Thus, these processes are bisimilar as required.

The second law of 5.12 states that interface processes can be constructed using either the whole divisor process or only a restricted behaviour of that because their composition with the divisor process are bisimilar, considering the side–condition.

Suppose $S_1 \xrightarrow{i_{s1}/o_{s1}} S_1'$, $S_2 \xrightarrow{i_{s2}/o_{s2}} S_2'$ and $P \xrightarrow{i_p/o_p} P'$ such that $S_1 \overset{c}{\sqsubseteq} P$, $(S_1 \mid S_2) \overset{c}{\sqsubseteq} P$, $\mathit{res\text{-}ch}(i_{s1}/o_{s1}, (\mathsf{Chs}(S_1) \cap \mathsf{Chs}(P))) = \mathit{res\text{-}ch}(i_p/o_p, (\mathsf{Chs}(S_1) \cap \mathsf{Chs}(P)))$, $S_1' \overset{c}{\sqsubseteq} P'$ and i_{s1}/o_{s1} CF i_{s2}/o_{s2}. From these assumptions, $(i_{s1}/o_{s1} \star i_{s2}/o_{s2})$ CF i_p/o_p is deduced, and

$$(S_1 \mid S_2) \xrightarrow{i_{s1}/o_{s1} \star i_{s2}/o_{s2}} (S_1' \mid S_2') \quad \text{and} \quad (S_1 \;/\; P) \xrightarrow{i_{s1}/o_{s1} \div i_p/o_p} (S_1' \;/\; P').$$

As a consequence,

$$
\begin{aligned}
&(S_1 \mid S_2) \;/\; P \xrightarrow{(i_{s1}/o_{s1} \star i_{s2}/o_{s2}) \div i_p/o_p} (P_1' \mid P_2') \;/\; P' \text{ and}\\
&(S_1 \;/\; P) \mid S_2 \xrightarrow{(i_{s1}/o_{s1} \div i_p/o_p) \star i_{s2}/o_{s2}} (S_1' \;/\; P') \mid S_2'.
\end{aligned}
$$

Since the above actions are only conflict free with i_p/o_p, for each action from $(((S_1 \mid S_2) \ / \ P) \mid P)$ there exists an action from $((S_1 \ / \ P) \mid S_2 \mid P)$ (and vice–versa) such that:

$$((S_1 \mid S_2) \ / \ P) \mid P \xrightarrow{((i_{s1}/o_{s1} \star i_{s2}/o_{s2}) \div i_p/o_p) \star i_p/o_p} ((S_1' \mid S_2') \ / \ P') \mid P' \text{ and}$$
$$((S_1 \ / \ P) \mid S_2) \mid P \xrightarrow{((i_{s1}/o_{s1} \div i_p/o_p) \star i_{s2}/o_{s2}) \star i_p/o_p} ((S_1' \ / \ P') \mid S_2) \mid P'$$

and

$$((i_{s1}/o_{s1} \star i_{s2}/o_{s2}) \div i_p/o_p) \star i_p/o_p = ((i_{s1}/o_{s1} \div i_p/o_p) \star i_{s2}/o_{s2}) \star i_p/o_p.$$

Both processes have identical initial actions and reach bisimilar processes after identical moves, they are then bisimilar as required.

The third law of 5.12 is a consequence of the other two:

$$\begin{array}{rcll} ((S_1 \mid S_2) \ / \ (P_1 \mid P_2)) \mid (P_1 \mid P_2) & \sim & (((S_1 \mid S_2) \ / \ P_1) \ / \ P_2) \mid (P_1 \mid P_2) & \text{by comp law 1} \\ & \sim & (((S_1 \ / \ P_1) \mid S_2) \ / \ P_2) \mid (P_1 \mid P_2) & \text{by comp law 2} \\ & \sim & (S_1 \ / \ P_1) \mid (S_2 \ / \ P_2) \mid (P_1 \mid P_2) & \text{by comp law 2} \end{array}$$

Decomposition

In trying to reuse a process in order to simulate another, we may realize that the interface process necessary for such a simulation is similar to another existing process. We could, for example, reuse process P_1 to simulate S and then realize that the process necessary to interface these processes is similar to an existing process P_2. Then, the interface process comprises only the S's behaviour that is not matched with those existing processes.

On the other hand, if we first realize that process P_2 is similar to S and find the corresponding interface process, we should be able to check similarity of P_1 with this interface process. The following law states bisimilarity between interface processes that are constructed by successive decomposition of a desired process with respect to the same set of divisor processes.

Proposition 5.13 *Decomposition laws:*

1. $(S \ / \ P_1) \ / \ P_2 \sim (S \ / \ P_2) \ / \ P_1$ *if* $S \overset{c}{\sqsubseteq} P_1, S \overset{c}{\sqsubseteq} P_2, \mathsf{Chs}_i(P_1) \cap \mathsf{Chs}_o(P_2) = \emptyset$ *and* $\mathsf{Chs}_i(P_2) \cap \mathsf{Chs}_o(P_1) = \emptyset$

This law is concerned with the first composition law previously defined:

$$\begin{array}{rcll} (S \ / \ P_1) \ / \ P_2 & \sim & S \ / \ (P_1 \mid P_2) & \text{by comp law 1} \\ & \sim & S \ / \ (P_2 \mid P_1) & \text{by commutativity of } \mid \\ & \sim & (S \ / \ P_2) \ / \ P_1 & \text{by comp law 1} \end{array}$$

5.6 Discussion

We have presented in this chapter a solution for the interface equation via application of a decomposition operator which depends on the relation between the desired and given components. As discussed earlier in this chapter, no unique solution exists for such an equation due to the implicit hiding channels of compound processes. So, the decomposition operator presented here is a particular solution for the interface equation, even when an algebraic combinator is required.

In Section 5.3, we analysed the minimal information necessary to ensure a solution for the interface equation. The interface process depends on behavioural similarities between the desired and the given component. For certain relations an interface process can be defined straightforwardly without applying any combinator. For channel restricted similar processes (or processes related by a weaker relation), however, an interface process must be constructed from information of both processes. Although we have suggested what information should be included in the input and output actions of the interface process, certain other pairs of input and output actions could be suggested to solve the same equation. In fact, we have chosen to overconnect the input actions while the output actions have the minimal information. This approach has been adopted in order to facilitate the final minimization (such a technique will be later discussed in Chapter 6).

5.6.1 One Way Substitution and Reuse

A solution for the interface equation has been shown without questioning whether bisimulation is the only relation that ensures substitutivity of reusable components. Broadly speaking, substitutivity of processes is related to the capacity of both processes to replace each other[6]. Reusability, however, is concerned with the idea of having existing processes substituting for others that are actually being developed. So, the substitutivity required for reuse lies in the capacity of an existing process (or a composition with) to substitute for a desired specification. In this particular case, only the existing process must substitute for the desired one, this is what we may call "one–way substitution", where only a unilateral replacement is required.

Bisimulation is recognized as a relation which permits processes to substitute for each other since it is a congruence. But, can simulation also ensure the "one–way substitutivity" required for reuse?

If P is a simulation of S ($S \sqsubseteq P$), P can respond similarly to S for all events that stimulate S. But, how does P behave if the environment produces input events not responded by S? S, obviously, deadlocks for all these unexpected events. P, however, might eventually communicate with the environment through these events or even deadlock as much as S. Moreover, after this exceptional stimulus, P might either continue to behave like S or provide an abnormal behaviour. Hence, the behaviour of P is unpredictable if the environment starts producing unexpected events.

On the other hand, if P is embedded in a perfect environment which only demands stimuli that can be responded by S, P will behave bisimilarly to S forever. Additionally, if attention is paid only to actions that can be performed by S (with no concern about deadlocks), P is a perfect substitute for S. Roughly speaking, the reuse of processes that are simulations of desired specifications depends upon the purpose of systems, the environment in which they are embedded and the abstraction level they are defined.

The behaviour of processes are described (using a process algebra) by successive application of basic operators to more primitive process terms (subterms). Also, reusability is applicable to processes describing the entire system or even a subsystem; a reusable element could be either a process subterm or a process term describing the whole system. When the candidate for reuse represents a subterm, it is summed, composed, etc, with other process subterms to provide the entire behaviour of the system. In order to ensure substitutivity of processes, even when the external environment is perfect, simulation must be preserved under application of any combinator (this guarantees reuse of processes that represent subterms of the entire process). Thus, simulation could ensure "one–way–substitutivity" only if for any processes S and P, such that $S \sqsubseteq P$, and any operator ϕ of a process algebra, $\phi(S) \sqsubseteq \phi(P)$.

Informally, simulation appears to be a relation enough for ensuring reuse of processes if constraints are applied to their environments. However, simulation *is not* preserved by *summation* and *composition* (a discussion on simulation versus substitutivity has been included in Appendix C). Thus, the interface equation cannot be redefined for simulation since, even for "one–way substitution", bisimulation is the only relation which allows for replacing processes.

[6] The idea of substitutivity is well–known in process algebras, see for example [Mil89].

Chapter 6

Minimization of Component Processes

6.1 Introduction

Minimization of finite state processes is concerned with obtaining an "equivalent" process, but constructed with a number of minimal states (transitions) [HU79, CL89].

Interface processes constructed by application of the decomposition operator are not minimal in certain aspects: they are "overconnected" and could also have more than one action to simulate a single action of the desired process. To obtain the minimal processes from an interface process, all of its connections which offer no effective participation in the final behaviour of the compound process (composition of the existing with the interface process) must be cut off. Furthermore, actions from the interface process that are not essential to make the compound process bisimilar to the desired one must also be removed. Since interface processes are essentially interacting elements, our greatest interest lies in minimization of concurrent processes in terms of the number of states and ports of communications.

Several minimization techniques have been developed to exploit the problem of minimization of deterministic automata in isolation [Aea71, HU79, CL89], or as an interacting element composed in a serial fashion [Ung69, KN72, Dev91, RHS91]. However, new techniques must be developed for concurrent processes that are interconnected in an arbitrary fashion.

This chapter is devoted to exploring various forms of minimization which enable component processes to be minimal in ports of communication and as an interacting element. Firstly, minimization of processes preserving bisimulation and simulation is discussed to elucidate the notion of minimal processes as isolated elements. Then, minimization of concurrent processes as interacting elements is further explored. A technique for minimizing ports of communication between component processes is presented, and finally, a discussion on minimization of interface processes is conducted to explain which techniques should be applied to obtain minimal interface processes.

6.2 Minimal Processes

Techniques for minimizing *deterministic finite automata* (which are fully defined) are well–known and algorithmically solved in the literature of "Automata Theory" (algorithms widely accepted can, for example, be found in [Aea71, HU79, CL89]). Minimization of *automata incompletely defined* has actually been treated in the "Switching" [Ung69] and Automata [Gil62] theories.

Minimization of completely defined processes[1] is based on trace equivalence [HU79] (two processes are equivalent if they can perform identical sequences of actions – traces), and then an equivalent process

[1] The term "process" is used instead of "labelled transition system of the process" in order to avoid such a long phrasal noun.

is obtained as a result. For incompletely defined processes, however, minimization under "compatibility of states" is exploited and processes performing a superset of traces of the original one are obtained. Unlikely, these techniques are applicable to deterministic processes while we are dealing with nondeterministic ones.

In this section we present minimization of nondeterministic processes with respect to bisimulation and simulation. Since bisimulation ensures substitutivity, the technique of minimization preserving bisimulation can be freely applied to the interface processes. Minimization preserving simulation, however, is shown as part of the technique for minimizing interacting processes (Section 6.3)[2].

Minimization of Processes Preserving Bisimulation

Minimization of processes to preserve bisimulation is reduced to the problem of finding classes of bisimilar processes. This technique is identical to that one for deterministic processes, except that bisimulation is adopted as the equivalence relation instead of trace equivalence. Refinement of partitions of process states respecting bisimulation has already been discussed in Section 4.6.1. Here, we present the properties necessary to obtain the minimal bisimilar process.

Due to the fact that bisimulation is an equivalence relation, properties of processes are also applicable to *sets of processes*[3]. Given the transition system of a process P, $TS(P) = \{\mathsf{Sts}_P, \mathcal{A}_P, T^P, P\}$[4], the equivalence class of Sts_P under bisimulation which contains P' is as follows:

$$\mathcal{C}_\sim(P') = \{P'' \mid P'' \in \mathsf{Sts}_P \text{ and } P'' \sim P'\}$$

Processes share the same class if they are in Sts_P and are bisimilar. Notice that by the definition above, any process P' is an element of its own equivalence class, and the classes of all elements in $\mathcal{C}_\sim(P')$ equals the P' equivalence class. Sts_P can then be partitioned into a minimal number of equivalence classes because bisimulation is a *right congruence* [CL89]. The states of the minimal process P (denoted as $\mathfrak{M}(P)$) is given by the set of classes of Sts_P under bisimulation:

$$\mathfrak{M}(P) = \{\mathcal{C}_\sim(P') \mid P' \in \mathsf{Sts}_P\}$$

The minimal process is constructed from blocks of this partition in such a way that each block represents a distinct process:

Definition 6.1 *Suppose $\rho = \{B_1^P, ...B_n^P\}$ is a partition of Sts_P. The process constructed from partition ρ is defined by:*

$$\begin{aligned} \lfloor B_k^P \rceil &= \{B_k^P \xrightarrow{\alpha} B_l^P \mid (P_i, P_j) \in T_\alpha^P, \quad P_i \in B_k^P \textit{ and } P_j \in B_l^P\} \\ \lfloor \rho \rceil &= \{\lfloor B_k^P \rceil \mid B_k^P \in \rho\} \end{aligned}$$

The notation is overloaded to define a process for a single block or the entire partition. The context in which they are inserted gives the meaning when the entire partition or only a single block is addressed. The process representing a partition block will be called "joint process" in the present work.

Minimization of Processes Preserving Simulation

The idea of minimizing processes to obtain one which performs a superset of traces of the original process has been addressed as minimization under *compatible states* [Gil62, Ung69]. *Compatibility of states* is a relation defined upon the so–called finite state machines incompletely defined. Here, we define a new relation, *compatibility of processes*, for nondeterministic processes (based on simulation and *compatible states*) in order to obtain minimal processes that are simulations of the original ones.

[2] This technique can be widely applied to minimize processes in general. But, for the interface approach, bisimilar processes are required – simulation is not preserved by composition.

[3] This assertion has been defined for states (automata theory) in [Gil62].

[4] Transition system of processes is presented in Chapter 2, page 24. As an abuse of notation, the process states are named as processes –P has the initial state P– and the term process is often used meaning process state.

Processes that respond identically to identical stimuli are called *compatible processes*. However, some actions performed by one of these processes might not be performed by the other and vice–versa. *Compatibility of processes* is more formally defined as follows:

Definition 6.2 *Process Compatibility (abbreviated by $\simeq$) is a symmetric binary relation on processes such that if $P \simeq Q$, then for every "proper" action $i_1/o_1 \in Act$, such that $i_1 \in (\mathsf{Sort}_{1i}(P) \cap \mathsf{Sort}_{1i}(Q))$:*

1. *whenever $P \xrightarrow{i_1/o_1} P'$ then $Q \xrightarrow{i_1/o_1} Q'$ for some Q' such that $P' \simeq Q'$,*
2. *whenever $Q \xrightarrow{i_1/o_1} Q'$ then $P \xrightarrow{i_1/o_1} P'$ for some P' such that $P' \simeq Q'$.*

Compatible processes must perform actions identically whenever they are defined for identical input actions, but there is no constraint over actions exclusively performed by one of those processes; no comparison is made upon initial actions exclusively performed by P or Q. Besides that, both processes must reach compatible processes when performing identical actions. Clearly, compatibility of processes is *symmetric* and *reflexive*, but *intransitive*.

For minimization of processes preserving simulation, classes of compatible processes will be required. But in contrast to bisimulation where the process states are partitioned into a unique element, partition of the process states induced by compatibility is not unique (it is not an equivalence relation), and each element of partition might depend on the existence of some other partition elements.

Processes are compatible if whenever they respond to identical input events, they do this identically and then reach compatible processes. To obtain classes of processes under compatibility, whenever an identical action is performed by processes from the same class, then they reach processes in an identical class. Classes of compatible processes depend thus on the relation upon their own elements (they must be compatible), and on the existence of some other classes of compatible processes. So, compatibility classes of a process is defined as follows:

Definition 6.3 *Suppose $\mathsf{Sts}_P = \{P_1 \dots P_n\}$ and ρ is a partition of Sts_P. B^P_n is a compatibility class in ρ only if:*

1. *B^P_n is an element of this partition ($B^P_n \in \rho$),*
2. *all processes within this class are compatible to each other ($\forall P_i, P_j \in B^P_n \cdot P_i \simeq P_j$),*
3. *for any P_i and P_j in B^P_n, whenever $P_i \xrightarrow{\alpha} P_k$, $P_j \xrightarrow{\alpha} P_l$, and $P_k \simeq P_l$ then $\{P_k, P_l\} \subseteq B^P_m$ such that B^P_m is a compatibility class in ρ.*

A process that represents ρ is constructed by making each class as a single process ($\lfloor\rho\rceil$). Since each compatibility class simulates all processes embedded in it and ρ exclusively comprise compatibility classes, $\lfloor\rho\rceil$ is indeed a simulation of P.

Proposition 6.1 *Suppose ρ is a partition of Sts_P such that all blocks of ρ represent compatibility classes. Then, $P \sqsubseteq \lfloor\rho\rceil$.*

Since we are dealing with finite states processes, the number of possible partitions of a process is finite. For instance, the number of partitions under compatibility is finite as well. Partitions of P under compatibility can be selected from the set of all partitions: they are partitions which exclusively contain compatibility classes. So, given any process with a finite number of states, we are able to create all partitions under compatibility. Furthermore, minimal compatible partitions are a selection of partitions under compatibility which have a minimal number of elements. Minimal partitions under compatibility are then a set of partitions with a minimal number of compatibility classes.

The discussion above leads to an algorithm to construct the set of minimal partitions of process states under compatibility. However, it would be extremely inefficient to generate all partitions, select those partitions that are under compatibility and then filter the set of partitions which have the minimal number of compatibility classes. A few algorithms suggested by Gill, in [Gil62], and Unger, in [Ung69], use a constructive approach to build minimal finite state machines under compatible states (deterministic automata). These algorithms can be adapted to cases of nondeterministic processes under compatibility of processes.

6.3 Minimization of Interacting Processes

Apart from being minimized in isolation, as discussed in the previous section, a process can still be simplified if its input or output information are not fully used when composed with another process.

This form of minimization has its origin in automata theory for minimizing processes that are serially connected. Essentially, the problem of minimizing interacting machines is characterized by either inhibiting information never used by those machines when they are run together or changing internal information to improve minimization. This problem has already had several solutions as commented in the following section.

The work devoted to minimizing interacting automata is confined to machines composed in a serial fashion. However, the interface processes defined in the present work are composed with the existing ones in an arbitrary fashion[5]. So, a technique to minimize concurrent processes (arbitrarily connected) is necessary to minimize interface processes.

In the following sections, the existing techniques for minimization of interacting machines (serially composed) is presented, followed by the techniques to minimize concurrent processes.

6.3.1 Related Work

This subject has been exploited in automata theory, and the terminology used there is then concerned with such a theory. In order to maintain consistence with the terminology we have employed throughout our work, we shall consider the techniques developed in automata theory using the process algebra terminology[6].

Minimization of interacting processes presented in the literature is restricted to cases where processes are connected in a serial fashion (Figure 6.1 – P_1 is called a *driving* process while P_2 is a *driven* one), and both component processes are deterministic.

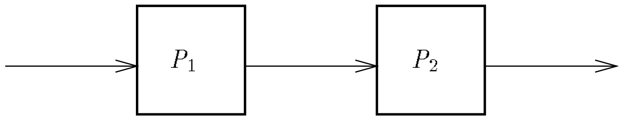

Figure 6.1: Serial Composition of Processes

Basically, composition of P_1 with P_2 hides all information interchanged by these processes (by definition of the composition combinator in automata theory [Shi87]). This means that only the P_1 input channels and the P_2 output channels are observable in the compound process. Since the goal is minimizing these processes without changing behaviour of the compound process, the only information that can be removed or changed is that interchanged between both processes. Even when these processes are minimal in isolation, they can still be changed to obtain a further minimization.

Minimizing Driven Processes

P_2 can only be stimulated by outputs of P_1. This means that if P_1 does not output a specific information which is accepted by P_2, this P_2 input is never stimulated when running together with P_1. This particular input information has no effect on the composition of P_1 with P_2 – this ineffectual information of driven processes is called *input don't care information*. As a result, transitions of P_2 that carry *input don't care information* can be removed from it.

Another kind of don't care information involved in driven processes are given by sequences of inputs of the driven process that are not produced as sequence of outputs by the driving process. This case of don't care information is not as trivial as non–existence of single outputs. Transitions involved in such a

[5] Even knowing that all EPA processes run concurrently, we use the term "serial processes (composition)" meaning processes interconnected in a serial fashion, and "concurrent processes" are those arbitrarily interconnected.

[6] Automata are now processes, transitions are actions, etc.

sequence cannot be freely removed because they might be used to form other sequences that are actually composed with the driving process. This problem has already been solved (algorithmic solutions), as for example in [Ung69, KN72, Dev91, RHS91]. Here, we only show the nature of the problem through an example, without going deeply in algorithmic solutions (the reader is referred to the work cited above for more details).

Suppose processes $Y(c_1, c_2/c_3, c_4)$ and $Z(c_3, c_4/c_5, c_6)$ are as specified in Figure 6.2.

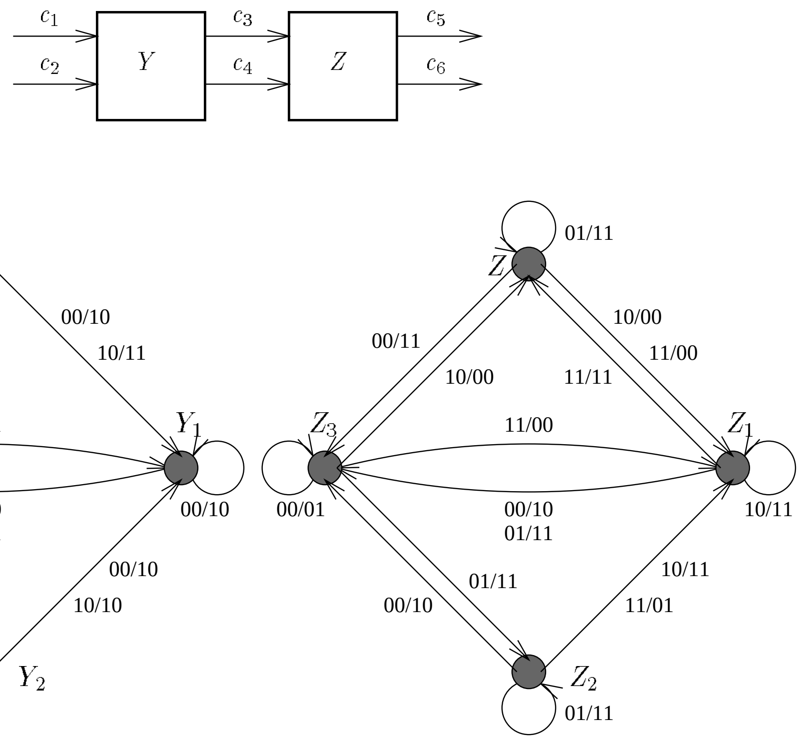

Figure 6.2: Processes $Y(c_1\,c_2/c_3\,c_4)$ and $Z(c_3\,c_4/c_5\,c_6)$

Z is minimal in isolation. Also, all input actions of Z are defined as output actions of Y. So, there is no Z input action that can be removed by only doing static analysis. Even so, certain sequences of inputs in Z are never produced as output sequences of Y. Let us explore some cases where it occurs. In composing Y with Z, the Z's initial action 01/11 cannot be stimulated because there is no output initial action from Y that carries value 01. Since Z is not composed with any Y's subprocess other than Y, this action will never be stimulated in this composition. This specific action can thus be removed from Z.

Continuing with the composition of these processes, process $Y_1 \mid Z_1$ is reached. Since Y_1 is unable to produce 11 as output, 11/11 can then be removed from Z_1 (Z_1 is only composed with Y_1). In Figure 6.3, all the shaded Z actions, shown on the left–hand side, can be removed since they cannot be stimulated when composed with Y. On the right–hand side of the same figure, a minimal process Z obtained by removing those useless actions is given.

Minimization under compatibility is applied to the driven process by the time the input don't care information is removed. Even resulting in a process that accepts a superset of traces (a simulation) of the original process, the compound process is bisimilar to the original one because the application domain of this technique is restricted to *deterministic* processes where all the driving actions can be synchronized with driven ones.

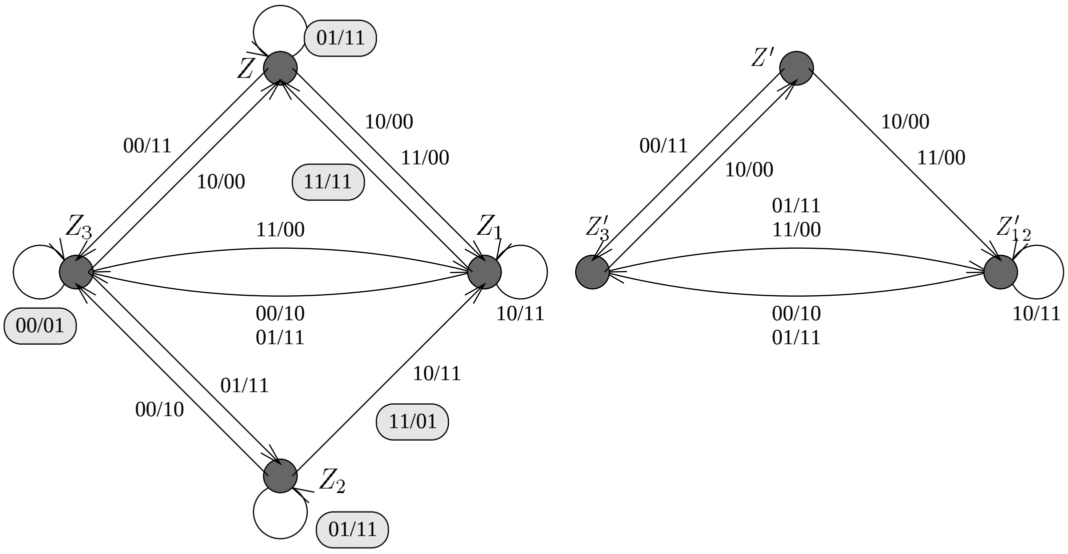

Figure 6.3: Minimal Z using Input Don't Care Information

Minimizing Driving Processes

The idea of constrained controllability of sequential processes can also be exploited to minimize driving processes. Some distinct input actions from the driving process could provide identical output behaviour on the compound process, even when the driving process itself has distinct outputs [Dev91, RHS91].

In the example above (Y and Z), the initial actions 00/10 and 10/11 from Y are composed with distinct actions from Z to provide actions 00/00 and 10/00 in the compound process ($Y \mid Z$) (this process is defined over channels $c_1 c_2 / c_5 c_6$). Note that these compound actions have identical output; in turn, those Y's actions have identical effect on the compound process. Since their output effect are identical, they could interchange their input actions and still obtain an identical compound process. These are the so–called *output don't care information* that can be exploited to obtain a further minimization of the driving process.

In the specific case mentioned above, 00/10 from Y is combined with 10/00 from Z, and 10/11 is combined with 11/00 to generate, respectively, actions 00/00 and 10/00 in the compound process. But, if the first Y's action has 11 as output instead of 10, then 00/11 can be composed with 11/00 from Z and then generate 00/00 in the compound process. Note that 00/11 has the same effect as 00/10 upon the observable behaviour of the compound process (it is also applicable to the other action). Exploiting this idea of multi–valued outputs for driving processes, minimization can be achieved by choosing one of those alternative values.

Figure 6.4 shows, on the left–hand side, the multi–valued Y. An appropriate choice for a further minimization of such process is given by removing the shaded actions. Once we have chosen the appropriate actions, the new process can be minimized in the process shown on the right–hand side of the same figure. Again, minimization under compatibility is employed to obtain a minimal process.

6.3.2 Exploiting Don't Care Information of Component Processes

The work presented thus far is restricted to cases of deterministic "serial processes". However, we are interested in deterministic as much as nondeterministic concurrent processes (arbitrarily interconnected). For concurrent processes, there is no process which is exclusively driving or driven; both of them are driving and driven processes at the same time. Minimization of each concurrent process can then exploit input and output don't care information.

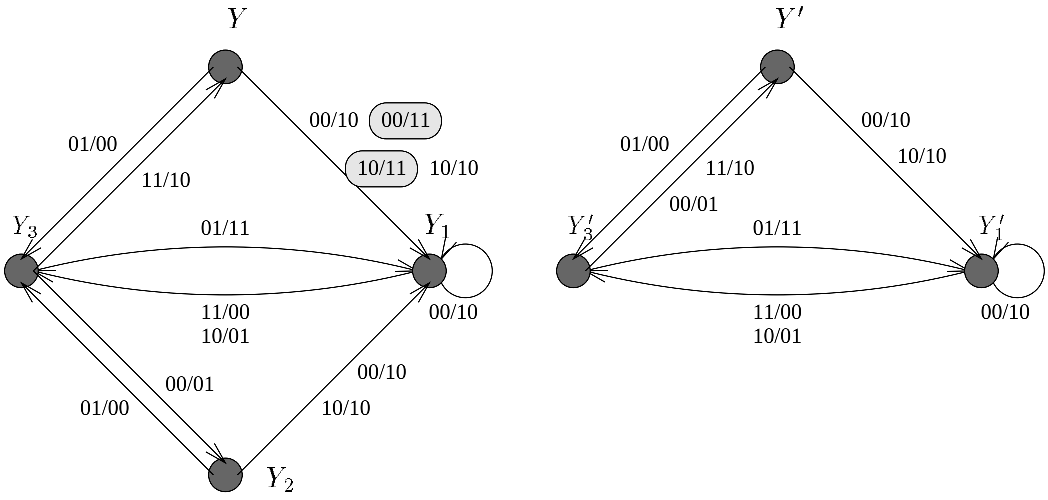

Figure 6.4: Minimal Y using Output Don't Care Information

Looking at the information that can be changed or removed from the driving or driven processes in the techniques presented above, it can be noticed that it is restricted to the information hidden in the compound process. In those techniques, the driven process had some of its actions dropped because they were unreachable. The way to discover these useless actions is by looking at the internal connections of both processes. Sequences of outputs that the driving process is unable to produce are exactly the input don't care sequences of the driven process. This means that only internal information is used to provide minimization of the driven process. Also, the information changed in the driving process to obtain its minimization focuses on the internal information that is passed to the driven process. Communications interchanged between the external environment and the compound process are unchangeable.

For processes composed in a serial fashion, hidden communications correspond to the input action of the driven process and output action of the driving one. The hidden information of concurrent processes is, in turn, given by a restricted set of channels that are not broadcast in the compound process. Thus, the information that can be changed in concurrent processes is constrained to that one hidden by application of the composition or hiding operators.

In the following sections, a technique to minimize concurrent processes by exploiting the don't care information will be presented.

Exploiting Conflicting Information

Composition of actions provides a "proper" (differing from $\boxtimes$) compound action only if they are conflict free. If an action conflicts with all actions from the other component process, it has no effect on the final behaviour of the compound process. Like cases of serial composition, actions that are not stimulated when composed with the other process can be removed with no loss on the observational behaviour of the compound process. So, a process concurrently running with another can have all its conflicting actions removed[7].

Suppose processes $P(c_1, c_2, c_5/c_4, c_6, c_7)$ and $Q(c_2, c_3, c_4, c_7/c_5, c_8, c_9)$ are composed as shown in Figure 6.5. The behaviours of these processes are shown in Figure 6.6.

Q is the target process to be minimized (we refer to Q as a target process and P is the "other component process"). These processes interact with each other through channels c_2, c_4, c_5 and c_7. By making a static analysis on these process, it can be realized that action 0001/001 from process Q_3 conflicts with all actions in P's sort. It can then be removed without changing the behaviour of $(P \mid Q) \setminus \{c_4, c_5\}$.

[7]Making analogy with the previous serial composition, *input don't care information* corresponds to the conflicting information.

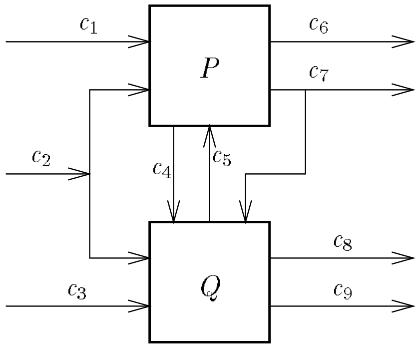

Figure 6.5: Composition of $P(c_1, c_2, c_5/c_4, c_6, c_7)$ with $Q(c_2, c_3, c_4, c_7/c_5, c_8, c_9)$

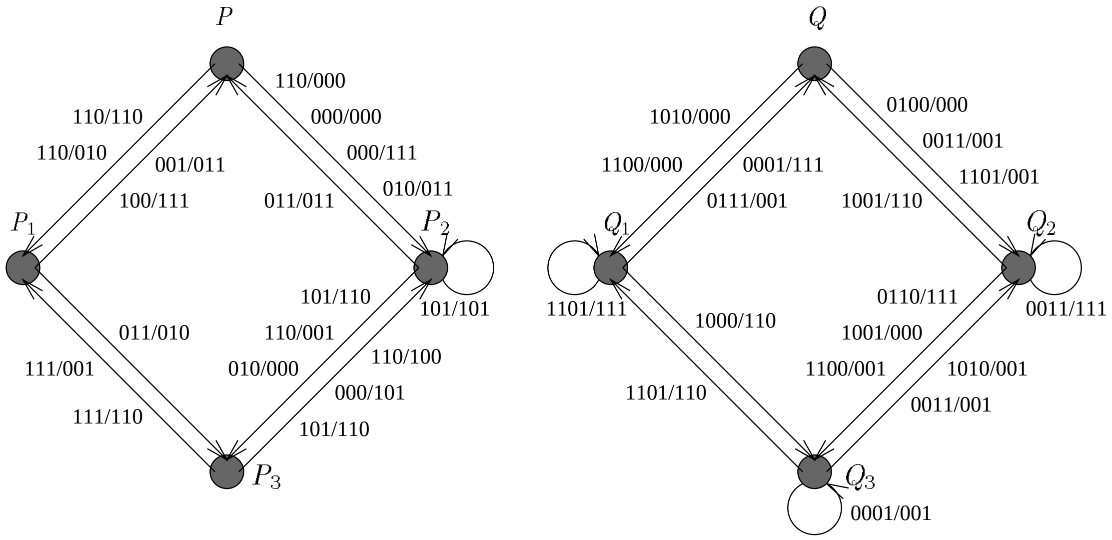

Figure 6.6: Processes $P(c_1\, c_2\, c_5/c_4\, c_6\, c_7)$ and $Q(c_2\, c_3\, c_4\, c_7/c_5\, c_8\, c_9)$

Also, certain sequences of actions of Q could also be unreachable under composition with P. So, actions that are exclusively elements of such sequences can also be removed. Composition of P with Q reaches only processes $(P \mid Q)$, $(P_1 \mid Q_1)$, $(P_2 \mid Q_2)$ and $(P_3 \mid Q_3)$. Despite action $Q_1 \xrightarrow{1101/111} Q_1$ being conflict free with a couple of actions in P's sort (011/010 and 111/000), it conflicts with all actions from P_1. It can then be dropped without affecting the behaviour of the compound process. Figure 6.7 shows the actions from Q (shaded actions) that are not stimulated under composition with P.

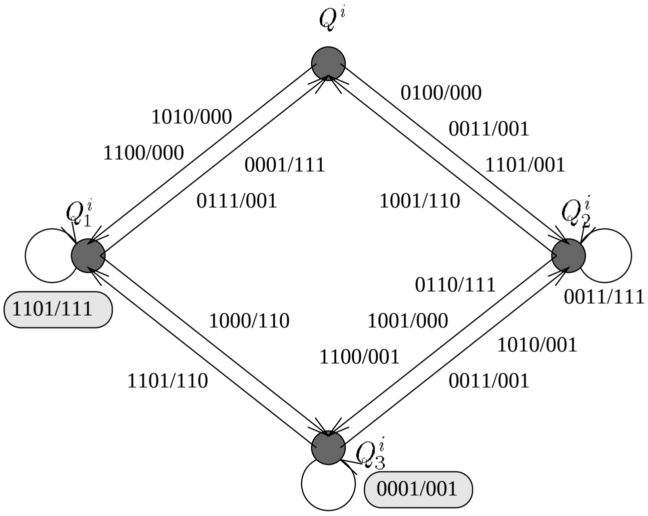

Figure 6.7: Useless $Q(c_2, c_3, c_4, c_7/c_5, c_8, c_9)$ Actions

Once all the useless actions are removed from the target process, it can then be minimized in the number of states. Minimization of target processes under compatibility has been discussed in Section 6.2 and it has been applied to reduce processes in a serial composition. However, this minimization method is inappropriate to cases of concurrent processes because minimization under compatibility results in a process similar, instead of bisimilar, to the original. A technique to minimize concurrent processes preserving bisimilarity with the compound process is presented in Section 6.3.3. We thus defer minimization of the example above to there.

Exploitation of conflicting actions to provide minimization of processes can be applied to both processes; both are driving and driven processes at the same time. But we confine ourselves to minimizing only the target process.

Exploiting Hiding Information

Similar to the serial composition, concurrent processes can be composed in such a way that distinct compound actions have identical output effect. So, the hiding information can be exploited to achieve better minimization of the "driving" process[8].

First of all, let us define what identical output effect means for concurrent processes. In the parallel composition, both processes can communicate with the environment through input and output channels. The "output effect" of the compound process is then given by all the output information from both component processes that communicate with the environment.

To exploit the hiding information, actions from the target process that can be multi–valued to obtain a further minimization must be found. The inherent property which enables to find multi–valued target actions is that composition of these alternative actions with actions from the other process shall produce identical compound actions.

[8]Making analogy with the serial composition, the *output don't care information* there corresponds to the *hiding information* exploited over concurrent processes here.

To make use of identical outputs to improve minimization of interacting processes, distinct input actions from the target process that generate identical compound output actions must be identified. To obtain identical output compound actions, there must exist at least two actions from each component process that are identical in the subset of observable channels.

The input information of both component processes can be communicated with the environment and between those processes. Considering that multi–valued actions must be combined to form identical compound actions, and the other component process is inviolable under minimization of the target process, external input information of the other component process must be identical in those actions that are going to be combined with those multi–valued actions.

Summarizing, multi–valuation of target actions depends on the existence of more than one action from the other component process which has identical observable behaviour[9], and equality of observable output actions of the target process.

Let us use the example given in the previous section to check if process Q can also be minimized by making use of hiding information. In composing P with Q, as shown in Figure 6.5, only channels c_1, c_2, c_3, c_6, c_7, c_8 and c_9 are observable. So, to obtain multi–valued Q actions, at least two actions from Q must be identical in their observable output behaviour (channels c_8 and c_9), and the corresponding conflict free actions from P must also have identical observable behaviour (channels c_1, c_2, c_6 and c_7).

Actions[10]

$$\alpha = c_2(1), c_3(0), c_4(1), c_7(0)/c_5(0), c_8(0), c_9(0) \text{ and}$$
$$\beta = c_2(1), c_3(1), c_4(0), c_7(0)/c_5(0), c_8(0), c_9(0),$$

from Q ($Q \xrightarrow{\alpha} Q_1, Q \xrightarrow{\beta} Q_1$), for example, are identical in their observable output behaviour. Could these actions be multi–valued to enhance minimization? To obtain such a multi–valuation, actions from P that are conflict free with them must be identical in their observable behaviour. Actions α and β are, respectively, conflict free with the P actions ($P \xrightarrow{\gamma} P_1, P \xrightarrow{\omega} P_1$)

$$\gamma = c_1(1), c_2(1), c_5(0)/c_4(1), c_6(1), c_7(0) \text{ and}$$
$$\omega = c_1(1), c_2(1), c_5(0)/c_4(0), c_6(1), c_7(0).$$

As it can be observed, these P actions are identical in their observable behaviour: channels c_1, c_2, c_6 and c_7. Besides that, the compound actions $\alpha \star \gamma$ and $\beta \star \omega$ move to an identical compound process $(P_1 \mid Q_1)$. Hence, multi–valued actions can be created for the target process to obtain the same effect on these compound actions:

$$\alpha' = c_2(1), c_3(0), c_4(0), c_7(0)/c_5(0), c_8(0), c_9(0)$$
$$\beta' = c_2(1), c_3(1), c_4(1), c_7(0)/c_5(0), c_8(0), c_9(0).$$

Notice that the alternative actions given to each original Q action (α' is an alternative for α and β' is an alternative for β) maintain the observable behaviour of the original action, but the internal connections are changed to be conflict free with the other action from P. For instance, α' is identical to α in the output and interchanging observable behaviour –c_2, c_7, c_8 and c_9– but its internal connections (c_4 and c_5) differ in order to be able to communicate with action ω from P. The fact that the observable behaviour of the corresponding P actions are identical implies that the observable information of Q that coincide with P's are identical; channels c_2 and c_7 of α' and α must carry identical values because these channels are elements of the observable behaviour of P.

Using these multi–valued actions, the behaviour of action $\textit{hide-act}(\alpha \star \gamma, \{c_4, c_5\})$ can alternatively be obtained by composition of α' with ω because $\textit{hide-act}(\alpha \star \gamma, \{c_4, c_5\}) = \textit{hide-act}(\alpha' \star \omega, \{c_4, c_5\})$. The multi–valued Q, Q^o, is shown in Figure 6.8. To obtain the compound action, only one of the alternative

[9] Observable behaviour means: channels from both process that are not hidden in the compound process.

[10] The channel and action names are introduced to the representation of these actions (in spite of differing from the graphical representation) to facilitate the discussion of "conflict free" actions.

actions is necessary, and then the other alternatives can be discarded. In doing so, minimization of the multi–valued process can be achieved by taking advantage of the flexibility of choosing alternative actions.

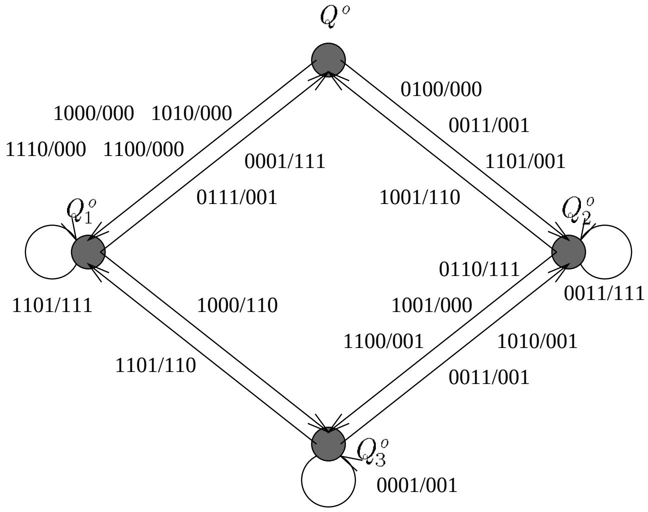

Figure 6.8: Q with Multi–Valued Actions ($Q^o(c_2, c_3, c_4, c_7/c_5, c_8, c_9)$)

Once more, minimization of this multi–valued process is deferred to the next section where minimization of concurrent processes preserving bisimilarity is presented.

6.3.3 Minimizing Concurrent Processes Preserving Bisimulation

A solution for the interface equation requires bisimilarity with the desired process. So, this section is devoted to minimization of target concurrent processes in such a way that the final compound processes are bisimilar to their original ones. This problem is stated as: given two processes running concurrently (arbitrarily interconnected), one of them (called target process) is desired to be minimized in such a way that composition of this minimal process with the other component is bisimilar to the original compound process. To facilitate the discussion, we assume P_0 ($TS(P_0) = \{\mathsf{Sts}_{P_0}, \mathcal{A}_{P_0}, T^{P_0}, P_0\}$) and Q_0 ($TS(P_0) = \{\mathsf{Sts}_{Q_0}, \mathcal{A}_{Q_0}, T^{Q_0}, Q_0\}$) are run in parallel and restricted to the set of observable channels E_x ($(P_0 \mid Q_0) \restriction_c E_x$), and Q_0 is the target process to be minimized.

To maintain bisimilarity with the original compound process, the target process can be minimized preserving bisimulation (Section 6.2), and then the new compound process is bisimilar to the original one – bisimulation is preserved by composition. However, this minimization technique is very restrictive and could rarely obtain a successful result when the don't care information is exploited.

The technique for minimizing processes preserving simulation (Section 6.2) has been applied to the minimization of interacting processes composed in a serial fashion, but it is not applicable to concurrent processes because composition does not even preserve simulation.

For concurrent processes, the key point of minimization is "joining" states of the target process and still maintaining bisimilarity with the original compound process. To do so, composition of each subprocess reachable from P_0, say P_i, with a Q_0's joint process, say $\lfloor Q_l, Q_m \rceil$, must exclusively result in the behaviour of $((P_i \mid Q_l) \restriction_c E_x)$ or $((P_i \mid Q_m) \restriction_c E_x)$; $(P_i \mid \lfloor Q_l, Q_m \rceil) \restriction_c E_x \sim (P_i \mid Q_m) \restriction_c E_x$, or $(P_i \mid \lfloor Q_l, Q_m \rceil) \restriction_c E_x \sim (P_i \mid Q_l) \restriction_c E_x$. Since actions of the compound process are obtained by composition of conflict free actions from the component processes, P_i actions must be exclusively conflict free with Q_l or Q_m actions. To obtain such a property, Q_m and Q_l restricted to the set of channels that interact with P ($C = \mathsf{Chs}(P) \cap \mathsf{Chs}(Q_m)$) must only have either actions that conflict with each other, or behave identically (in the observational behaviour) for conflict free actions. Compatibility of concurrent processes is more formally defined as follows:

Definition 6.4 *Compatibility of Concurrent Processes (abbreviated by $\simeq_{\shortparallel}$) is a symmetric binary relation on processes such that if $Q_i \simeq_{\shortparallel} Q_j$, considering the sets of interacting channels C and external channels E_x, then for every "proper" action $\alpha, \beta \in Act$ such that $\alpha \in \mathsf{Sort}_1(Q_i)$, $\beta \in \mathsf{Sort}_1(Q_j)$ and res-ch$(\alpha, C) =$ res-ch(β, C):*

1. *whenever $Q_i \xrightarrow{\alpha} Q_i'$ then $Q_j \xrightarrow{\beta} Q_j'$ for some Q_j' such that res-ch$(\alpha, E_x) =$ res-ch(β, E_x) and $Q_i' \simeq_{\shortparallel} Q_j'$,*
2. *whenever $Q_j \xrightarrow{\beta} Q_j'$ then $Q_i \xrightarrow{\alpha} Q_i'$ for some Q_i' such that res-ch$(\alpha, E_x) =$ res-ch(β, E_x) and $Q_i' \simeq_{\shortparallel} Q_j'$.*

Concurrently compatible processes perform identical observable actions whenever their actions coincide in a restricted set of channels C, and then reach concurrently compatible processes.

Suppose Q_m and Q_l are concurrently compatible considering the set of channels C ($C = \mathsf{Chs}(P_i) \cap \mathsf{Chs}(Q_l) = \mathsf{Chs}(P_i) \cap \mathsf{Chs}(Q_m)$)[11]. Then, a single action from P_i can communicate with $\lfloor Q_m, Q_l \rceil$ in one of the following ways:

1. it communicates only with Q_l;
2. it communicates only with Q_m; or
3. it communicates with Q_l and Q_m to produce identical compound actions.

When P_i can communicate with both processes, actions from both processes which communicate with this specific P_i action are identical in the restricted sets of channels C and E_x, and then reach concurrently compatible processes (Definition 6.4). As a result, their resulting compound actions are indistinguishable. In the other two cases, P_i actions are conflict free only with actions from one of those processes. Thus, composition of P_i with $\lfloor Q_m, Q_l \rceil$ exclusively behaves like $(P_i \mid Q_m)$ or $(P_i \mid Q_l)$.

To illustrate such a relation, let us use the target process Q which was simplified by exploitation of conflicting information in the previous section (Q^i), and check which pairs of Q^i's states are concurrently compatible. The set of channels interchanged with P (Figure 6.9) is $C = \{c_2, c_4, c_5, c_7\}$ while the observable channels are $Ex = \{c_2, c_3, c_8, c_9\}$. So, the Q^i subprocesses must be concurrently compatible considering these sets of channels. Figure 6.10 shows the actions from Q^i states that make them concurrently incompatible.

Actions $c_2(0), c_3(1), c_4(1), c_7(1)/c_5(0), c_8(0), c_9(1)$ from Q_1^i and $c_2(0), c_3(0), c_4(1), c_7(1)/c_5(0), c_8(0), c_9(1)$ from Q^i, for example, make these processes concurrently incompatible: they carry identical values on channels C ($\{c_2, c_4, c_5, c_7\}$) but their observational behaviour are distinct (channel c_3 is external, but carries distinct values on both actions). This means that if these two processes were joined, this joint process could communicate with the same action from P producing distinct compound actions; action $c_1(0), c_2(0), c_5(0)/c_4(1), c_6(1), c_7(1)$ from P could be composed with $c_2(0), c_3(1), c_4(1), c_7(1)/c_5(0), c_8(0), c_9(1)$ and $c_2(0), c_3(0), c_4(1), c_7(1)/c_5(0), c_8(0), c_9(1)$, from Q_1^i and Q^i respectively, producing different results. A few other states are also incompatible, as shown in Figure 6.10. Only processes Q_1^i with Q_2^i and Q_2^i with Q_3^i are concurrently compatible considering the sets of channels C and E_x. But, is this relation enough to partition states of Q^i and ensure bisimilarity with $(P \mid Q^i) \setminus \{c_4, c_5\}$?

The other component process might have some actions never combined with actions from the target process. For instance, action $c_1(1), c_2(1), c_5(0)/c_4(0), c_6(0), c_7(0)$ from P conflicts with all Q^i initial actions. This means that P cannot move through this action when running together with Q^i. Another example of an unused P action is found when P_3 is composed with Q_3^i: action $c_1(0), c_2(0), c_5(1)/c_4(1), c_6(1), c_7(0)$. Since Q^i only synchronizes with P, any state to be joined to Q^i should not be able to communicate with the P actions that are unable to communicate with Q^i, otherwise composition of P with the Q^i joint process would achieve a process non–bisimilar to $(P \mid Q^i) \setminus \{c_4, c_5\}$.

In the example presented above, we have shown that Q_2^i and Q_3^i are concurrently compatible. Looking for reachable processes from $(P \mid Q^i) \setminus \{c_4, c_5\}$, we obtain:

[11] Q_l and Q_m are defined over the same set of channels.

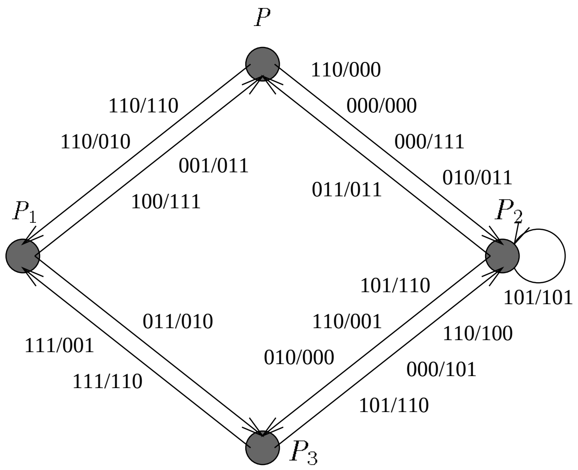

Figure 6.9: Process $P(c_1, c_2, c_5/c_4, c_6, c_7)$

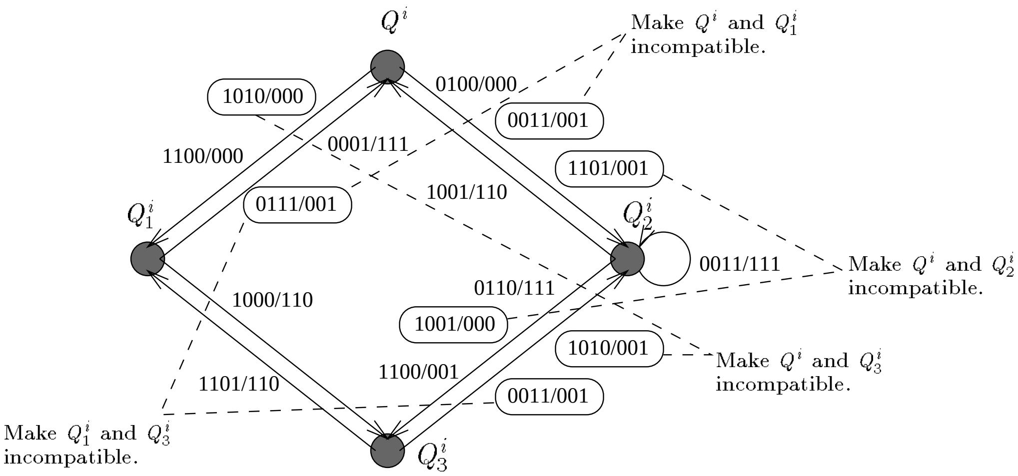

Figure 6.10: Incompatible Processes of $Q^i(c_2, c_3, c_4, c_7/c_5, c_8, c_9)$

$(P \mid Q^i) \setminus \{c_4, c_5\}$
$(P_1 \mid Q_1^i) \setminus \{c_4, c_5\}$
$(P_2 \mid Q_2^i) \setminus \{c_4, c_5\}$ and$(P_3 \mid Q_3^i) \setminus \{c_4, c_5\}$.

To make Q_2^i with Q_3^i a joint process and still maintain bisimilarity with the original $(P \mid Q^i) \setminus \{c_4, c_5\}$, $(P_2 \mid \lfloor Q_2^i, Q_3^i \rceil) \setminus \{c_4, c_5\}$ must be bisimilar to $(P_2 \mid Q_2^i) \setminus \{c_4, c_5\}$ and $(P_3 \mid \lfloor Q_2^i, Q_3^i \rceil) \setminus \{c_4, c_5\}$ must be bisimilar to $(P_3 \mid Q_3^i) \setminus \{c_4, c_5\}$. But, composition of P_3 with $\lfloor Q_2^i, Q_3^i \rceil$ enables P_3 to communicate with all the Q_3^i actions and also with the Q_2^i action $c_1(1), c_2(1), c_5(1)/c_4(1), c_6(1), c_7(0)$ (using the P_3's action which is unable to communicate with Q_3). As a result,

$$(P_3 \mid \lfloor Q_2^i, Q_3^i \rceil) \setminus \{c_4, c_5\} \not\sim (P_3 \mid Q_3^i) \setminus \{c_4, c_5\}.$$

Thus, partition of Q^i under concurrent compatibility is insufficient to ensure bisimilarity with the original compound process. In addition, the partition elements of the target process must be exclusively combined with one subprocess of the other component process. To some extent, partitions of a process Q_0 under composition with P_0 is defined as follows:

Definition 6.5 *Suppose processes P_0 ($TS(P_0) = \{\mathsf{Sts}_{P_0}, \mathcal{A}_{P_0}, T^{P_0}, P_0\}$) and Q_0 ($TS(Q_0) = \{\mathsf{Sts}_{Q_0}, \mathcal{A}_{Q_0}, T^{Q_0}, Q_0\}$) are composed as $(P_0 \mid Q_0) \upharpoonright_c E_x$, and their interchanging channels are $C = \mathsf{Chs}(P_0) \cap \mathsf{Chs}(Q_0)$. $\rho = \{B_1^{Q_0}, \ldots, B_n^{Q_0}\}$ is a partition of Sts_{Q_0} under communication with P_0 and observable behaviour E_x only if ($1 \leq k \leq n$, $Q_i, Q_j \in \mathsf{Sts}_{Q_0}$):*

1. *for all Q_i, Q_j in $B_k^{Q_0}$, $Q_i \simeq_{\shortparallel} Q_j$,*
2. *for any Q_i, Q_j in $B_k^{Q_0}$, whenever $Q_i \xrightarrow{\alpha} Q_i'$, $Q_j \xrightarrow{\beta} Q_j'$, $res\text{-}ch(\alpha, C) = res\text{-}ch(\beta, C)$ and $Q_i' \simeq_{\shortparallel} Q_j'$, then $\{Q_i', Q_j'\} \subseteq B_l^{Q_0}$ such that $B_l^{Q_0} \in \rho$, and*
3. *for any Q_i, Q_j in $B_k^{Q_0}$, if $(P_i \mid Q_i)$ is reachable from $(P_0 \mid Q_0)$, then $\mathsf{Sort}_1(P_i \upharpoonright_c C) \cap \mathsf{Sort}_1(Q_j \upharpoonright_c C) \subseteq \mathsf{Sort}_1(Q_i \upharpoonright_c C)$.*

Since concurrent compatibility is not an equivalence relation (it is intransitive), various partitions of a process states can be obtained under such relation. Also, each class of concurrently compatible processes depends on the existence of some other classes, it is very much concerned with the context in which it is embedded.

The third property constrains the partitioning of Q_0 with respect to its interaction with P_0. Whenever a $(P_i \mid Q_i)$ is reachable from $(P_0 \mid Q_0)$, Q_i can only be joined to another process if P_i can exclusively communicate with Q_i actions when it is composed with such a joint process: the P_i initial actions are conflict free only with Q_i initial actions. This fact ensures that if Q_i is an element of $B_k^{Q_0}$, then $(P_i \mid Q_i) \sim (P_i \mid \lfloor B_k^{Q_0} \rceil)$.

Since the set of Q_0 states is finite, the set of its possible partitions is finite as well. Then, the set of Q_0 partitions under composition with P_0 can be selected from the set of all partitions. Afterwards, the set of minimal partitions can be selected from the set of partitions of Sts_{Q_0} under composition with P_0.

In the example above, we have checked that Q_2^i with Q_3^i and Q_2^i with Q_1^i are concurrently compatible, but processes Q_2^i and Q_3^i cannot be joined because they violate the third property. So, is $\rho_1 = \{\{Q^i\}, \{Q_1^i, Q_2^i\}, \{Q_3^i\}\}$ a partition of Q^i states under composition with P? Since Q_2^i and Q_1^i do not coincide in actions restricted to the set of P and Q^i coincident channels, the second property of Definition 6.5 holds. Also, all actions from P_1 and P_2 can exclusively communicate with Q_1^i or Q_2^i: the third property then holds. ρ_1 is indeed a partition of Sts_Q under composition with P.

The other example that has been deferred to this section is minimization of Q with multi–valued actions, Q^o. Figure 6.11 shows the multi–valued Q with its concurrently incompatible actions.

Choosing actions 1000/000 and 1100/000 as alternatives, Q^o and Q_3^o become concurrently compatible. So, $\rho_2 = \{\{Q^o, Q_3^o\}, \{Q_1^o\}, \{Q_2^o\}\}$ is a partition of the multi–valued Q under concurrently compatibility with P (ρ_2 holds all properties in Definition 6.5).

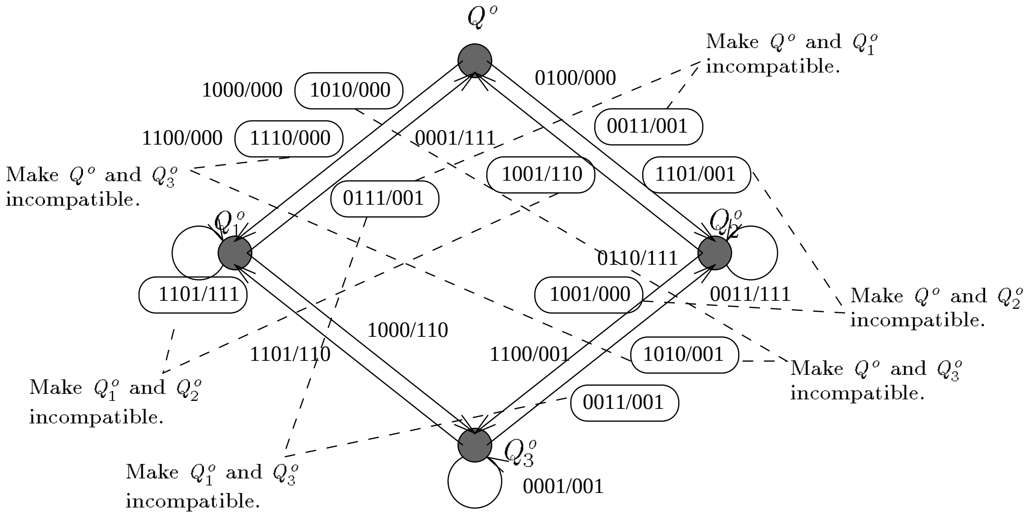

Figure 6.11: Incompatible Processes of the Multi–Valued Q (Q^o)

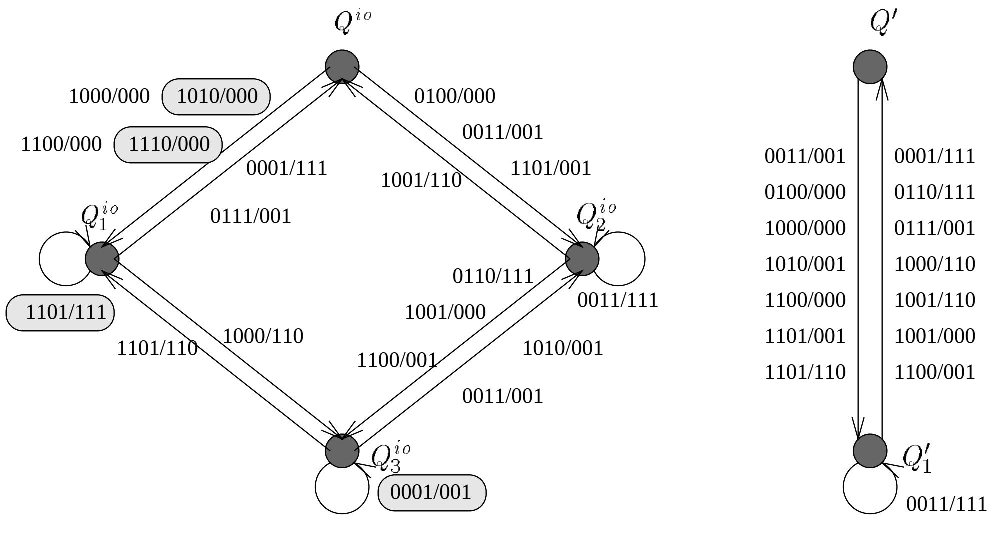

Figure 6.12: A Minimal Concurrently Compatible Q

However, to minimize process Q under composition with P, conflicting and hiding information can be exploited. Figure 6.12 shows, on the left–hand side, Q making use of conflicting and hiding information (Q^{io}), and on the right–hand side, the final minimization of such an interacting process.

In this particular case, Q has a unique minimal process because only the pairs of states Q^{io} with Q_3^{io}, and Q_1^{io} with Q_2^{io} can be joined. To some extent, minimization of interacting processes results in a set of minimal processes because concurrent compatibility is reflexive, symmetric but intransitive.

6.4 Minimization of Processes Communication

Apart from minimizing the number of states, ports of communication of interacting processes must also be as minimized. All information used to represent components has its expenses to be designed. The information not effectively used in a process represents a waste of resource throughout the design and final implementation of components. Thus, the communications that do not affect the behaviour of processes must be removed.

In this section, minimization of communications is presented. A discussion on the non–removable communication and elimination of duplicate connections is conducted, followed by a presentation of algorithms to eliminate unnecessary communications.

Non–Removable Information – Eliminating Duplicates

First, the non–removable communications of concurrent processes are analysed. The behaviour of a compound process is determined by its external communications: what the environment gives as a stimulus and what it is expecting as a response. All communications with the environment are then essential to maintain bisimilarity with the original compound process. Also, the internal connections necessary to run both processes in parallel are essential to maintain bisimilarity with the compound process.

A priori, all the observable and internal information are non–removable elements but some of them could be *duplicates* and, consequently inessential information. The component processes could, for example, communicate with the environment through identical channels: when processes either receive the same stimuli from the environment, or give identical responses. Both cases are forms of duplicates and only one of those processes is necessary to communicate with the environment to maintain bisimilarity with the original compound process.

Another form of duplicate information arise with internal connections that always require values identical to the output connections of the target process. If so, this connection can be eliminated as far as all the ports wired to the eliminated connection are renamed to the non–removed output channel name. This analysis can be done by checking all the internal outputs of the target process that are duplicates.

Considering Q_0 as a target process composed with P_0, and E_x is the set of external communication $((P_0 \mid Q_0) \upharpoonright_c E_x)$, Q_0 is free of duplicate outputs if it holds the following property:

$$\begin{aligned} Out!(Q_0) = {} & (\mathsf{Chs}_o(Q_0) - E_x) \subseteq \mathsf{Chs}_i(P_0) \wedge \\ & \forall c_1 \in (\mathsf{Chs}_o(Q_0) - E_x), c_2 \in \mathsf{Chs}_o(Q_0) \cdot \\ & \exists \alpha \in \mathsf{Sort}(Q_0) \cdot \mathit{act\text{-}value}(\alpha, c_1) \neq \mathit{act\text{-}value}(\alpha, c_2) \end{aligned}$$

Thus, only connections which singularly communicate with the environment or are required to run the other process, are essential to the target process ($\mathit{act\text{-}value}(\alpha, c_1)$ results in the action passed through channel c_1 in action α). The set of non–removable channels of a target process Q_0 with no duplicate outputs ($Out!(Q_0)$ holds) is as follows:

$$\mathsf{Chs}_{NRm}(Q_0) \;=\; (\mathsf{Chs}(Q_0) - (\mathsf{Chs}(P_0) \cap E_x)) \cup (\mathsf{Chs}_o(Q_0) \cap \mathsf{Chs}_i(P_0)).$$

In the previous example, the set of channels of Q' $\{c_3, c_8, c_9\}$ are essential to maintain identical communication with the environment. c_2, despite being an external connection, has also been communicated with the external environment through process P. Then, it becomes an inessential communication of Q' when composed with P. c_5 is necessary for P to run in parallel with Q'. However, it carries values identical to c_8 for all the Q' actions. Since c_8 singularly communicates with the external environment, it cannot be eliminated, but c_5 can be dropped as long as channel c_5 is renamed to c_8 in P. So, the process to be further minimized in communication is $(Q' \setminus \{c_5\})$ when the renamed P is given as the other component process. The set of non–removable channels of $(Q' \setminus \{c_5\})$ is then:

$$\mathsf{Chs}_{NRm}(Q' \setminus \{c_5\}) = \{c_3, c_8, c_9\}$$

Preserving Communication

Besides eliminating duplicate and useless output information, certain channels of the target process which coincide with channels of the other process can be removed as long as communication between these processes is maintained. This means that if an action α from the target process is conflict free only with action β from the other process, then some of the α channels can be hidden only if the restricted α, say α', is exclusively able to communicate with β.

Essentially, connections can be cut off as composition of the reduced target process with the other component process is bisimilar to the original compound process. Since actions of the compound process corresponds to a synchronous communication between the target and the other process, each action from the reduced target process must communicate with the other process as much as the original target action. Thus, connections of the target process can be removed as long as all essential connections are maintained and the reduced process maintains the capacity of communication with the other process.

To eliminate unnecessary connections, a dynamic analysis on communication of the target with the other process must then be performed. Let us first analyse the minimal connections necessary for the initial actions to maintain identical communication, and then analyse the minimal connections for the entire process.

To maintain identical capacity of communication, each reduced action must conflict actions from the other process as the original action does. The sets of channels with which an action α maintains the same capacity of communication is as follows:

$$\begin{aligned}\mu_\alpha(Q_0) \triangleq \{&\mathsf{Chs}_{NRm}(Q_0) \subseteq chs_1 \subseteq \mathsf{Chs}(Q_0) \mid \\ &\alpha \in \mathsf{Sort}_1(Q_0) \wedge \\ &(\beta \in \mathsf{Sort}_1(P_0) \wedge \neg(\alpha \mathrel{\mathsf{CF}} \beta)) \Rightarrow \neg(\mathit{res\text{-}ch}(\alpha, chs_1) \mathrel{\mathsf{CF}} \beta) \wedge \\ &\forall chs_2 \subset chs_1 \cdot \exists \gamma \in \mathsf{Sort}_1(P_0) \cdot \neg(\alpha \mathrel{\mathsf{CF}} \gamma) \wedge \mathit{res\text{-}ch}(\alpha, chs_2) \mathrel{\mathsf{CF}} \gamma\}\end{aligned}$$

A minimal set of channels comprises all the Q_0 non–removable channels and whenever α conflicts with an action from P_0, then α restricted to this minimal set of channels is also conflicting with the same action. Furthermore, the channel sets are minimal because none of their subsets hold such a property.

Once a set of minimal channel sets can be found for each Q_0 initial action, minimal sets of channels that satisfy all the initial actions at the same time can be selected.

$$\begin{aligned}\mu_1(Q_0) \triangleq \{chs \subseteq \mathsf{Chs}(Q_0) \mid \forall \alpha \in \mathsf{Sort}_1(Q_0) \cdot \mathit{exists\text{-}subset}(chs, \mu_\alpha(Q_0)) \wedge \\ \forall chs_1 \subset chs \cdot \exists \beta \in \mathsf{Sort}_1(Q_0) \cdot \neg \mathit{exists\text{-}subset}(chs_1, \mu_\beta(Q_0))\}\end{aligned}$$

$$\mathit{exists\text{-}subset}(chs, \mathit{ch\text{-}set}) = \exists chs_n \in \mathit{ch\text{-}set} \cdot chs_n \subseteq chs$$

$\mu_1(Q_0)$ gives all possible minimal sets of channels to which Q_0 initial actions can be restricted and still maintain identical communication with P_0 initial actions. But, we are looking for minimal sets of channels which make composition of P_0 with a restricted Q_0 bisimilar to the original compound process. Since these sets can be found for the initial actions of P_0 and Q_0, they can also be found for all pair of subprocesses of P_0 and Q_0 that are reachable from $P_0 \mid Q_0$. A minimal set of channels that makes the entire Q_0 to maintain identical communication with P_0 must satisfy $\mu_1(Q_0)$ for all pairs of processes reachable from $P_0 \mid Q_0$.

$$\mu(Q_0) \triangleq \{chs \subseteq \mathsf{Chs}(Q_0) \mid (P_0 \mid Q_0) \longrightarrow^* (P_i \mid Q_j) \Rightarrow \textit{exists-subset}(chs, \mu_1(Q_j)) \wedge \forall chs_1 \subset chs \cdot \exists P_k \mid Q_l \cdot (P_0 \mid Q_0) \longrightarrow^* (P_k \mid Q_l) \wedge \neg \textit{exists-subset}(chs_1, \mu_1(Q_l))\}$$

Notice that all of these sets of channels must maintain identical communication between initial actions of all pairs of processes reachable from $(P_0 \mid Q_0)$. If so, communication throughout all the restricted process is identical to communication of the original Q_0 with P_0.

Having all of these minimal sets of channels, a number of them with a minimal number of channels can be selected. Composition of Q_0 (restricted to any of these minimal sets) with P_0, and then restricted to channels E_x, is bisimilar to $(P_0 \mid Q_0) \upharpoonright_c E_x$.

In the example presented above, Q' maintains identical communication with the renamed P even when channels c_5 and c_4 are hidden. This means that composition of the renamed P ($P[c_8/c_5]$) with the restricted Q' is bisimilar to the original compound process:

$$((P[c_8/c_5] \mid (Q' \setminus \{c_5, c_4\})) \upharpoonright_c \{c_1, c_2, c_3, c_6, c_7, c_8, c_9\}) \sim (P \mid Q') \upharpoonright_c \{c_1, c_2, c_3, c_6, c_7, c_8, c_9\}.$$

6.5 Minimization of Interface Processes

Once we have analysed how to minimize a target process in the number of states and ports of communication, interface processes created by application of the presented decomposition combinator can be minimized as a concurrent process. Let us first analyse how it can be minimized in the number of states (as a concurrent process), and then check how its ports of communication can also be minimized.

To minimize the interface process as an interacting element, the don't care information must be eliminated. Then, let us check if there exists any information in the interface process which is not used when it is composed with the existing one (conflicting information). Looking at the definition of the decomposition combinator, every action in the interface process is a result of interfacing an existing action to yield the behaviour of a desired one. This means that each action of the interface process is actually composed with existing actions to simulate desired actions. As a result, the interface process is free of conflicting information.

Secondly, hiding information must be exploited to achieve minimal interface processes. The technique presented in Section 6.3 for exploitation of hiding information is based on the creation of multi–valued actions. Notice that the technique employed to choose alternative actions which achieve minimal processes, can also be used to remove alternative actions included in the interface process when distinct actions from the existing process are used to simulate the same desired action. This means that the technique of minimization exploiting hiding information is twofold: eliminating overspecification of the interface process introduced by the time it is constructed, and enhancing minimization by multi–valuation of internal communications.

Let us use the interface process constructed by decomposition of the traffic light controller with respect to the major road process (Section 5.4, and now shown in Figure 6.13) to illustrate how it can be minimized with respect to the interaction with the major road process.

The don't care hiding information can only be found when distinct stimuli drive the compound process to respond identically. Despite the fact that composition of TLC_{mj} with TLC_{div} responds identically to some distinct initial actions, the internal information interchanged between the component actions that respond identically are also identical. Moreover, the external information of both processes is distinct for all those actions that respond identically. So, no alternative actions can be created to improve minimization of such a process. But, it can still be minimized under compatibility of concurrent processes (Section 6.3.3). To do so, we must first check which pairs of subprocesses are concurrently compatible and then find the possible minimal partitions.

The sets of concurrently compatible subprocesses reachable from TLC_{div} under composition with TLC_{mj} are as follows:

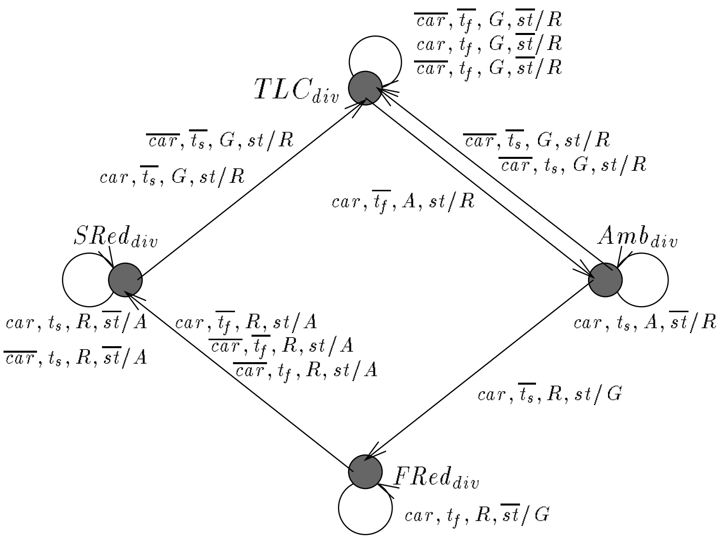

Figure 6.13: Interface between the Traffic Light Controller and the Major Road Process

$$\begin{aligned}
&\{\{TLC_{div}\}, \{Amb_{div}\}, \{FRed_{div}\}, \{SRed_{div}\},\\
&\{TLC_{div}, Amb_{div}\}, \{TLC_{div}, FRed_{div}\}, \{TLC_{div}, SRed_{div}\}, \{Amb_{div}, FRed_{div}\},\\
&\{TLC_{div}, Amb_{div}, FRed_{div}\}\}.
\end{aligned}$$

The pairs of processes Amb_{div} with $SRed_{div}$ and $SRed_{div}$ with $FRed_{div}$ are concurrently incompatible under composition with TLC_{mj}. Amb_{div} and $SRed_{div}$ are incompatible because t_f and t_s (and their negated elements) represent intervals instead of a single values. Actions

$$\begin{aligned}
&Amb_{div} \xrightarrow{car, \overline{t_s}, R, st/G} SRed_{div} \text{ and}\\
&SRed_{div} \xrightarrow{car, \overline{t_f}, R, st/A} FRed_{div}
\end{aligned}$$

cannot be distinguished by values $\overline{t_f}$ and $\overline{t_s}$ because by the time t_f is negated, t_s is also negated (interval t_f is longer than t_s). Since these actions differ in outputs, they are concurrently incompatible. For the same reason, processes $FRed_{div}$ and $SRed_{div}$ are concurrently incompatible.

Partitions of this interface process can now be given as in Definition 6.5. The minimal partitions that can be constructed by those sets of concurrently compatible processes are as follows:

$$\begin{aligned}
\rho_1 &= \{\{TLC_{div}, SRed_{div}\}, \{Amb_{div}, FRed_{div}\}\}\\
\rho_2 &= \{\{SRed_{div}\}, \{TLC_{div}, Amb_{div}, FRed_{div}\}\}
\end{aligned}$$

These processes have their behaviour shown in Figure 6.14 where the partitions elements are named by:

process $G_{mj\text{-}mn}(s\colon sensor, t\colon intervs, l_{mj}\colon colour, setc\colon stcount/l_{mn}\colon colour) \triangleq$
$\lfloor TLC_{div}, SRed_{div} \rceil$

process $A_{mj\text{-}mn}(s\colon sensor, t\colon intervs, l_{mj}\colon colour, setc\colon stcount/l_{mn}\colon colour) \triangleq$
$\lfloor Amb_{div}, FRed_{div} \rceil$

process $R_{mj\text{-}mn}(s\colon sensor, t\colon intervs, l_{mj}\colon colour, setc\colon stcount/l_{mn}\colon colour) \triangleq$
$\lfloor SRed_{div} \rceil$

process $GAR_{mj\text{-}mn}(s\colon sensor, t\colon intervs, l_{mj}\colon colour, setc\colon stcount/l_{mn}\colon colour) \triangleq$
$\lfloor TLC_{div}, Amb_{div}, FRed_{div} \rceil$

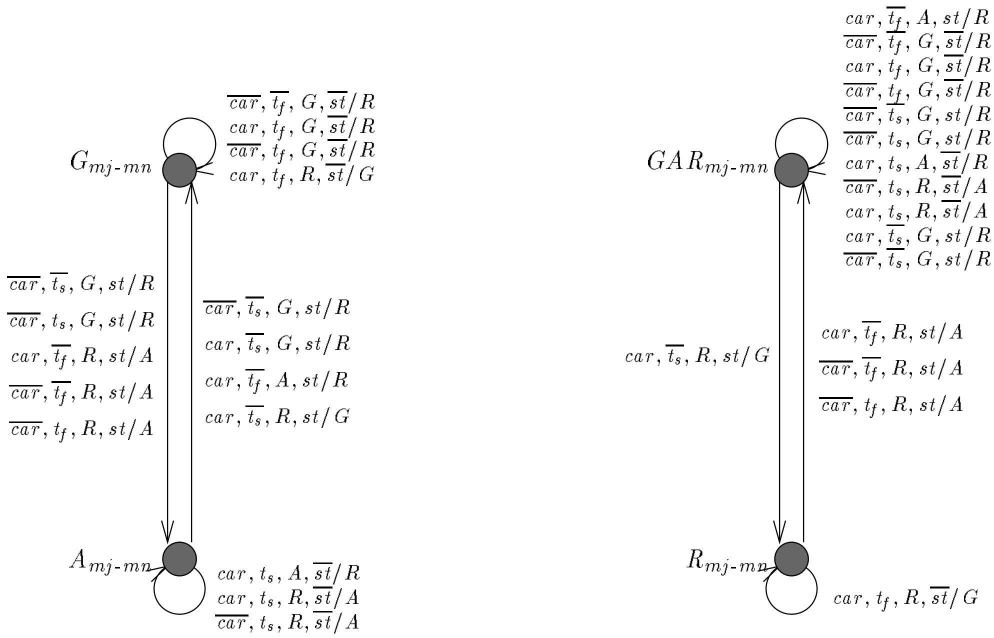

Figure 6.14: Minimization of States of the Traffic Light Interface Process

Once we have minimized the interface process under concurrent compatibility with TLC_{mj}, the minimal connections necessary to maintain identical communication with TLC_{mj} can be fixed. l_{mn} is the only non–removable communication because it is singularly communicated by the interface processes. Also, these processes have no duplicate output. So, their ports of communication can be minimized as far as they maintain identical capacity of communication with TLC_{mj} and l_{mn} is included.

Process $G_{mj\text{-}mn}$, for example, must be composed with TLC_{mj} and with $FRed_{mj}$ to simulate processes TLC_2 and $FRed_2$ respectively. Whenever TLC_{mj} signals G in l_{mj}, $G_{mj\text{-}mn}$ signals R in l_{mn}. Also, all actions from TLC_{mj} that signals A can only be combined with actions from $G_{mj\text{-}mn}$ that respond R. However, when R is signaled, $G_{mj\text{-}mn}$ is able to respond either G or A. So, besides communicating with TLC_{mj} and $FRed_{mj}$ through channel l_{mj}, $G_{mj\text{-}mn}$ must communicate through another port in order to maintain identical capacity of communication with $FRed_{mj}$. If $G_{mj\text{-}mn}$ communicates with TLC_{mj} or $FRed_{mj}$ through channels l_{mj} and $setc$, then all actions that have a distinct output is also distinct in communication with those processes. This means that communication of $G_{mj\text{-}mn}$ with TLC_{mj} or $FRed_{mj}$ is maintained if $G_{mj\text{-}mn}$ is restricted to channels l_{mn}, l_{mj} and $setc$.

A minimal set of communication has been defined for the initial actions of $G_{mj\text{-}mn}$ when combined with TLC_{mj} and $FRed_{mj}$. To conclude the analysis on the minimal number of ports of communication, the other reachable pairs of component processes must also have their minimal communication checked. Using the same sort of analysis, it can be verified that communication through channels l_{mj} and l_{mn} are sufficient to maintain identical communication of $A_{mj\text{-}mn}$ with Amb_{mj} or $SRed_{mj}$. So, the minimal set of channels that maintain identical communication between TLC_{mj} and $G_{mj\text{-}mn}$ is $\{l_{mn}, l_{mj}, setc\}$. Figure 6.15 shows the structural definition of the traffic light controller implemented by composition of the major road process with an interface process.

The other interface process, $GAR_{mj\text{-}mn}$, can also have its ports of communication restricted to channels l_{mn}, l_{mj} and $setc$. Figure 6.16 shows the behaviour of both processes restricted to the set of channels mentioned above (the restricted $G_{mj\text{-}mn}$ is on the left–hand side and $GAR_{mj\text{-}mn}$ is on the right).

Any of these processes can be picked up as the new component to be developed in order to yield bisimilarity with the desired traffic light controller. It is important to note that these processes do not represent the minor road process; they are only synchronous interfaces between the major road process and the traffic light controller. The question in Chapter 3 if the minor road process is the minimal interface process required can now be answered. It is not the minimal process required and, besides

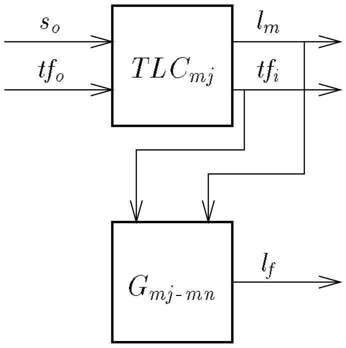

Figure 6.15: Interconnection between the Major Road and an Interface Process

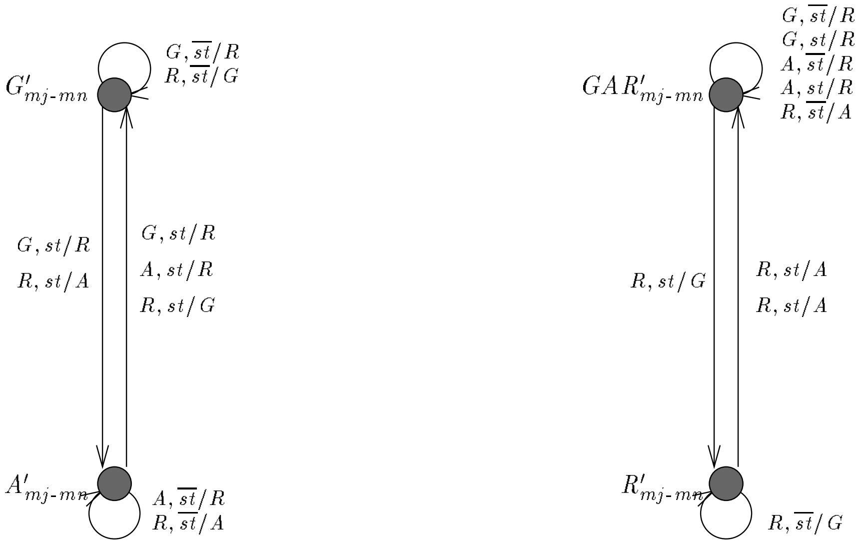

Figure 6.16: The Minimal Interface Processes for the TLC

that, some minimal processes less complex than the minor road process can be found by using the technique for reuse presented here.

6.6 Discussion

As we have already mentioned, all the interface actions are actually used to provide the behaviour of the desired process. Even in cases where the interface process has more than one action to simulate a single desired action, all of its actions are able to communicate with at least one action from the existing process. As a result, any interface process is minimal in the sense of having no useless actions. Making analogy with the serial composition, it has no input don't care information.

However, interface processes are not minimal as interacting processes because, under certain similarities of compound actions, they could eventually have some multi–valued actions to allow minimization. This kind of minimization can only occur when some distinct actions of the compound process are able to communicate identical outputs (alternative actions are obtained by multi–valuation of internal connections). So, applicability of this technique can be previously analysed by realising whether the desired process can communicate identical outputs for distinct actions. In spite of that, it is unpredictable whether the interface process can actually be minimized by the use of alternative actions. Even knowing in advance which actions can be multi–valued, a minimal process cannot be predicted, a procedure for minimization has to be applied to the interface processes. Since minimization depends on partitions under concurrent compatibility by making use of alternative actions, the entire process must be constructed in order to find out which states are concurrently compatible. This means that the best alternative actions cannot be chosen without having the whole process constructed. This fact disallows the creation of a decomposition operator to construct minimal interface processes.

In this minimization technique, ports of communication of processes can be reduced as long as each restricted action of the interface process can communicate with the existing process exactly as the non–reduced action. The technique for minimizing ports of communication was introduced by an algorithm. But, the underlying property to maintain identical communication with the existing process can be rephrased as: the minimal set of channels (including the non-removable ones) over which the interface process is deterministic for all the input/output actions that are combined with the existing process. This property ensures that each action from the interface process is conflict free with a single action from the existing process since the existing process is deterministic on input/output actions with respect to the desired one.

Minimization of communications is therefore a mechanism that provides all minimal sets of channels that include the non–removable channels and makes the reduced interface process deterministic for all elements of communication with the existing process (all minimal C such that $\mathsf{Chs}_{NRm}(Q_0) \in C$ and $\mathcal{D}!_{P_0}(Q_0 \lceil_c C)$). As a result, it can be proved that composition of a minimal interface process with the existing one is bisimilar to the desired process.

The procedures for minimization of communication based on the idea of removing all the inessential ports of communications. Deletion of unnecessary ports of communication preceding minimization under concurrent compatibility might inhibit minimization of the number of states. Certain removed channels could be essential to provide distinct communications of processes under concurrent compatibility. Similarly, minimization under concurrent compatibility may also inhibit minimization of ports of communication. Hence, the order in which these procedures are applied depends on whether the priority is given to the number of states or ports of communication of processes. We have deliberately chosen the "number of states" as priority for minimization of the interface processes.

The techniques presented here for minimization of the number of states and ports of communication of concurrent processes have been implemented as a prototype in SML [Pau92]. This prototype was used to check the main ideas of the minimization techniques, no further analysis on efficient algorithmic solutions has been made.

Chapter 7

Conclusions and Open Problems

The reuse of hardware components through formal means is still in its infancy (synthesis of hardware design has been discussed in Chapter 1 as a particular case of reuse). Even in the software domain, the formal reuse has only been approached by a few works that concentrate on partial definition of processes to accomplish reuse [Gog89, KRT89, VK89, PL89]. All these works emphasize the language used in order to achieve abstraction on the systems specification, and then enhance their potential of reuse; some of them are pure development of languages while others are a mixture of language and mechanisms for composing partial definitions.

In fact, abstract definition of processes is essential to obtain an effective reuse. In the present work we have assumed an existing model for representing hardware to concentrate on mechanisms to identify processes by making certain abstractions. This work has indicated that besides abstraction on representation of processes, mechanisms based on abstractions to identifying processes are also useful for reuse. Other works, such as PARIS [KRT89], have mechanisms for identifying processes based on attributes previously defined. In our work, however, the attributes to be matched are dynamically chosen. As a result, identification of similarities between processes is language independent (the similarity relations in Chapter 4 can be redefined for other synchronous process algebras). Nevertheless, these distinct directives to achieve reuse are not contradictory. It makes us believe that they can be worked together to reach more accurate methods of reuse.

Regarding the creation of interface processes, the present work has indicated the theoretical limits of reusing synchronous processes undertaking the interface approach (a process can only be reused if it is deterministically defined for all actions which simulate the desired specification). The interface approach depends essentially on the composition operator used, and then for each process algebra the essential preconditions must be calculated accordingly. The appropriate limits have not been considered in solutions for the interface equation defined for CCS, for example, and, as a result, only deterministic specification processes were considered there. It makes us believe that nondeterministic processes can be implemented by reuse of others in many process algebras as long as the behavioural similarities between processes are checked in advance. That is, the behavioural similarity between processes should be checked in advance rather than creating the interface process "on–the–fly".

A formal reuse of hardware design leads to a decompositional verification of processes. With the reuse of components formally verified, only the new elements being constructed need to be verified. Since composition of the interface with the existing process is proved bisimilar to the desired specification, verification of the interface element assures verification of the whole system. In fact, the topmost definition of the interface process is "correct" by construction, and only its refinements must be verified.

In the following sections we analyse the creation of interface processes as taking advantage of the behavioural approximation between components and specifications. Also, suggestions for further work and a brief discussion on certain problems involved in reuse that have not been addressed in the present work will be conducted.

7.1 On the Similarity Relations and Interface Processes

The techniques presented in this thesis is a basis for formal reuse of complex hardware components. The interface process is constructed in such a way that its composition with the given component is proved bisimilar to the desired specification. So, communication between the interface process and the component is formally established as well as the behaviour of the interface process.

The construction of "correct" interface processes has been proved throughout this work, but we have not yet stated that the best reuse of a component is achieved by using the decomposition operator. To do so, we must first establish the maximal similarity between synchronous processes and then show that such a similarity is reused. Another aspect to be discussed is whether the reuse of these similarities helps in developing processes less complex than the specification.

Recognizing and Reusing Similarities

Synchronous processes are characterized by the presence of a number of actions running simultaneously. EPA processes, in particular, have a predefined number of actions running in parallel. So, behaviour of synchronous processes might have all or only a subset of synchronous actions identified. The maximum behavioural approximation of processes is given by the maximum number of subactions that can be identified between one and another processes.

For the synchronous interface approach, however, similarity of processes is restricted to cases where each action of the specification is identified with at least one action of the component. Thus, using this approach, the maximum behavioural similarity is subjected to partial identification of synchronous actions such that each specification action is identified with a component one.

The similarity relations presented in this work identify partial actions by abstracting on the sorts defined for input and output actions. In fact, the similarity relations identify the greatest behavioural approximation between the specification and the component. Whenever a specification and a component have some similarities that are suitable for reuse, these similarities can be recognized by the presented similarity relations. But, are all these similarities actually reused when the specification is substituted by a component composed with the interface process?

The interface processes are created from similarities between specifications and components; each specification action is exclusively simulated by the component actions that is similar to it. Considering that the similarity relations are able to identify all the reusable similarities between a component and a specification, the decomposition combinator constructs interface processes that provide reuse of all similarities when running together with the component. Since each component action simulates a specification action and makes reuse of such similarities, the decomposition combinator makes indeed the best reuse of similarities, even knowing it does not generate a minimal process.

The interface process is not only non–minimal but it might be even more complex than the specification. The desire of creating a generic process that comprises all minimal processes (considering "overconnected" inputs) and, beyond that, the desire of providing algebraic manipulation of interface processes, drove us to construct a process which has, sometimes, more "states" than the specification. In spite of that, the number of states of the interface process is controlled by the number of pairs of subprocesses reachable from the specification and component that are related by a kind of similarity. Nevertheless, the final minimal process does never exceed the number of states of the specification. To be smaller than the desired process, it depends on how much similar the specification and the component are.

Similarities of Processes vs. the Number of States of Interface Processes

Certain similarities between specifications and components provide minimal interface processes smaller than the specification. We have claimed so far that the more similar processes, the smaller are the interface processes. However, no analysis on the complexity of interface processes against similarities has yet been provided. Let us now check how similarities between processes could contribute to having interface processes less complex than the specification one.

The complexity of minimal interface processes can only be realized by the time the minimization procedures are applied. Regarding minimization of concurrent processes, what are the approximations, in behaviour, between the specification and the component that achieve smaller interface processes?

The interface process has some information that is necessary to run the component or communicate with the environment. This information has been named as non–removable communications in the algorithms for minimization of communication. This is indeed the minimal information the interface process must supply in order to interface the desired and given components.

On the other hand, by minimization under concurrent compatibility, states of the interface process can only be joined if whenever they communicate with the existing process using identical information, they are also identical in communication with the external environment.

The more processes coincide in behaviour, the less non–removable channels exist. Any interface process should, in the best case, keep only its non–removable channels as long as its communication with the component is sufficient to ensure the properties of concurrent compatibility. If the number of external non–removable channels is as minimal as possible, then the chance of these external elements differing from each other is also minimized. As a result, minimization under concurrent compatibility could achieve better results. However, the number of distinct external behaviour is given by the actual values signaled on those external channels. So, the more these non–removable external channels coincide in behaviour, a better minimization is achieved under concurrent compatibility.

Summarizing, the best approximation of processes is constrained to have the external non–removable behaviour differing from each other as communication with the existing process also differs. In this case a single state process could be found. To some extent, having fewer external non–removable channels increases the chance of achieving a better minimization, but minimal processes are not predictable by a simple analysis on the number of non–removable channels. In fact, this analysis can only be realized under evaluation of the action values passed through these channels. So, fewer external non–removable channels is desirable because it increases the chance of resulting in processes less complex, but the actual minimization is unpredictable by the number of channels.

Then, let us now amend our claim about similarity of channels and complexity of interface processes. We cannot ensure that the more processes coincide in channels, the less complex is the interface process. It depends on the actual actions passing through the external non–removable channels. But, the chance of achieving a less complex interface process is increased by making the set of external non–removable channels (non–coincident channels) smaller.

7.2 On the Minimization Techniques

The techniques for minimization of concurrent processes presented in this work are restricted to synchronous processes. Minimization of processes under concurrent compatibility is only applicable to synchronous processes using a synchronous composition. This means that besides the similarity relations, all the other techniques for reuse of synchronous processes are dedicated to the EPA composition operator.

A few points must be discussed about the minimization techniques presented here: the reason for generating input "overconnected" processes, and the wide spectrum of application of such techniques.

To construct the minimal interface processes, minimization under concurrent compatibility as much as minimization of communication must be applied. We have already stated that the order in which these procedures are applied depends on whether priority is given to either the minimal number of states or communications. Certain connections will be dropped in all the minimal processes either when the number of states or connections is given priority. Even so, there is no advantage of creating interface processes without these "spare" connections.

Suppose the number of communications is given priority. Then, the interface process could be created over a reduced set of channels rather than the full set (creating the interface process from the specification and component processes restricted to the minimal set of channels). In spite of that, the minimization of communication anticipating the construction of interface processes makes no improvement in generating minimal processes. First, the procedure for minimizing communications does not depend on the number of “spare” communication. The complexity of such procedure is exclusively related to useful communications and the number of actions; no matter if “spare” communications are added. Second, the interface process is non–minimal in communication, even when created over a restricted set of channels. So, a procedure to minimize connections must be applied to achieve the minimal processes at the end. Third, a minimization procedure must be applied to find the “non–spare” connections at the beginning. The number of communications involved in this procedure is identical to the number of channels of the corresponding interface process without previous minimization. Thus, it does not justify anticipating minimization of connections when either the number of connections or states is given priority.

The minimization under concurrent compatibility and minimization of communication techniques are applicable to synchronous EPA processes in general, they are not confined to the reuse approach presented here. The process to be minimized is not necessarily an interface process, it could be any synchronous concurrent process. So, the minimization techniques can be applied to other fields apart from this formal reuse approach. For instance, both techniques can be introduced to a top–down method in order to enhance decomposition and reduce communication dependences among processes.

An improvement in minimizing concurrent processes can be achieved if both processes are simultaneously minimized. The minimization techniques presented here are confined to minimizing one process subjected to interaction with another. They were deliberately created in this way to support minimization of interface processes. Nevertheless, similar techniques can be developed in order to have both interacting processes minimized in terms of the number of states and ports of communication at the same time.

The interface processes in the present work are created to achieve bisimilarity with the specifications. The techniques for minimization of the interface process, therefore, result in a minimal interface processes that ensure bisimilarity with the specification. However, some other techniques can also be developed to obtain a similar (or strong similar) instead of a bisimilar compound process. Similarly, some other minimization techniques could be developed based on weaker similarity relations.

7.3 Reuse vs. Hardware Development

The suggestion of reusing similar processes and, at the same time, requiring bisimilarity (with the interface equation) appears controversial at a first stance, but they are justified by the fact that different approaches are currently adopted to develop hardware.

Various methods can be adopted to develop hardware using a top–down or a bottom–up approach. The technique for reuse proposed here does not conflict with any of these approaches, it could then be embedded in existing methods. It is indeed a mixture of both approaches. From one side, it is decompositional in the sense that the desired process is split into an existing and an interface process. From the other side, the constructive approach is regarded as the desired process is constructed from existing components. Obviously, the more abstract is the level of definition, the more effective is the reuse of design of processes when the library comprises design of processes from the definition level being used down to the implementation.

We have not presented any method of development because the reuse technique can be integrated to various methods of hardware development. Regarding development of hardware using top–down and/or bottom—up approaches, the idea of having simulation and algebraic manipulation of interface processes as part of the same technique is plausible.

Reuse of processes that are simulation of others is concerned with the idea of existing processes “matching” the topmost definition of desired processes. That is, the desired process communicates only with its environment. In this case, if deadlocks are unimportant or if a perfect environment is considered, a similar process can substitute for a desired one with no loss on behaviour and no need to develop a new component.

On the other hand, if the desired process is a subprocess embedded in a bigger system, then the process that is going to substitute for the desired one communicates with other processes to construct the entire system. If a constructive approach is adopted, algebraic manipulation of interface processes is useful in simplification of complex systems. Thus, the requirement of having both simulation and substitutivity is justified by the fact that constructive and top–down approaches could be used with this reuse technique.

Viability of a Formal Reuse

Interface processes can be more complex than the specifications, but their complexity is controllable by the number of pairs of related subprocesses from the specification and component. Furthermore, the construction of interface processes is a very simple task once the relation between these processes is known. So, viability of formal reuse depends basically on the decision on the similarity between processes, and minimization methods.

The "bottle neck" of constructing interface processes is the algorithm to decide similarity. Certain algorithms were presented to show decidability of such a problem, but these algorithms were not proposed as optimal solutions. Optimization of algorithms to decide similarity between processes is still an open problem. All of those algorithms presented in this work are based on an algorithm to decide bisimulation, by Fernandez [FM91], which has been proved efficient. However, no other technique has been tried to enhance performance of solutions for this problem.

Despite having complex algorithms to decide similarity, we have suggested throughout this work that reuse should be started at the topmost level of definition. At this level of definition, even complex system are manageable. Since processes are reused at the topmost level, the remaining process to be developed is less complex than the original one, even when it is required to be represented at a more detailed level. So, if processes can be reused at all levels of definition, the complexity of more detailed definitions is dropped.

Looking from a formal development point of view, verification of the whole system is continuously minimized by the fact that only unmatched parts of the specification are going to be developed. Since composition of the interface process with the component is assured bisimilar to the specification, the verification task can concentrate on the interface process. So, the formal verification of systems is decreased along with reuse of verified components.

This reuse technique can be combined with a top–down, a constructive, or a mixture of both approaches. Considering that reuse technique does not define a method for developing hardware design, how much does it improve hardware development if embedded into existing methods?

In a pure top–down design, processes are partitioned into smaller ones until the subprocesses are suitable for development. The reuse technique improves such a method as decomposition is automatically performed and the process to be developed is only an interface with the specification. We have seen as with the construction of the traffic light controller from the major road process that an interface process can be far less complex than the process to be designed using a conventional decomposition technique. Apart from decreasing the complexity of processes to be developed, this reuse technique helps the top–down design in minimizing communication among the defined subprocesses. A disadvantage of this technique is that the interface process depends on the definition of the component. It cannot be developed independently from the component as usually done in the top–down design. In some sense, it inhibits the decomposition technique applied in the top–down design in favor of finding the interface process with respect to the existing component.

In constructive approaches, reuse technique is basically used to identify the processes that have already been developed. Communication among the process network can be minimized if each subprocess has its communication minimized in relation to the other elements of this network. Again, independence of all subprocesses can only be achieved as minimization of communication is performed.

7.4 Alternative Solutions

A decomposition operator has been defined in this work to support algebraic manipulation of interface processes. But, having the interface equation as a target, a procedural solution can also be given to this problem.

The application of the decomposition operator generates overspecified and overconnected interface processes. As a result, minimization techniques were required at the end. However, a procedural solution might result in the various minimal solutions for this process rather than an interface process that comprise all minimal solutions.

In order to obtain minimal interface processes by construction, we must first check how the specification and the component can be reduced to generate minimal interfaces. Interface processes must be minimal under concurrent compatibility with the component. Since the interface actions comprise information from the similar specification and component actions, interface subprocesses are concurrently compatible if whenever component actions are identical, the corresponding specification actions also coincide. Notice that the set of coincident channels of the interface and the component processes correspond to the set of channels of the component, and the set of external channels are those from the specification. So, interface subprocesses can only be concurrently compatible if the component subprocesses that are going to be composed with them have only distinct actions, or whenever they coincide, they simulate identical specification actions.

Thus, the specification and the component can be reduced looking forward to obtaining a minimal interface process. To generate minimal interface processes, the specification must be reduced in such a way that for each of its subpartitions, all distinct actions from subprocesses in there are simulated by distinct actions from the component. If these actions are identical and simulated by identical component actions, then they must reach processes that lay on an identical subpartition. On the other hand, the component must also be reduced: all component subprocesses that are "reuse similar" (related by any similarity relation) to elements in a subpartition of the reduced specification must form a partition of the reduced component.

A procedural solution for this problem can be roughly described as follows:

1. the similarity between the specification and the component is checked;
2. the specification and component are minimized undergoing concurrent compatibility of the interface process with the component, and the set of minimal processes is selected;
3. interface processes are constructed from pairs of these reduced processes;
4. each interface process is minimized by exploiting hiding information;
5. select the set of minimal processes;
6. the communication of all the minimal processes must also be minimized; and then
7. select all processes that are minimal in the number of states and ports of communication.

Minimization of the specification and component undergoing concurrent compatibility results in a set of reduced processes. So, a number of processes might require further minimization as an interacting process and number of connections.

The advantage of this approach is that the complexity of the interface process is known from the beginning (apart from minimization of interacting processes) without even constructing it. On the other hand, algebraic manipulation of interface processes is not supported in this approach, and the computational "bottle neck" of the whole problem still remains: checking similarities between the specification and the component.

Variants of this procedural approach can be adopted. Minimization exploiting hiding information could, for example, be pulled to the stage where the specification and the component are firstly minimized; overspecification could be abolished in the first phase. Also, other solutions for this problem when minimization on communication is given priority require variations of such a procedure.

Even in an algebraic approach, minimization of the specification and the component could be employed as long as they can still comprise all minimal solutions. Before applying the decomposition operator, a minimization of the specification and the component (similar to that one suggested in the procedural approach) can be performed. Rather than giving a set of minimal processes, the reduced specification and component should comprise all the minimal solutions. This minimal generic process should partition the specification in such a way that processes can only be joined if they can be collapsed into all of those minimal solutions. This generic minimal process could possibly be less complex than the specification and consequently generate a less complex interface process to be minimized. However, it is not certain that any improvement could be achieved on complexity of the generic interface process and, besides that, minimization under concurrent compatibility should be applied twice. So, this solution cannot be stated as better than the solution given in the present work.

To avoid overspecification of the generic interface process, a minimization could also be performed in order to make each specification action to be simulated by exactly one component action. This kind of minimization would provide reduction of the number of dependence equations to be solved. Many other solutions could be tried, but the main computational problem remains: deciding for the behavioural similarity between the specification and the component.

Appendix A

Glossary of Symbols

Basic Entities

Stop	process Stop
$P, Q, S...$	process name
$\alpha, \beta, \gamma...$	action name
i/o	input/output action
io	input or output action
$a, b, ..., k$	input action name
$r, s, ..., z$	output action name
c_n	channel name

Sets of Entities

$\mathcal{A}$	set of actions
$\mathsf{Sort}(P)$	actions of P
$\mathsf{Sort}_i(P)$	input actions of P
$\mathsf{Sort}_o(P)$	output actions of P
$\mathsf{Sort}_1(P)$	initial actions of P
$\mathsf{Sort}_{1i}(P)$	initial input actions of P

Sets of Entities –cont.

$\mathsf{Sort}_{1o}(P)$	initial output actions of P
C	set of channels
E_x	set of obsevational channels
$\mathsf{Chs}(P)$	channels of P
$\mathsf{Chs}_i(P)$	input channels of P
$\mathsf{Chs}_o(P)$	output channels of P
$\mathsf{Chs}_{NRm}(P)$	non–removable channels of P

Basic Action Constructions

$(i_p/o_p) \oplus (i_q/o_q)$	summation
$(i_p/o_p) \ominus (i_q/o_q)$	subtraction
$(i_p/o_p) \star (i_q/o_q)$	product
$(i_p/o_p) \div (i_q/o_q)$	division

Basic Process Constructions

$\alpha :: P$	prefixing
$P + Q$	summation
$P \mid Q$	composition
$P \ / \ Q$	decomposition
$P \setminus C$	hide channels
$P \upharpoonright_c C$	restriction to channels
$P \upharpoonright_a \mathcal{A}$	restriction to actions

Similarity Entities

$\mathcal{B}$	bisimulation
$\mathcal{S}$	simulation
$\mathcal{S}_{tf}$	simulation under a total functional interface
$\mathcal{S}_f$	simulation under a functional interface
$\mathcal{B}_c$	channel restricted bisimulation
$\mathcal{S}_c$	channel restricted simulation
$\mathcal{S}_{ctf}$	channel restricted simulation under a total functional interface
$\mathcal{S}_{cf}$	channel restricted simulation under a functional interface
$\mathcal{S}_s$	strong simulation
$\mathcal{S}_{sf}$	strong simulation under a functional interface

Processes Relations

$P \sim Q$	P and Q are bisimilar
$P \sqsubseteq Q$	Q is a simulation of P
$P \overset{f}{\approx} Q$	Q is a simulation of P under a total functional interface
$P \overset{f}{\sqsubseteq} Q$	Q is a simulation of P under a functional interface
$P \overset{c}{\sim} Q$	P and Q are channel restricted bisimilar
$P \overset{c}{\sqsubseteq} Q$	Q is a channel restricted simulation of P
$P \overset{cf}{\approx} Q$	Q is a channel restricted simulation of P under a total functional interface
$P \overset{cf}{\sqsubseteq} Q$	Q is a channel restricted simulation of P under a functional interface
$P \sqsubseteq_s Q$	Q is a strong simulation of P
$P \overset{f}{\sqsubseteq}_s Q$	Q is a strong simulation of P under a functional interface
$P \simeq Q$	P and Q are compatible
$P \simeq_{\parallel} Q$	P and Q are concurrently compatible

Further Entities and Properties

$TS(P) = \{\mathsf{Sts}_P, \mathcal{A}_P, T^P, s_0^P\}$	transition system of P
Sts_P	the set of P states
s^P	a state of P
$\mathcal{A}_P$	the set of P actions
T^P	P transitions
T^P_α	P transitions for action α
$T^{P^{-1}}$	from incident to emerging P transitions
$T_\alpha^{P^{-1}}[B^P]$	states from which states in B^P are reached through α
ρ	partition of process states
B^P	set of states (block) of process P
$\mathcal{C}_\sim(P)$	the class of P's bisimilar processes
$\lfloor P, Q \rceil$	P and Q are made a joint process
α CF β	conflict free actions
$\mathcal{D}!_\alpha(P)$	P is deterministic for action α
$\mathcal{D}!(P)$	P is deterministic for all input/output actions
$\mathcal{D}!_S(P)$	P is deterministic with respect to communication with S

Appendix B

EPA – semantics

B.1 Actions and their Operators

B.1.1 Actions

$$
\begin{aligned}
&\alpha \ \in Act ::= \ \epsilon \ \mid \ \boxtimes \ \mid \ (i/o) \\
&i, o \in IOAct ::= \ \epsilon \ \mid \ \boxtimes \ \mid \ io_1 \oplus io_2 \ \mid \ pa \\
&\epsilon/\epsilon = \epsilon \\
&\boxtimes/\boxtimes = \boxtimes
\end{aligned}
$$

$\alpha, \beta, ... \in Act$	synchronous actions
$i/o \in Act$	input/output action
$io \in IOAct$	input or output action
$\epsilon \in Act$	empty action
$\epsilon \in IOAct$	input or output empty action
$\boxtimes \in Act$	zero action
$\boxtimes \in IOAct$	zero input or output action
$pa \in pIOAct$	primitive action

B.1.2 Action Operators

Summation

$$
\oplus : \ pIOAct \cup \{\boxtimes, \epsilon\} \times pIOAct \cup \{\boxtimes, \epsilon\} \rightarrow IOAct
$$

$$
\begin{aligned}
(c_1 \neq c_2) \quad &\Rightarrow \quad c_1(v_1) \oplus c_2(v_2) = c_1(v_1) \oplus c_2(v_2) \\
(v_1 \neq v_2) \quad &\Rightarrow \quad c(v_1) \oplus c(v_2) = \boxtimes \\
& \quad\ \ \ c(v) \oplus c(v) = c(v) \\
& \quad\ \ \ c(v) \oplus \boxtimes = \boxtimes = \boxtimes \oplus c(v) \\
& \quad\ \ \ c(v) \oplus \epsilon = c(v) = \epsilon \oplus c(v)
\end{aligned}
$$

Subtraction

$$\ominus : pIOAct \cup \{\boxtimes, \epsilon\} \times pIOAct \cup \{\boxtimes, \epsilon\} \rightarrow IOAct$$

$$\begin{aligned}
(c_1 \neq c_2) \quad \Rightarrow \quad & c_1(v_1) \ominus c_2(v_2) = c_1(v_1) \\
& c(v) \ominus c(v) = \epsilon \\
& c(v) \ominus \boxtimes = \boxtimes = \boxtimes \ominus c(v) \\
& c(v) \ominus \epsilon = c(v) = \epsilon \ominus c(v)
\end{aligned}$$

Causal Product

$$\star : Act^2 \rightarrow Act$$

$$\forall\ i_p/o_p,\ i_q/o_q \in Act \bullet$$

$$(i_p/o_p) \star (i_q/o_q) = ((i_p \oplus i_q) \ominus (o_p \oplus o_q))/(o_p \oplus o_q)$$

Interface Division

$$\div : Act^2 \rightarrow Act$$

$$\forall\ i_p/o_p,\ i_q/o_q \in Act \bullet$$

$$(i_p/o_p) \div (i_q/o_q) = (i_p \oplus o_q)/(o_p \ominus o_q) \oplus (i_q \ominus i_p)$$

Hide Channels

$$hide\text{-}act : Act \times \mathcal{F}(ChanNames) \rightarrow Act$$

$$hide\text{-}act(\epsilon, C) = \epsilon$$

$$hide\text{-}act(\boxtimes, C) = \boxtimes$$

$$hide\text{-}act(i/o, C) = (hide\text{-}io\text{-}act(i, C)/hide\text{-}io\text{-}act(o, C))$$

$$hide\text{-}io\text{-}act : IOAct \times \mathcal{F}(ChanNames) \rightarrow IOAct$$

$$hide\text{-}io\text{-}act(\epsilon, C) = \epsilon$$

$$hide\text{-}io\text{-}act(\boxtimes, C) = \boxtimes$$

$$hide\text{-}io\text{-}act(c(v), C) = hide\text{-}prim\text{-}act(c(v), C)$$

$$hide\text{-}io\text{-}act(io_1 \oplus io_2, C) = hide\text{-}io\text{-}act(io_1, C) \oplus hide\text{-}io\text{-}act(io_2, C)$$

$$hide\text{-}prim\text{-}act : pIOAct \times \mathcal{F}(ChanNames) \rightarrow pIOAct \cup \{\epsilon\}$$

$$\begin{aligned}
hide\text{-}prim\text{-}act(c(v), C) = {} & \textsf{if } chan\text{-}of(c(v)) \in C \\
& \textsf{then } \epsilon \\
& \textsf{else } c(v)
\end{aligned}$$

Restriction to Channels

$$res\text{-}ch\colon Act \times \mathcal{F}(ChanNames) \rightarrow Act$$

$$res\text{-}ch(\epsilon, C) = \epsilon$$

$$res\text{-}ch(\boxtimes, C) = \boxtimes$$

$$res\text{-}ch(i/o, C) = (res\text{-}io\text{-}ch(i, C)/res\text{-}io\text{-}ch(o, C))$$

$res\text{-}io\text{-}ch\colon IOAct \times \mathcal{F}(ChanNames) \rightarrow IOAct$
$res\text{-}io\text{-}ch(\epsilon, C) = \epsilon$
$res\text{-}io\text{-}ch(\boxtimes, C) = \boxtimes$
$res\text{-}io\text{-}ch(c(v), C) = res\text{-}prim\text{-}ch(c(v), C)$
$res\text{-}io\text{-}ch(io_1 \oplus io_2, C) = res\text{-}io\text{-}ch(io_1, C) \oplus res\text{-}io\text{-}ch(io_2, C)$

$res\text{-}prim\text{-}ch : pIOAct \times \mathcal{F}(ChanNames) \rightarrow pIOAct \cup \{\epsilon\}$

$$res\text{-}prim\text{-}ch(c(v), C) \quad \underline{\triangle} \quad \begin{array}{l} \textsf{if } chan\text{-}of(c(v)) \in C \\ \textsf{then } c(v) \\ \textsf{else } \epsilon \end{array}$$

Subtraction Laws:

(1) $hide\text{-}io\text{-}act(io_1 \ominus io_2, C) = hide\text{-}io\text{-}act(io_1, C) \ominus hide\text{-}io\text{-}act(io_2, C)$
(2) $res\text{-}io\text{-}ch(io_1 \ominus io_2, C) = res\text{-}io\text{-}ch(io_1, C) \ominus res\text{-}io\text{-}ch(io_2, C)$

B.1.3 Action Properties

Propagation of Zero Action:

1. $(i/\boxtimes) = \boxtimes$
2. $(\boxtimes/o) = \boxtimes$

Conflict Freedom:

This relation is defined upon input or output actions.

1. $(\_ \;\mathsf{CF}\; \_)$ is symmetric.
2. $\neg(\boxtimes \;\mathsf{CF}\; \boxtimes)$.
3. The Triangle Law
$(io_1 \oplus io_2) \;\mathsf{CF}\; io_3 \Leftrightarrow (io_1 \;\mathsf{CF}\; io_2) \wedge (io_1 \;\mathsf{CF}\; io_3) \wedge (io_2 \;\mathsf{CF}\; io_3)$.

B.2 Transitional Semantics for EPA Combinators

Let $\alpha, \beta \in Act$, x be a variable, set be a set of literal values of actions, $C \subseteq ChanNames$, pnm be a process name, and P, P', Q, Q' be process terms:

$$\boxed{\textsf{Vacuum}}\ \frac{}{P \xrightarrow{\boxtimes} P'}$$

$$\boxed{\textsf{Prefixing}}\ \frac{}{\alpha :: P \xrightarrow{\alpha} P}$$

$$\boxed{\textsf{Binary Sum}}\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

$$\boxed{\textsf{ProcDefn}}\ \frac{pnm(\vec{x}) \triangleq Q, \quad Q[\vec{v}/\vec{x}] \xrightarrow{\alpha} P'}{pnm(\vec{v}) \xrightarrow{\alpha} P'}$$

$$\boxed{\textsf{Hiding}}\ \frac{P \xrightarrow{\alpha} P'}{P \backslash C \xrightarrow{hide\text{-}act(\alpha, C)} P' \backslash C}$$

$$\boxed{\textsf{Product}}\ \frac{P \xrightarrow{\alpha} P', \quad Q \xrightarrow{\beta} Q'}{P \mid Q \xrightarrow{\alpha \star \beta} P' \mid Q'}$$

$$\boxed{\textsf{ProcChoice}}\ \frac{\text{for some } v \in set, \quad P\,[\,v\,/\,x\,] \xrightarrow{\alpha} P'}{\Box\, x:\ set\ \bullet\ P \xrightarrow{\alpha} P'}$$

$$\boxed{\textsf{RestrChannels}}\ \frac{P \xrightarrow{\alpha} P'}{P \upharpoonright_c C \xrightarrow{res\text{-}ch(\alpha, C)} P' \upharpoonright_c C}$$

$$\boxed{\textsf{RestrActions}}\ \frac{P \xrightarrow{\alpha} P', \quad \alpha \in \mathcal{A}, \quad \mathcal{A} \subseteq Act}{P \upharpoonright_a \mathcal{A} \xrightarrow{\alpha} P' \upharpoonright_a \mathcal{A}}$$

Appendix C

Validation of Propositions and Theorems

C.1 Chapter 4

C.1.1 Section 4.4

Theorem 4.1 *$\mathcal{B} \subseteq \mathcal{S}$, $\mathcal{B} \subseteq \mathcal{S}_{tf}$, $\mathcal{B} \subseteq \mathcal{B}_c$, $\mathcal{S} \subseteq \mathcal{S}_f$, $\mathcal{S} \subseteq \mathcal{S}_c$, $\mathcal{S}_{tf} \subseteq \mathcal{S}_f$, $\mathcal{S}_{tf} \subseteq \mathcal{S}_{ctf}$, $\mathcal{B}_c \subseteq \mathcal{S}_c$, $\mathcal{B}_c \subseteq \mathcal{S}_{ctf}$, $\mathcal{S}_{ctf} \subseteq \mathcal{S}_{cf}$, $\mathcal{S}_f \subseteq \mathcal{S}_{cf}$ and $\mathcal{S}_c \subseteq \mathcal{S}_{cf}$.*
Proof.

- $\mathcal{B} \subseteq \mathcal{S}$: If $P \sqsubseteq Q$, for every "proper" action i_p/o_p such that i_p is in the input sort of P's initial actions, both processes perform identical actions. In bisimulation, both processes must behave equivalently for any proper action, while in simulation only the P's actions are taken into account. Bisimulation is obviously an instance of simulation where processes behave equivalently for any proper action rather than P's proper actions. As being an instance of simulation, all bisimilar processes are also similar: $\mathcal{B} \subseteq \mathcal{S}$.

- $\mathcal{B} \subseteq \mathcal{S}_{tf}$: For processes P and Q related under $\mathcal{S}_{tf}$, the existence of a function f to identify behaviour of these processes is necessary. This means that for every P's initial action i_p/o_p, there exists an action i_q/o_q in Q, and vice-versa, such that $f(i_p, i_q/o_q) = o_p$. For instance, if f exclusively maps actions of P and Q that are identical, then no interface is necessary to identify actions of P and Q and they are indeed bisimilar. Bisimulation is a particular case of simulation under a total functional interface where f exclusively relates identical actions of P and Q.

- $\mathcal{B} \subseteq \mathcal{B}_c$: Looking at the definition of $\mathcal{B}$, we can, with no loss of generality, redefine it as a symmetric binary relation such that if $(P, Q) \in \mathcal{B}$ and $\mathsf{Chs}(P) = \mathsf{Chs}(Q) = C$ then for every "proper" action $\alpha \in Act$:

  1. whenever $P \xrightarrow{\alpha} P'$ then $Q \xrightarrow{\alpha'} Q'$ for some α' and Q' such that
     $res\text{-}ch(\alpha, C) = res\text{-}ch(\alpha', C)$ and $P' \sim Q'$,
  2. whenever $Q \xrightarrow{\alpha} Q'$ then $P \xrightarrow{\alpha'} P'$ for some α' and P' such that
     $res\text{-}ch(\alpha, C) = res\text{-}ch(\alpha', C)$ and $P' \sim Q'$.

  Observe that $res\text{-}ch(\alpha, C) = res\text{-}ch(\beta, C)$ is reduced to $\alpha = \beta$ because all channel names of actions α and β are in C. Like bisimulation, channel restricted bisimulation can also be redefined as a symmetric binary relation such that if $(P, Q) \in \mathcal{B}_c$ and $\mathsf{Chs}(P) \cap \mathsf{Chs}(Q) = C$ then for every "proper" action $\alpha \in Act$:

1. whenever $P \xrightarrow{\alpha} P'$ then $Q \xrightarrow{\alpha'} Q'$ for some α' and Q' such that $res\text{-}ch(\alpha, C) = res\text{-}ch(\alpha', C)$ and $P' \sim Q'$,
2. whenever $Q \xrightarrow{\alpha} Q'$ then $P \xrightarrow{\alpha'} P'$ for some α' and P' such that $res\text{-}ch(\alpha, C) = res\text{-}ch(\alpha', C)$ and $P' \sim Q'$.

This definition resembles that one for bisimulaiton, except that $\mathsf{Chs}(P) \cap \mathsf{Chs}(Q) = C$ appears instead of $\mathsf{Chs}(P) = \mathsf{Chs}(Q) = C$. The constraint $\mathsf{Chs}(P) \cap \mathsf{Chs}(Q) = C$ is less restrictive than that one for bisimulation. Bisimulation is therefore an instance of channel restricted bisimulation such that $\mathsf{Chs}(P) = \mathsf{Chs}(Q)$, and then $\mathcal{B} \subseteq \mathcal{B}_c$.

- $\mathcal{S} \subseteq \mathcal{S}_f$: The arguments used for $\mathcal{B} \subseteq \mathcal{S}_{tf}$ also apply to this assertion, except that only actions of P are identified with Q actions through f. Simulation is thus a particular case of simulation under a functional interface where f exclusively relates identical actions of P and Q.

- $\mathcal{S} \subseteq \mathcal{S}_c$: This assertion is proved like $\mathcal{B} \subseteq \mathcal{B}_c$. The definition of $\mathcal{S}$ can be rephrased as an antisymmetric binary relation such that if $(P, Q) \in \mathcal{S}$ and $\mathsf{Chs}(P) = \mathsf{Chs}(Q) = C$ then for every "proper" action $i_1/o_1, i_2/o_2 \in Act$ such that $i_1 \in \mathsf{Sort}_{1i}(P)$:

  1. whenever $P \xrightarrow{i_1/o_1} P'$ then $Q \xrightarrow{i_2/o_2} Q'$ for some Q' such that $res\text{-}ch(i_1/o_1, C) = res\text{-}ch(i_2/o_2, C)$ and $P' \sqsubseteq Q'$,
  2. whenever $Q \xrightarrow{i_2/o_2} Q'$ and $res\text{-}ch(i_1/o_1, C) = res\text{-}ch(i_2/o_2, C)$ then $P \xrightarrow{i_1/o_1} P'$ for some P' such that $P' \sqsubseteq Q'$.

  Channel restricted simulation can also be redefined as an antisymmetric binary relation such that if $(P, Q) \in \mathcal{S}_c$ and $\mathsf{Chs}(P) \cap \mathsf{Chs}(Q) = C$ then for every "proper" action $i_1/o_1, i_2/o_2 \in Act$ such that $i_1 \in \mathsf{Sort}_{1i}(P)$:

  1. whenever $P \xrightarrow{i_1/o_1} P'$ then $Q \xrightarrow{i_2/o_2} Q'$ for some Q' such that $res\text{-}ch(i_1/o_1, C) = res\text{-}ch(i_2/o_2, C)$ and $P' \sqsubseteq Q'$,
  2. whenever $Q \xrightarrow{i_2/o_2} Q'$ and $res\text{-}ch(i_1/o_1, C) = res\text{-}ch(i_2/o_2, C)$ then $P \xrightarrow{i_1/o_1} P'$ for some P' such that $P' \sqsubseteq Q'$.

  If the constraint $\mathsf{Chs}(P) = \mathsf{Chs}(Q) = C$ is required in the definition above, it equals the previous one for simulation and then $\mathcal{S} \subseteq \mathcal{S}_c$ holds.

- $\mathcal{S}_{tf} \subseteq \mathcal{S}_f$: Once $\mathcal{B} \subseteq \mathcal{S}$, $\mathcal{S} \subseteq \mathcal{S}_f$ and $\mathcal{B} \subseteq \mathcal{S}_{tf}$ (above) have been proved, we can conclude $\mathcal{S}_{tf} \subseteq \mathcal{S}_f$. In other words, $\mathcal{S}_{tf}$ is an instance of $\mathcal{S}_f$ where all actions of the simulation process simulate (through f) at least one action of the simulated process.

- $\mathcal{S}_{tf} \subseteq \mathcal{S}_{ctf}$: $\mathcal{B} \subseteq \mathcal{B}_c$ has already been stated. Also, by the definition of $\mathcal{S}_{ctf}$, f is only applicable to channel restricted actions. If a constraint is established such that C comprises all channels of both processes, then $\mathcal{S}_{ctf}$ is reduced to the definition of $\mathcal{S}_{tf}$. $\mathcal{S}_{tf}$ is indeed an instance of $\mathcal{S}_{ctf}$ where the full sets of channels of both processes are considered.

- $\mathcal{B}_c \subseteq \mathcal{S}_c$: $\mathcal{B}_c \subseteq \mathcal{S}_c$ is deduced from the premises $\mathcal{B} \subseteq \mathcal{S}$ and $\mathcal{B} \subseteq \mathcal{B}_c$.

- $\mathcal{B}_c \subseteq \mathcal{S}_{ctf}$: $\mathcal{B}_c \subseteq \mathcal{S}_{ctf}$ is concluded from $\mathcal{B} \subseteq \mathcal{S}_{tf}$ and $\mathcal{B} \subseteq \mathcal{B}_c$. $\mathcal{B}_c$ is an instance of $\mathcal{S}_{ctf}$ where f only relates actions that are channel restricted identical.

- $\mathcal{S}_{ctf} \subseteq \mathcal{S}_{cf}$: it is deduced from $\mathcal{B}_c \subseteq \mathcal{S}_c$ and $\mathcal{S}_{tf} \subseteq \mathcal{S}_f$. Roughly speaking, $\mathcal{S}_{ctf}$ is a particular case of $\mathcal{S}_{cf}$ where each restricted actions from Q simulate at least one restricted action of P.

- $\mathcal{S}_f \subseteq \mathcal{S}_{cf}$: it is a consequence of $\mathcal{S}_{tf} \subseteq \mathcal{S}_{ctf}$ and $\mathcal{S}_{tf} \subseteq \mathcal{S}_f$. Informally, $\mathcal{S}_f$ is a particular case of $\mathcal{S}_{cf}$ where C comprises all channels of both processes.

- $\mathcal{S}_c \subseteq \mathcal{S}_{cf}$: $\mathcal{S}_c \subseteq \mathcal{S}_{cf}$ is deduced from $\mathcal{B}_c \subseteq \mathcal{S}_c$ and $\mathcal{B}_c \subseteq \mathcal{S}_{ctf}$.

$\square$

C.1.2 Section 4.5

Proposition 4.1 $P \overset{cf}{\approx} Q$ *iff exists a non–empty* C *such that*
$C \subseteq (\mathsf{Chs}(P) \cup \mathsf{Chs}(Q))$ *and*
$P \restriction_c C \overset{f}{\approx} Q \restriction_c C$

Proof. By the definition of $\mathcal{S}_{ctf}$, f is applied to a restricted set of channels defined by equality: $f(\textit{res-io-ch}(i_p, C), \textit{res-io-ch}(i_q/o_q, C)) = \textit{res-io-ch}(o_p, C)$.

⇒: Existence of a non–empty C is explicitly proved by the $\mathcal{S}_{ctf}$'s definition. Since P and Q are restricted to channels C, the restricted io–actions from these processes are given by (Proposition 2.1):

$$\textit{res-io-ch}(i_p, C) = i'_p \quad \text{and} \quad \textit{res-io-ch}(o_p, C) = o'_p$$
$$\textit{res-io-ch}(i_q, C) = i'_q \quad \text{and} \quad \textit{res-io-ch}(o_q, C) = o'_q$$

Therefore, the equality defined for io–actions in $\mathcal{S}_{ctf}$ can now be redefined as:

$$f(i'_p, i'_q/o'_q) = o'_p$$

Substituting this new equality on the $\mathcal{S}_{ctf}$'s definition we may check that $P \restriction_c C$ and $Q \restriction_c C$ are similar under a total functional interface at least for their initial actions. Since restriction to channels is propagated over all actions, the initial actions of the reachable processes are also reduced to similarity under a total functional interface.

⇐: Similar to the previous case, the existence of C is proved by definition. Actions i'_p/o'_p, i'_q/o'_q, from $P \restriction_c C$ and $Q \restriction_c C$, have their corresponding extended versions in P and Q (i_p/o_p, i_q/o_q). Then, the action equality defined in $\mathcal{S}_{tf}$ (over those restricted processes),

$$f(i'_p, i'_q/o'_q) = o'_p$$

can be rewritten as (Proposition 2.2):

$$f(\textit{res-io-ch}(i_p, C), \textit{res-io-ch}(i_q/o_q, C)) = \textit{res-io-ch}(o_p, C)$$

Since this is applicable to all actions of those restricted processes, $\mathcal{S}_{tf}$ on $P \restriction_c C$ and $Q \restriction_c C$ is degenerated to $\mathcal{S}_{ctf}$ on P and Q.

$\square$

Proposition 4.2 $P \overset{cf}{\approx} Q$ *iff exists a non–empty* C *and a function* f *such that*
$C \subseteq (\mathsf{Chs}(P) \cup \mathsf{Chs}(Q))$,
$f\colon \mathsf{Sort}_i(P \restriction_c C) \times \mathsf{Sort}(Q \restriction_c C) \rightarrow \mathsf{Sort}_o(P \restriction_c C)$ *and*
$P \overset{c}{\sim} (Q \mid P_f)$

Proof.

⇒: The existence of C and f is proved by the definition of $\mathcal{S}_{ctf}$. Still looking at the $\mathcal{S}_{ctf}$'s definition, for each action i_p/o_p in P there exists an action i_q/o_q in Q that satisfies the equality

$$f(\textit{res-io-ch}(i_p, C), \textit{res-ch}(i_q/o_q, C)) = \textit{res-io-ch}(o_q, C).$$

Also, P_f implements f and then for each corresponding i_p/o_p and i_q/o_q actions, there also exists an action i_{pf}/o_{pf} in P_f such that

$$\begin{aligned} i_{pf} &= (\textit{res-io-ch}(i_p, C) \oplus \textit{res-io-ch}(o_q, C)) \ominus \textit{res-io-ch}(o_p, C) \\ o_{pf} &= (\textit{res-io-ch}(i_q, C) \oplus \textit{res-io-ch}(o_p, C)) \ominus \textit{res-io-ch}(i_p, C) \end{aligned}$$

Hence, composition of Q with P_f comprises action

$$\begin{aligned} (i_q/o_q) \star (i_{pf}/o_{pf}) &= (i_q/o_q)\star \\ &\quad (((\textit{res-io-ch}(i_p, C) \oplus \textit{res-io-ch}(o_q, C)) \ominus \textit{res-io-ch}(o_p, C))/ \\ &\qquad (\textit{res-io-ch}(i_q, C) \oplus \textit{res-io-ch}(o_p, C) \ominus \textit{res-io-ch}(i_p, C))) \\ &= ((i_q \oplus ((\textit{res-io-ch}(i_p, C) \oplus \textit{res-io-ch}(o_q, C)) \ominus \textit{res-io-ch}(o_p, C))) \\ &\quad \ominus(o_q \oplus (\textit{res-io-ch}(i_q, C) \oplus \textit{res-io-ch}(o_p, C) \ominus \textit{res-io-ch}(i_p, C))))/ \\ &\qquad (o_q \oplus (\textit{res-io-ch}(i_q, C) \oplus \textit{res-io-ch}(o_p, C)) \ominus \textit{res-io-ch}(i_p, C)) \\ &= (i_q \oplus \textit{res-io-ch}(i_p, C))/ \\ &\qquad (o_q \oplus \textit{res-io-ch}(i_q, C) \oplus \textit{res-io-ch}(o_q, C) \ominus \textit{res-io-ch}(i_p, C)) \end{aligned}$$

Notice that each compound action comprises the channel restricted P's action $\textit{res-ch}(i_p/o_p, C)$. So, for each action i_p/o_p from P, there exists a corresponding action $(i_q \oplus \textit{res-io-ch}(i_p, C))/o_q \oplus (\textit{res-io-ch}(i_q, C) \oplus \textit{res-io-ch}(o_q, C) \ominus \textit{res-io-ch}(i_p, C)))$ in the compound process. The compound process $Q \mid P_f$ is thus channel restricted bisimilar to P such that the set of coincident channels is given by $C \cap \mathsf{Chs}(P)$.

$\Leftarrow$: The implication backwards is also trivial. The existence of C and f are proved by their explicit definition on both hand sides. By the definition of $\mathcal{B}_c$, for each channel restricted action of P (i'_p/o'_p), there exists an action from $(Q \mid P_f)$, $(i'_q/o'_q \star i'_{pf}/o'_{pf})$, such that

$$\textit{res-ch}(i'_p/o'_p, C) = \textit{res-ch}(i'_q/o'_q \star i'_{pf}/o'_{pf}, C).$$

Since f, and consequently P_f, is defined over P and Q sorts, i'_{pf}/o'_{pf} must be defined over the corresponding actions i'_p/o'_p and i'_q/o'_q to achieve the equality above. Considering this constraint, the assertion $P \overset{cf}{\approx} Q$ holds.

$\square$

Proposition 4.3 $P \overset{cf}{\sqsubseteq} Q$ *iff exists a non–empty C and a function f such that*
$C \subseteq (\mathsf{Chs}(P) \cup \mathsf{Chs}(Q))$,
$f\colon \mathsf{Sort}_i(P \upharpoonright_c C) \times \mathsf{Sort}(Q \upharpoonright_c C) \rightarrow \mathsf{Sort}_o(P \upharpoonright_c C)$ *and*
$P \overset{c}{\sqsubseteq} (Q \mid P_f)$

A proof for this proposition is similar to that one for Proposition 4.1.

Proposition 4.4 $P \overset{cf}{\sqsubseteq} Q$ *iff exists a non–empty C such that*
$C \subseteq (\mathsf{Chs}(P) \cup \mathsf{Chs}(Q))$ *and*
$P \upharpoonright_c C \overset{f}{\sqsubseteq} Q \upharpoonright_c C$

This proposition is proved similarly to Proposition 4.2.

C.2 Chapter 5

C.2.1 Section 5.3

$\textit{res-ch}(i_x/o_x \star i_p/o_p, \mathsf{Chs}(S)) = i_s/o_s$

is rewritten as:

$$\begin{aligned} res\text{-}io\text{-}ch((i_x \oplus i_p) \ominus (o_x \oplus o_p), \mathsf{Chs}(S)) &= i_s \\ res\text{-}io\text{-}ch(o_x \oplus o_p, \mathsf{Chs}(S)) &= o_s. \end{aligned}$$

Equality of Output Actions

$$\begin{aligned} o_s &= res\text{-}io\text{-}ch(o_x \oplus o_p, \mathsf{Chs}(S)) \\ &= res\text{-}io\text{-}ch((o_s \ominus o_p) \oplus (i_p \ominus i_s) \oplus o_p, \mathsf{Chs}(S)) \\ &= res\text{-}io\text{-}ch(((o_s \ominus o_p) \oplus o_p) \oplus (i_p \ominus i_s), \mathsf{Chs}(S)) \\ &= res\text{-}io\text{-}ch((o_s \oplus o_p) \oplus (i_p \ominus i_s), \mathsf{Chs}(S)) \\ &= res\text{-}io\text{-}ch(o_s \oplus o_p, \mathsf{Chs}(S)) \oplus res\text{-}io\text{-}ch(i_p \ominus i_s, \mathsf{Chs}(S)) \\ &= res\text{-}io\text{-}ch(o_s, \mathsf{Chs}(S)) \oplus res\text{-}io\text{-}ch(o_p, \mathsf{Chs}(S)) \oplus res\text{-}io\text{-}ch(i_p \ominus i_s, \mathsf{Chs}(S)) \\ &= o_s \oplus res\text{-}io\text{-}ch(o_p, \mathsf{Chs}(S)) \oplus res\text{-}io\text{-}ch(i_p \ominus i_s, \mathsf{Chs}(S)). \end{aligned}$$

The equality above depends on such a summation comprising exactly the o_s channels ($\mathsf{Chs}_o(S)$). Since o_s has already been included in this summation, it is enough to check that the other two elements do not include any channel apart from those in o_s. This analysis is performed in terms of channels where i_s is given as $\mathsf{Chs}_i(S)$, o_s is $\mathsf{Chs}_o(S)$, i_p is $\mathsf{Chs}_i(P)$, and o_p is $\mathsf{Chs}_o(P)$. The analysis is as follows:

- $i_p \ominus i_s$ ($\mathsf{Chs}_i(P) - \mathsf{Chs}_i(S)$): by definition, the sets of input and output channels of a single process are disjoint (($\mathsf{Chs}_i(S) \cap \mathsf{Chs}_o(S)) = \varnothing$). Also, channel restricted similarity relates input and output channels separately. So, these input actions could only be related to S's input actions. However, $(\mathsf{Chs}_i(P) - \mathsf{Chs}_i(S)) \cap \mathsf{Chs}(S) = \varnothing$ and $o_s \oplus \epsilon = o_s$ as required;
- o_p ($\mathsf{Chs}_o(P)$): o_p could only coincide with o_s because channel restricted similarity matches input or output channels separately:
  1. If $\mathsf{Chs}_o(P) \cap \mathsf{Chs}_o(S) = \varnothing$, then $res\text{-}ch(o_p, \mathsf{Chs}(S)) = \epsilon$ and $\epsilon \oplus o_s = o_s$ as required;
  2. If $\mathsf{Chs}_o(P) \cap \mathsf{Chs}_o(S) \neq \varnothing$, then $(\mathsf{dom}\, res\text{-}ch(o_p, \mathsf{Chs}(S))) \subseteq (\mathsf{dom}\, o_s)$ and $res\text{-}ch(o_p, \mathsf{Chs}(S)) \oplus o_s = o_s$ as required;

Equality of Input Actions

$$\begin{aligned} i_s &= res\text{-}io\text{-}ch((i_x \oplus i_p) \ominus (o_x \oplus o_p), \mathsf{Chs}(S)) \\ &= res\text{-}io\text{-}ch((i_s \oplus o_p \oplus i_p) \ominus (((o_s \ominus o_p) \oplus (i_p \ominus i_s)) \oplus o_p), \mathsf{Chs}(S)) \\ &= res\text{-}io\text{-}ch(i_s \ominus ((o_s \ominus o_p) \oplus (i_p \ominus i_s) \oplus o_p), \mathsf{Chs}(S)) \oplus \\ &\quad\; res\text{-}io\text{-}ch(o_p \ominus ((o_s \ominus o_p) \oplus (i_p \ominus i_s) \oplus o_p), \mathsf{Chs}(S)) \oplus \\ &\quad\; res\text{-}io\text{-}ch(i_p \ominus ((o_s \ominus o_p) \oplus (i_p \ominus i_s) \oplus o_p), \mathsf{Chs}(S)) \\ &= res\text{-}io\text{-}ch(i_s \ominus ((o_s \ominus o_p) \oplus (i_p \ominus i_s) \oplus o_p), \mathsf{Chs}(S)) \oplus \\ &\quad\; res\text{-}io\text{-}ch(\epsilon, \mathsf{Chs}(S)) \oplus \\ &\quad\; res\text{-}io\text{-}ch(i_p \ominus ((o_s \ominus o_p) \oplus (i_p \ominus i_s) \oplus o_p), \mathsf{Chs}(S)) \\ &= res\text{-}io\text{-}ch(i_s \ominus ((o_s \ominus o_p) \oplus (i_p \ominus i_s) \oplus o_p), \mathsf{Chs}(S)) \\ &\quad\; \epsilon \oplus \\ &\quad\; res\text{-}io\text{-}ch(i_p \ominus ((o_s \ominus o_p) \oplus (i_p \ominus i_s) \oplus o_p), \mathsf{Chs}(S)) \end{aligned}$$

This equality is held only if summation of all restricted elements comprise all i_s channels but no other. Once more, the equality is analysed at the channels level:

- $(i_s \ominus ((o_s \ominus o_p) \oplus (i_p \ominus i_s)) \oplus o_p)$: by definition of processes in EPA, $(\mathsf{Chs}_i(S) \cap \mathsf{Chs}_o(S)) = \emptyset$ and, by Lemma 4.2, $(\mathsf{Chs}_i(S) \cap \mathsf{Chs}_o(P)) = \emptyset$ and $(\mathsf{Chs}_o(S) \cap \mathsf{Chs}_i(P)) = \emptyset$. So, this expression is reduced to $(i_s \ominus (i_p \ominus i_s)) = i_s$ and $\mathit{res\text{-}ch}(i_s, \mathsf{Chs}(S)) = i_s$ as required;
- $i_p \ominus ((o_s \ominus o_p) \oplus (i_p \ominus i_s)) \oplus o_p)$: this expression is reduced to $i_p \ominus (i_p \ominus i_s)$. It might basically have one of the following relations:
  1. $\mathsf{Chs}_i(P) \cap \mathsf{Chs}_i(S) = \emptyset$, then $\mathit{res\text{-}ch}(i_p \ominus (i_p \ominus i_s), \mathsf{Chs}(S)) = \epsilon$ and $\epsilon \oplus i_s = i_s$ as required;
  2. or $\mathsf{Chs}_i(P) \cap \mathsf{Chs}_i(S) \neq \emptyset$, then $(\mathsf{dom}\, \mathit{res\text{-}ch}(i_p \ominus (i_p \ominus i_s), \mathsf{Chs}(S))) \subseteq (\mathsf{dom}\, i_s)$ as required because the expression is reduced to elements of i_p which coincide with i_s.

C.2.2 Section 5.4

Restriction to Channels

$$\begin{array}{ccc} (i) & = & (ii) \\ \mathit{res\text{-}ch}(i_s/o_s \div i_p/o_p, C) & = & \mathit{res\text{-}ch}(i_s/o_s, C) \div \mathit{res\text{-}ch}(i_p/o_p, C) \end{array}$$

$$\begin{aligned} (i) &= \mathit{res\text{-}ch}(i_s/o_s \div i_p/o_p, C) \\ &= \mathit{res\text{-}io\text{-}ch}((i_s \oplus o_p)/(o_s \ominus o_p) \oplus (i_p \ominus i_s), C) \\ &= (\mathit{res\text{-}io\text{-}ch}(i_s \oplus o_p, C))/(\mathit{res\text{-}io\text{-}ch}((o_s \ominus o_p) \oplus (i_p \ominus i_s), C)) \\ &= (\mathit{res\text{-}io\text{-}ch}(i_s, C) \oplus \mathit{res\text{-}io\text{-}ch}(o_p, C))/ \\ &\quad ((\mathit{res\text{-}io\text{-}ch}(o_s, C) \ominus \mathit{res\text{-}io\text{-}ch}(o_p, C)) \oplus (\mathit{res\text{-}io\text{-}ch}(i_p, C) \ominus \mathit{res\text{-}io\text{-}ch}(i_s, C))) \\ &= \mathit{res\text{-}io\text{-}ch}(i_s/o_s, C) \div \mathit{res\text{-}io\text{-}ch}(i_p/o_p, C) \\ &= (ii) \end{aligned}$$

Hide

$$\begin{array}{ccc} (i) & = & (ii) \\ \mathit{hide\text{-}act}(i_s/o_s \div i_p/o_p, C) & = & \mathit{hide\text{-}act}(i_s/o_s, C) \div \mathit{hide\text{-}act}(i_p/o_p, C) \end{array}$$

$$\begin{aligned} (i) &= \mathit{hide\text{-}act}(i_s/o_s \div i_p/o_p, C) \\ &= \mathit{hide\text{-}io\text{-}act}((i_s \oplus o_p)/(o_s \ominus o_p) \oplus (i_p \ominus i_s), C) \\ &= (\mathit{hide\text{-}io\text{-}act}(i_s \oplus o_p, C))/(\mathit{hide\text{-}io\text{-}act}((o_s \ominus o_p) \oplus (i_p \ominus i_s), C)) \\ &= (\mathit{hide\text{-}io\text{-}act}(i_s, C) \oplus \mathit{hide\text{-}io\text{-}act}(o_p, C))/ \\ &\quad ((\mathit{hide\text{-}io\text{-}act}(o_s, C) \ominus \mathit{hide\text{-}io\text{-}act}(o_p, C)) \oplus \\ &\quad (\mathit{hide\text{-}io\text{-}act}(i_p, C) \ominus \mathit{hide\text{-}io\text{-}act}(i_s, C))) \\ &= \mathit{hide\text{-}io\text{-}act}(i_s/o_s, C) \div \mathit{hide\text{-}io\text{-}act}(i_p/o_p, C) \\ &= (ii) \end{aligned}$$

Composition Law 1

$$\begin{array}{ccc} (i) & = & (ii) \\ (i_{s1}/o_{s1} \div (i_{p1}/o_{p1} \star i_{p2}/o_{p2})) & = & ((i_{s1}/o_{s1} \div i_{p1}/o_{p1}) \div i_{p2}/o_{p2}) \end{array}$$

$$\begin{aligned} (i) &= (i_{s1}/o_{s1} \div (i_{p1}/o_{p1} \star i_{p2}/o_{p2})) \\ &= (i_{s1}/o_{s1} \div ((i_{p1} \oplus i_{p2}) \ominus (o_{p1} \oplus o_{p2})/(o_{p1} \oplus o_{p2}))) \\ &= (i_{s1} \oplus (o_{p1} \oplus o_{p2}))/((o_{s1} \ominus (o_{p1} \oplus o_{p2})) \oplus ((i_{p1} \oplus i_{p2}) \ominus (o_{p1} \oplus o_{p2}) \ominus i_{s1})) \\ &= (i_{s1} \oplus o_{p1} \oplus o_{p2})/((o_{s1} \ominus o_{p1} \ominus o_{p2}) \oplus (i_{p1} \oplus i_{p2} \ominus i_{s1})) \end{aligned}$$

$$
\begin{aligned}
(ii) &= ((i_{s1}/o_{s1} \div i_{p1}/o_{p1}) \div i_{p2}/o_{p2}) \\
&= (i_{s1} \oplus o_{p1})/((o_{s1} \ominus o_{p1}) \oplus (i_{p1} \ominus i_{s1})) \div i_{p2}/o_{p2}) \\
&= (i_{s1} \oplus o_{p1} \oplus o_{p2})/((o_{s1} \ominus o_{p1}) \oplus (i_{p1} \ominus i_{s1}) \ominus o_{p2}) \oplus (i_{p2} \ominus (i_{s1} \oplus o_{p1})) \\
&= (i_{s1} \oplus o_{p1} \oplus o_{p2})/((o_{s1} \ominus o_{p1} \ominus o_{p2}) \oplus (i_{p1} \ominus i_{s1}) \oplus (i_{p2} \ominus i_{s1})) \\
&= (i_{s1} \oplus o_{p1} \oplus o_{p2})/((o_{s1} \ominus o_{p1} \ominus o_{p2}) \oplus (i_{p1} \oplus i_{p2} \ominus i_{s1})) \\
&= (i)
\end{aligned}
$$

Composition Law 2

$$
\begin{array}{ccc}
(i) & = & (ii) \\
((i_{s1}/o_{s1} \star i_{s2}/o_{s2}) \div i_p/o_p) \star i_p/o_p & = & ((i_{s1}/o_{s1} \div i_p/o_p) \star i_{s2}/o_{s2}) \star i_p/o_p
\end{array}
$$

$$
\begin{aligned}
(i) &= ((i_{s1}/o_{s1} \star i_{s2}/o_{s2}) \div i_p/o_p) \star i_p/o_p \\
&= (((i_{s1} \oplus i_{s2}) \ominus (o_{s1} \oplus o_{s2}))/(o_{s1} \oplus o_{s2}) \div i_p/o_p) \star i_p/o_p \\
&= (((i_{s1} \oplus i_{s2}) \ominus (o_{s1} \oplus o_{s2})) \oplus o_p)/ \\
&\quad ((o_{s1} \oplus o_{s2} \ominus o_p) \oplus (i_p \ominus ((i_{s1} \oplus i_{s2}) \ominus (o_{s1} \oplus o_{s2})))) \star i_p/o_p \\
&= (((i_{s1} \oplus i_{s2}) \ominus (o_{s1} \oplus o_{s2})) \oplus o_p \oplus i_p) \ominus \\
&\quad ((o_{s1} \oplus o_{s2} \ominus o_p) \oplus (i_p \ominus (i_{s1} \oplus i_{s2})) \oplus o_p)/ \\
&\quad ((o_{s1} \oplus o_{s2} \ominus o_p) \oplus (i_p \ominus (i_{s1} \oplus i_{s2})) \oplus o_p) \\
&= ((i_{s1} \oplus i_{s2}) \ominus (o_{s1} \oplus o_{s2}))/((o_{s1} \oplus o_{s2} \oplus o_p) \oplus (i_p \ominus (i_{s1} \oplus i_{s2})))
\end{aligned}
$$

$$
\begin{aligned}
(ii) &= ((i_{s1}/o_{s1} \div i_p/o_p) \star i_{s2}/o_{s2}) \star i_p/o_p \\
&= ((i_{s1} \oplus o_p)/(o_{s1} \ominus o_p) \oplus (i_p \ominus i_{s1}) \star i_{s2}/o_{s2}) \star i_p/o_p \\
&= (((i_{s1} \oplus o_p) \oplus i_{s2}) \ominus ((o_{s1} \ominus o_p) \oplus (i_p \ominus i_{s1}) \oplus o_{s2}))/ \\
&\quad ((o_{s1} \ominus o_p) \oplus ((i_p \ominus i_{s1}) \oplus o_{s2})) \star i_p/o_p \\
&= (((i_{s1} \oplus o_p) \oplus i_{s2} \oplus i_p) \ominus ((o_{s1} \ominus o_p) \oplus (i_p \ominus i_{s1}) \oplus o_{s2}) o_p)/ \\
&\quad ((o_{s1} \ominus o_p) \oplus ((i_p \ominus i_{s1}) \oplus o_{s2}) \oplus o_p) \\
&= ((i_{s1} \oplus i_{s2}) \ominus (o_{s1} \oplus o_{s2}))/((o_{s1} \oplus o_{s2} \oplus o_p) \oplus (i_p \ominus i_{s1})) \\
&= (i) \text{ considering the side condition } \mathsf{Chs}_i(S_1) \cap \mathsf{Chs}_i(P) = \mathsf{Chs}_i(S_2) \cap \mathsf{Chs}_i(P)
\end{aligned}
$$

C.2.3 Simulation vs. Substitutivity

Despite a simulation process behaving bisimilarly to the simulated one for all the actions performed by the simulated process, simulation does not ensure substitutivity because summation and composition do not preserve simulation.

Suppose processes S_+, P_+ and R_+ defined over identical sets of channels[1]:

$$
\begin{aligned}
&\mathsf{process}\ S_+ \ \underline{\triangle}\ a/x::\mathsf{Stop} + b/y::\mathsf{Stop} \\
&\mathsf{process}\ P_+ \ \underline{\triangle}\ a/x::\mathsf{Stop} + b/y::\mathsf{Stop} + c/z::\mathsf{Stop} \\
&\mathsf{process}\ R_+ \ \underline{\triangle}\ d/w::R_+ + c/z::R_+,
\end{aligned}
$$

It is trivially checked that P_+ is a simulation of S_+. However, the process term $(R_+ + S_+)$ cannot simulate $(R_+ + S_+)$:

$$
\begin{aligned}
&\mathsf{process}\ (S_+ + R_+)\ \underline{\triangle}\ a/x::\mathsf{Stop} + b/y::\mathsf{Stop} + d/w::R_+ + c/z::R_+ \\
&\mathsf{process}\ (P_+ + R_+)\ \underline{\triangle}\ a/x::\mathsf{Stop} + b/y::\mathsf{Stop} + c/z::\mathsf{Stop} + d/w::R_+ + c/z::R_+.
\end{aligned}
$$

[1] The channel names are omitted in these examples to avoid unnecessary details.

Notice that despite doing all moves that $(R_+ + S_+)$ does, $(R_+ + P_+)$ is nondeterministic for the input action c, while $(S_+ + R_+)$ is deterministic. As a result, $(R_+ + P_+)$ cannot simulate $(R_+ + S_+)$.

Proposition C.1 *Simulation is not preserved by Summation.*

Due to the implicitly hiding of channels performed by the EPA composition operator, certain "spare actions" of a simulation process could eventually be combined with actions from another process and be identical, in inputs, to other compound actions. Suppose processes $S_{|}$, $P_{|}$ and $R_{|}$ defined as follows:

$$
\begin{aligned}
\textsf{process } S_{|} \;&\underline{\triangle}\; (c_1(0), c_2(0)/c_3(1))\!::\!\textsf{Stop} + (c_1(0), c_2(1)/c_3(0))\!::\!\textsf{Stop}\\
\textsf{process } P_{|} \;&\underline{\triangle}\; (c_1(0), c_2(0)/c_3(1))\!::\!\textsf{Stop} + (c_1(0), c_2(1)/c_3(0))\!::\!\textsf{Stop}+\\
&\quad (c_1(1), c_2(0)/c_3(1))\!::\!\textsf{Stop}\\
\textsf{process } R_{|} \;&\underline{\triangle}\; (c_2(0), c_5(1)/c_1(0), c_6(0))\!::\!\textsf{Stop} + (c_2(1), c_5(0)/c_1(0), c_6(1))\!::\!\textsf{Stop}+\\
&\quad (c_2(0), c_5(1)/c_1(1), c_6(1))\!::\!\textsf{Stop}.
\end{aligned}
$$

$P_{|}$ is a simulation of $S_{|}$ and composition of these similar processes with $R_{|}$ are as follow:

$$
\begin{aligned}
\textsf{process } S_{|} \mid R_{|} \;&\underline{\triangle}\; (c_2(0), c_5(1)/c_3(1), c_1(0), c_6(0))\!::\!\textsf{Stop} +\\
&\quad (c_2(1), c_5(0)/c_3(0), c_1(0), c_6(1))\!::\!\textsf{Stop}\\
\textsf{process } P_{|} \mid R_{|} \;&\underline{\triangle}\; (c_2(0), c_5(1)/c_3(1), c_1(0), c_6(0))\!::\!\textsf{Stop} +\\
&\quad (c_2(1), c_5(0)/c_3(0), c_1(0), c_6(1))\!::\!\textsf{Stop} +\\
&\quad (c_2(0), c_5(1)/c_3(1), c_1(1), c_6(1))\!::\!\textsf{Stop}
\end{aligned}
$$

Composition of the $P_{|}$'s "spare action" $(c_1(1), c_2(0)/c_3(1))$ with $R_{|}$ results in action $(c_2(0), c_5(1)/c_3(1), c_1(1), c_6(1))$. Besides that, action $(c_2(0), c_5(1)/c_3(1), c_1(0), c_6(0))$ is also a result from composition of $R_{|}$ with $P_{|}$. Notice that process $(S_{|} \mid R_{|})$ has a deterministic choice for input action $c_2(0), c_5(1)$ while $(P_{|} \mid R_{|})$ is nondeterministic for the same input. These compound processes are obviously incomparable under simulation and then,

Proposition C.2 *Simulation is not preserved by Composition.*

Since simulation is not preserved by all combinators, a simulation process cannot substitute for the simulated one because, in general,

$$S \sqsubseteq P \text{ but } S + R \not\sqsubseteq P + R \text{ and } S \mid R \not\sqsubseteq P \mid R.$$

In fact, simulation is preserved by all EPA combinators except summation and composition:

Proposition C.3 *Simulation is preserved by* prefixing, hiding, restriction to channels *and* restriction to actions *combinators*[2]*; that is, if* $S \sqsubseteq P$ *then* $(a :: S) \sqsubseteq (a :: P)$, $(S \setminus C) \sqsubseteq (P \setminus C)$, $(S \upharpoonright_c C) \sqsubseteq (P \upharpoonright_c C)$ *and* $(S \upharpoonright_a \mathcal{A}) \sqsubseteq (P \upharpoonright_a \mathcal{A})$.

[2] A proof for this proposition is trivial; it is given by the semantic definition of the referred operators. We shall then omit such a proof.

Appendix D

Transition Graphs – TLCs

D.1 Traffic Light Controllers – Transition Graphs

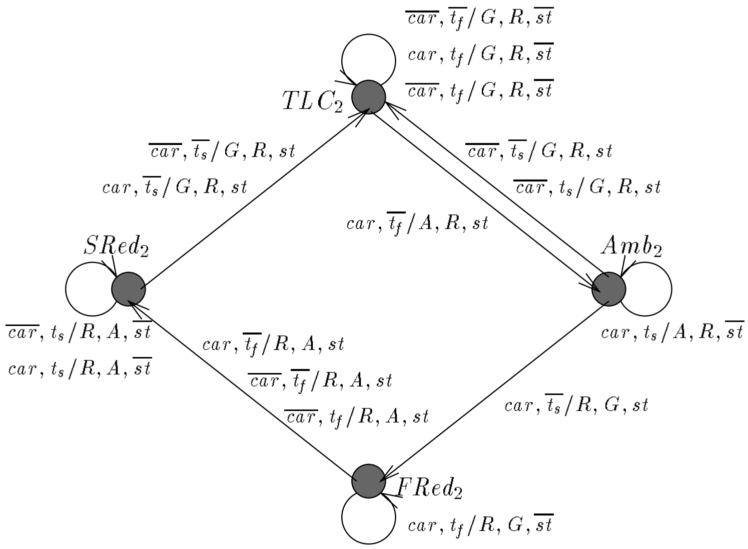

Figure D.1: Transition Graph for $TLC_2(s, t/l_{mj}, l_{mn}, setc)$

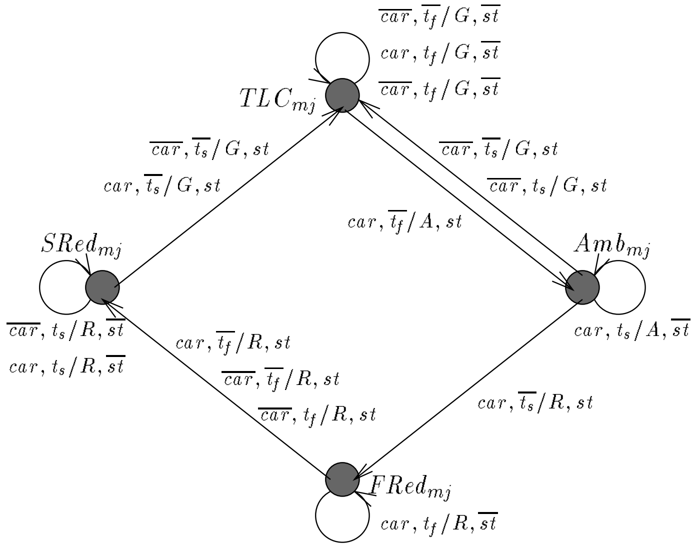

Figure D.2: Transition Graph for $TLC_{mj}(s, t/l_{mj}, setc)$

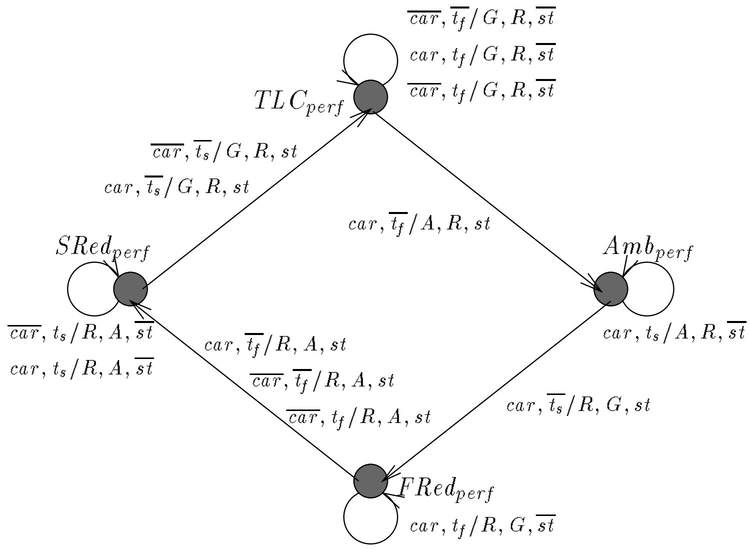

Figure D.3: Transition Graph for $TLC_{perf}(s, t/l_{mj}, l_{mn}, setc)$

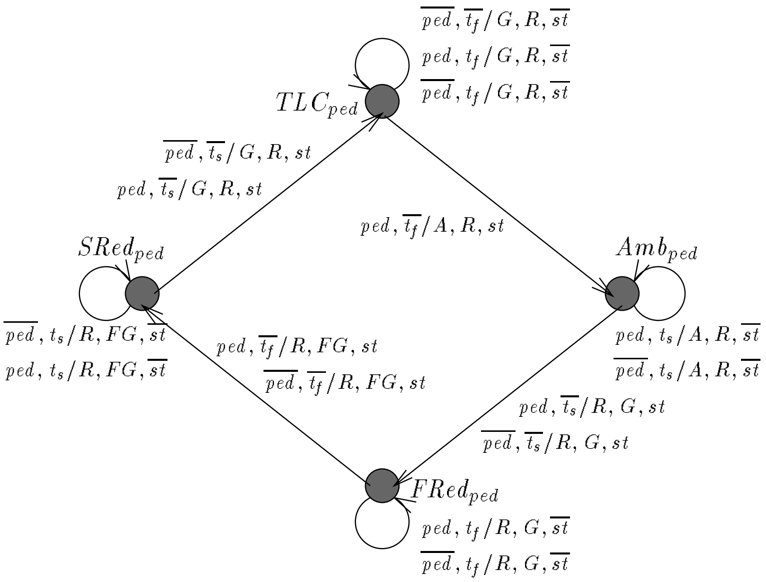

Figure D.4: Transition Graph for $TLC_{ped}(b, t/l_r, l_{pd}, setc)$

D.2 Interface Processes

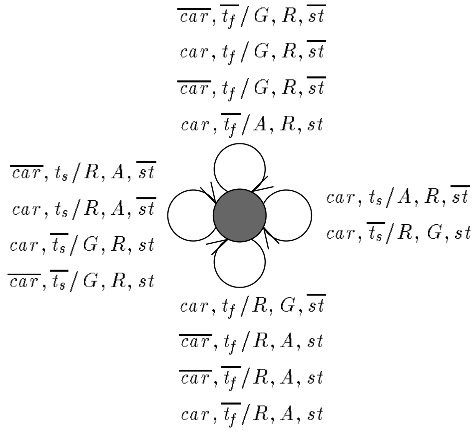

Figure D.5: An Interface Process for Implementing TLC_{perf} using TLC_2

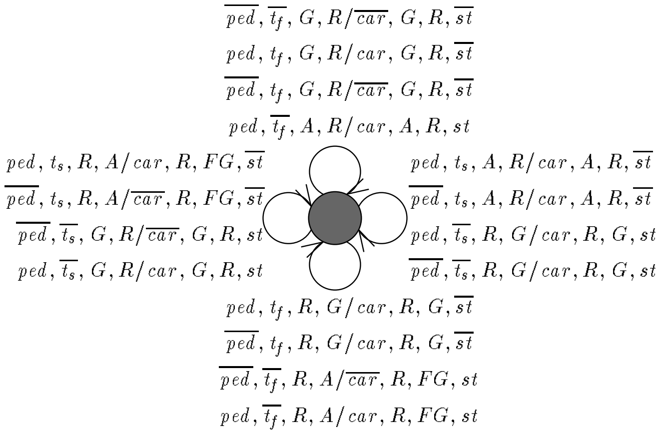

Figure D.6: An Interface Process for Implementing TLC_{ped} using TLC_2

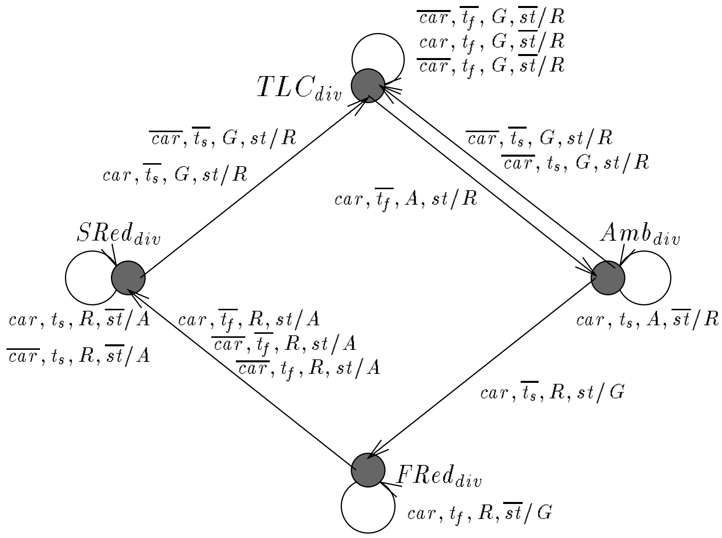

Figure D.7: Interface between the Traffic Light Controller and the Major Road Process

D.3 Minimal Processes

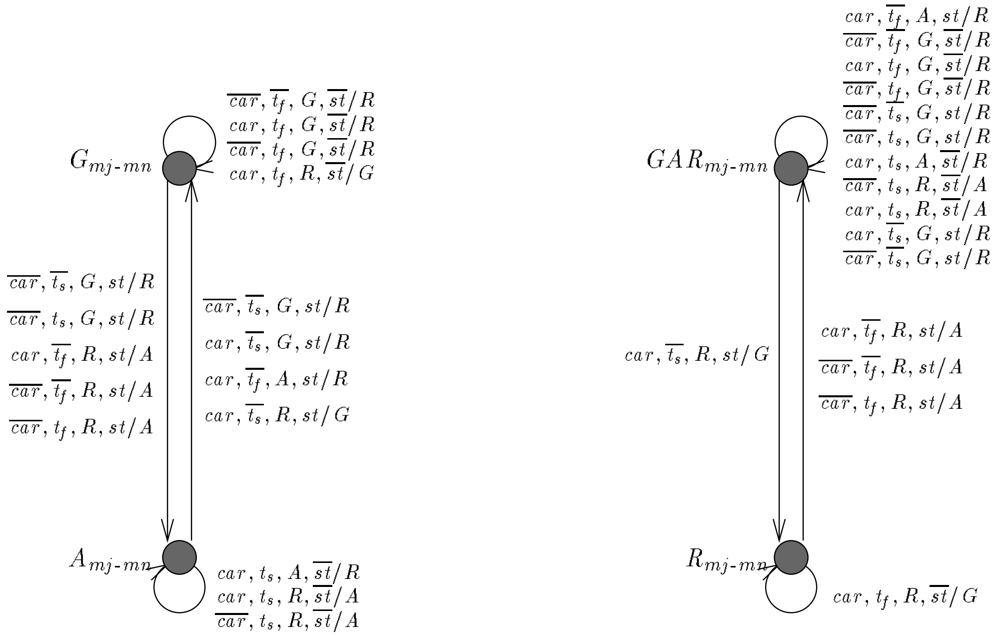

Figure D.8: Minimization of States of the Traffic Light Interface Process

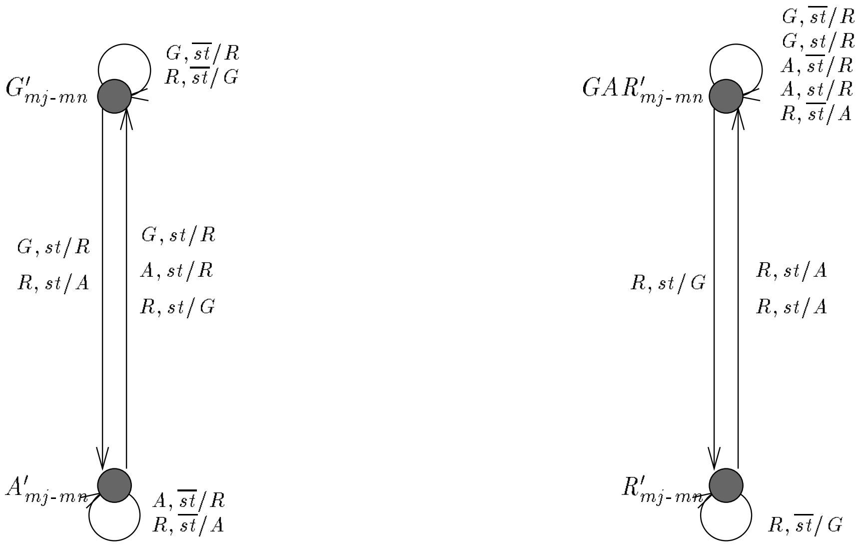

Figure D.9: The Minimal Interface Processes for the TLC

Bibliography

[Aea71] Mark A. Aiserman and et al. *Logic, Automata, and Algorithms.* Academic Press, first edition, 1971.

[AH91] L. Aceto and M. Hennessy. Adding action refinement to a Finite Process Algebra. In J. L. Albert, B. Monien, and M. R. Artalejo, editors, *Automata, Languages and Programming – 18th International Colloquium*, volume 510 of *LNCS*. Springer–Verlag, 1991.

[Arn94] André Arnold. *Finite Transition Systems.* International Series in Computer Science. Prentice–Hall, 1994.

[Bar93] J. E. Barnes. *A Mathematical Theory of Synchronous Communication.* PhD thesis, Oxford University, 1993.

[BB90] J. C. M. Baeten and J. A. Bergstra. Process algebra with a Zero Object. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR'90 – Theories of Concurrency: Unification and Extension*, volume 458 of *LNCS*. Springer–Verlag, 1990.

[BGMW92] H. Barringer, G. Gough, B. Monahan, and A. Williams. A semantics and verification framework for ELLA. Technical Report UMCS–92–4–6, University of Manchester, 1992.

[BGMW93] H. Barringer, G. Gough, B. Monahan, and A. Williams. An algebraic framework for action and process. D2.3c, Formal Verification Support for ELLA, IED project 4/1/1357, University of Manchester, May 1993.

[BGMW94] H. Barringer, G. Gough, B. Monahan, and A. Williams. A process algebra foundation for reasoning about core ELLA. Technical Report 94–12–1, University of Manchester, Dec 1994.

[BM79] R. S. Boyer and J. S. Moore. *A Computational Logic.* New York – Academic Press, 1979.

[BNR93] H. Busch, H. Nusser, and T. Rössel. Formal methods for synthesis. In P. Michael, U. Lauther, and P. Duzy, editors, *The Synthesis Approach to Digital System Design.* Kluwer Academic Publishers, 1993.

[Bou85] G. Boudol. *Notes on Algebraic Calculi of Processes, Logic and Models of Concurrent Systems.* NATO ASI. 1985.

[BP89a] T. J. Biggerstaff and A. J. Perlis. *Software Reusability – Applications and Experience*, volume 2. Addison–Wesley Publishing Company, first edition, 1989.

[BP89b] T. J. Biggerstaff and A. J. Perlis. *Software Reusability – Concepts and Models*, volume 1. Addison–Wesley Publishing Company, first edition, 1989.

[BR89] T. J. Biggerstaff and C. Richter. Reusability framework, assessment, and directions. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability.* Addison–Wesley Publishing Company, 1989.

[Cav89] Michael J. Cavaliere. Reusable code at the Hartford Insurance Group. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability – Applications and Experience*, volume 2. Addison–Wesley Publishing Company, 1989.

[CFG91] M. L. Creech, D. F. Freeze, and M. L. Griss. Using hypertext in selecting reusable software components. In *Hypertext'91 – Third ACM Conference on Hypertext*. ACM Press, 1991.

[CL89] John Carroll and Darrell Long. *Theory of Finite Automata*. Pentice Hall, first edition, 1989.

[Dev91] Srinivas Devadas. Optimizing interacting finite state machines using sequential don't cares. *IEEE Transactions on Computer–Aided Design*, December 1991.

[dS85] R. de Simone. Higher–level synchronising devices in Meije–SCCS. *Journal of Theoretical Computer Science*, 37, 1985.

[DS92] J. Davies and S. Schneider. A brief history of timed CSP. Technical Report Technical Monograph – PRG –96, Oxford University, April 1992.

[Fer90] Jean-Claude Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2–3), May 1990.

[FM91] Jean-Claude Fernandez and L. Mounier. On the Fly verification of behavioural equivalences and preorders. In K. G. Larsen and A. Skou, editors, *Computer Aided Verification (CAV'91)*, volume 757 of *LNCS*. Springer–Verlag, 1991.

[Fre87a] P. Freeman. A perspective on reusability. In Peter Freeman, editor, *Tutorial: Software Reusability*. IEEE – The Computer Society Press, 1987.

[Fre87b] Peter Freeman. *Tutorial: Software Reusability*. IEEE – The Computer Society Press, 1987.

[Gil62] Arthur Gill. *Introduction to the Theory of Finite–State Machines*. McGraw–Hill Book Company, first edition, 1962.

[GM88] M. C. Gaudel and T. Moineau. A theory of software reusability. In *ESOP'88*, volume 300 of *LNCS*. Springer Verlag, 1988.

[GM92] J. F. Groote and F. Moller. Verification of parallel systems via decomposition. In W. R. Cleaveland, editor, *CONCUR'92*, volume 630 of *LNCS*. Springer–Verlag, 1992.

[GM93] M. Gordon and T. F. Melham. *An Introduction to HOL*. Cambridge University Press, 1993.

[Gog89] Joseph A. Goguen. Principles of parameterized programming. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability – Concepts and Models*, volume 1. Addison–Wesley Publishing Company, 1989.

[Gor86] M. J. Gordon. Why higher–order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*. North–Holland, 1986.

[GS89a] P. K. Garg and W. Scacchi. A hypertext system to manage software life cycle documents. *IEEE Software*, 1989.

[GS89b] P. K. Garg and W. Scacchi. On designing intelligent software hypertext system. *IEEE Software Engineering*, 1989.

[HC91] J. W. Hooper and R. O. Chester. *Software Reuse – Guidelines and Methods*. Software Science and Engineering. Plenum Press, 1991.

[Hen88] Matthew Hennessy. *Albegraic Theory of Processes*. The MIT Press, first edition, 1988.

[HI91] M. Hennessy and A. Ingolfsdottir. A theory of communicating process with value–passing. Technical Report 7/91, University of Sussex, August 1991.

[HL92] M. Hennessy and H. Lin. Symbolic bisimulations. Technical Report 1/92, University of Sussex, April 1992.

[HM87] E. Horowitz and J. B. Munson. An expansive view of software reuse. In Peter Freeman, editor, *Tutorial: Software Reusability.* IEEE – The Computer Society Press, 1987.

[HM89] E. Horowitz and J. B. Munson. An expansive view of reusable software. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability.* Addison–Wesley Publishing Company, 1989.

[Hoa81] C. A. R. Hoare. A model for communicating sequential processes. Technical Report TM–PRG–22, Oxford University Computing Laboratory, June 1981.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice–Hall, first edition, 1985.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison–Wesley, first edition, 1979.

[Jon87] T. C. Jones. Reusability in programming: A survey of the state of the art. In Peter Freeman, editor, *Tutorial: Software Reusability.* IEEE – The Computer Society Press, 1987.

[KN72] Joonki Kim and Monroe M. Newborn. The simplification of sequential machines with input restrictions. *IEEE Transactions on Computers*, C21, December 1972.

[KRT89] Shmuel Katz, Charles A. Richer, and Khe-Sing The. PARIS: A system for reusing partially interpreted schemas. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability – Concepts and Models*, volume 1. Addison–Wesley Publishing Company, 1989.

[KS90] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite states processes, and three problems of equivalence. *Information and Computation*, 86, 1990.

[LMS86] R. Lipsett, E. Marschner, and M. Shahdad. VHDL – the language. *IEEE– Test and Design*, April 1986.

[Mat87] Y. Matsumoto. A software factory: An overall approach to software production. In Peter Freeman, editor, *Tutorial: Software Reusability.* IEEE – The Computer Society Press, 1987.

[MC93] J D Morison and A S Clarke. *ELLA2000: A Language for Electronic System Design.* McGraw-Hill, 1993.

[McI76] M. D. McIlroy. Mass–produced software components. In J. M. Buxton, P. Naur, and B. Randell, editors, *Software Eng. Concepts and Techniques –1968 NATO Conf. Software Eng.* Petrocelli/Charter, 1976.

[Mey88] Bertrand Meyer. *Object–oriented Software Construction.* Pentice Hall –Int. Ser. In Comp. Sci., first edition, 1988.

[Mey89] Bertrand Meyer. Reusability: The case for object–oriented design. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability – Applications and Experience*, volume 2. Addison–Wesley Publishing Company, 1989.

[Mey94] Bertrand Meyer. *Reusable Software – The base object–oriented component libraries.* The Object–Oriented Series. Prentice Hall, 1994.

[Mil71] Robin Milner. An algebraic definition of simulation between programs. In *Proc. 2nd Int. Joint Conf. on Artificial Inteligence.* British Comp. Soc., 1971.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer–Verlag, 1980.

[Mil85] George Milne. Circal and the representation of communication, concurrency and time. *ACM Trans. Prog. Langs. and Sys.*, 7, 1985.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice–Hall, first edition, 1989.

[Mil91] Robin Milner. Polyadic π–calculus: A tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

[Mil93] Robin Milner. An action structure for synchronous π–calculus. Technical report, University of Edinburgh, 1993.

[Mil94] George Milne. *Formal Specification and Verification of Digital Systems*. McGraw–Hill, first edition, 1994.

[MS92] J. D. McGregor and D. A. Sykes. *Object–Oriented Software Development: Engineering Software for Reuse*. Van Nostrand Reinhold, 1992.

[Nag92] N. S. Nagaraj. OPSYN – OASYS based pseudosynthesis tool. In *VLSI Design 1992 – The fifth International Conference on VLSI Design*. IEEE – The Computer Society Press, 1992.

[Nei89] J. M. Neighbors. Draco: A method for engineering reusable software systems. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*. Addison–Wesley Publishing Company, 1989.

[Par81] David Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science – 5th GI–Conference*, volume 104 of *LNCS*. Spring–Verlag, 1981.

[Par89] Joachim Parrow. Submodule construction as equation solving in CCS. *Theoretical Computer Science*, 68:175–202, 1989.

[Pau92] L. C. Paulson. *ML for the Working Programmer*. Cambridge Univ. Press, 1992.

[PD89] Rúben Prieto-Díaz. Classification of reusable modules. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability – Applications and Experience*, volume 1. Addison–Wesley Publishing Company, 1989.

[PL89] Noah S. Prywes and Evan D. Lock. Use of the model equational language and program generator by management professionals. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability – Applications and Experience*, volume 2. Addison–Wesley Publishing Company, 1989.

[Pom85] Lucia Pomello. Some equivalence notions for concurrent systems. an overview. In *Advances in Petri Nets 1985*, volume 222 of *LNCS*. Spring–Verlag, 1985.

[PP83] N. S. Prywes and A. Pnueli. A compilation of non–procedural specification into computer programs. *IEEE Trans. Soft. Eng.*, SE–9, May 1983.

[PPS79] N. S. Prywes, A. Pnueli, and S. Shastry. Use of non–procedural specification language and associated program generator in software development. *ACM Trans., Prog. Lang. Syst*, October 1979.

[PT87] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6), December 1987.

[PW87] Franklin P. Prosser and David E. Winkel. *The art of Digital Design*. Pentice Hall International Editions, second edition, 1987.

[QL91] Huajan Qin and Philip Lewis. Factorization of finite state machines under observational equivalence. In J. C. M. Baeten and J. F. Groote, editors, *CONCUR'91*, volume 527 of *LNCS*. Spring–Verlag, 1991.

[RHS91] JK. Rho, G. Hachtel, and F. Somenzi. Don't care sequences and the optimization of interacting finite state machines. In *Computer–Aided Design 91*. IEEE, 1991.

[Ros88] A. W. Roscoe. Two papers on CSP. Technical Report TM–PRG–67, Oxford University, July 1988.

[Shi87] M. W. Shields. *An Introduction to Automata Theory*. Blackwell Scientific Publications, first edition, 1987.

[Shi89] M. W. Shields. Implicit system specification and the interface equation. *The Computer Journal*, 32(5), 1989.

[SPDM94] W. Schäfer, R. Prieto-Díaz, and M. Matsumoto. *Software Reusability*. Ellis Horwood, 1994.

[TM91] D. E. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.

[Ung69] Stephen H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley–Interscience, first edition, 1969.

[vG90] R. J. van Glabbeek. The Linear Time – Branching Time Spectrum. In *CONCUR'90*, volume 458 of *LNCS*. Spring–Verlag, 1990.

[vG93] R. J. van Glabbeek. The Linear Time – Branching Time Spectrum II. In *CONCUR'93*, volume 715 of *LNCS*. Spring–Verlag, 1993.

[VK89] Dennis M. Volpano and Richard B. Kieburtz. The templates approach to software reuse. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability – Concepts and Models*, volume 1. Addison–Wesley Publishing Company, 1989.

[Weg87] Peter Wegner. Varieties of reusability. In Peter Freeman, editor, *Tutorial: Software Reusability*. IEEE – The Computer Society Press, 1987.

[Weg89] Peter Wegner. Capital–intensive software technology. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability – Applications and Experience*, volume 1. Addison–Wesley Publishing Company, 1989.

[WHS89] M. Wirsing, R. Hennicker, and R. Stahl. Menu – an example for the systematic reuse of specifications. In *2nd European Software Engineering Conference*, volume 387 of *LNCS*. Springer Verlag, 1989.

[Win83] Glynn Winskel. Event structure semantics for CCS and related languages. Technical Report DAIMI PB-159, Aarhus University, April 1983.