Universitat Ramon Llull

TESI DOCTORAL

Títol **Contributions to Formal Communication Elimination for System Models with Explicit Parallelism**

Realitzada per **Francesc-Xavier Babot Pagès**

en el Centre **La Salle. Universitat Ramon Llull**

i en el Departament **d'Informàtica**

Dirigida per **Prof. Miquel Bertran Salvans**

C. Claravall, 1-3
08022 Barcelona
Tel. 936 022 200
Fax 936 022 249
E-mail: urlsc@sec.url.es
www.url.es

RESUM

Els mètodes de *verificació formal* s'estan usant cada vegada més en la indústria per establir la correctessa i trobar els errors en *models de sistemes*; per exemple la descripció de hardware, protocols, programes distribuïts, etc. En particular, els verificadors de models ho fan automàticament per sistemes d'estats finits, però estan limitats degut al problema de l'explosió d'estats; i la *verificació formal interactiva*, l'àrea d'aquesta tesi, es necessita.

L'enfocament de la verificació automàtica treballa sobre el sistema de transicions del model, el qual defineix la seva semàntica. Aquest sistema de transicions té sovint molts estats, i sempre una mida gran comparada amb la mida del model del sistema, el qual és sempre infinit. Aquestes consideracions suggereixen un enfocament de *verificació estàtica* com els d'aquesta tesi, evitant els sistemes de transicions, treballant directament sobre el model del sistema, en principi, la complexitat computacional hauria de ser menor. L'enfocament estàtic d'aquest treball es fa sobre models de sistemes expressats en *notació imperativa* amb *paral·lelisme* explícit, sentències de *comunicacions síncrones* i *variables d'emmagatzematge* locals.

Els *raonaments d'equivalència* són molt empleats per números, matrius i altres camps. Tanmateix, per programes imperatius amb paral·lelisme, comunicacions i variables, encara que potencialment sigui un mètode de verificació molt intuïtiu, no han estat massa explorats. La *seqüencialització formal* via l'*eliminació de comunicacions* internes, l'àrea d'aquesta tesi, és una demostració basada en el raonament estàtic d'equivalències que, donat que disminueix la magnitud del vector d'estats, pot complementar altres mètodes de demostració. Es basa en l'aplicació d'un *conjunt de lleis* , apropiades per tal propòsit, com reduccions de reescriptura del model del sistema. Aquestes depenen de la noció d'equivalència i de les suposicions de justícia.

Aquesta tesi contribueix a la quasi inexplorada àrea de l'eliminació de comunicacions formal i seqüencialització de models de sistema. Les lleis estan definides sobre una equivalència feble: *equivalència d'interfície*. L'eliminació de comunicacions està limitada a *models sense seleccions*, per exemple models en els quals les comunicacions internes no estan dins de l'àmbit de sentències de selecció. Aplicacions interessants existeixen dins d'aquest marc. Les lleis són vàlides només per *justícia feble* o *sense justícia*. Aquesta ha estat desenvolupada seguint la semàntica proposada per *Manna i Pnueli* per a sistemes reactius [MP91, MP95]. S'han formulat les *condicions d'aplicabilitat* per les lleis de la pròpia eliminació de comunicacions. A més a més, es proposa un *procediment de construcció de demostracions* per l'eliminació de comunicacions, el qual intenta aplicar automàticament les lleis de la eliminació. També s'ha dissenyat un conjunt de *procediments de transformació*, els quals garanteixen que la transformació equivalent sempre correspon a l'aplicació d'una seqüència de lleis. Degut a que la construcció de les demostracions és impracticable, normalment impossible, sense l'ajuda d'una eina, s'ha desenvolupat un *demostrador interactiu* per la construcció semiautomàtica de la seqüencialització de models de sistemes i demostracions d'eliminació. Tant els procediments de transformació com els de l'eliminació de comunicacions estan integrats en l'eina. Amb l'ajuda del demostrador s'ha construït la demostració de seqüencialització d'un model, no trivial, de *processador pipeline*. Per aquest exemple s'ha assolit una reducció, respecte del model original, de la cota superior del nombre d'estats de 2^{-672}.

Malgrat l'enorme quantitat d'esforç dedicat a l'àrea, abans i durant la tesi, encara queda molt treball per a que l'eliminació de comunicacions i la seqüencialització sigui realment un mètode pràctic. No obstant els resultats d'aquesta tesi han establert els fonaments i han donat l'estímul necessari per continuar l'esforç.

RESUMEN

Los métodos de *verificación formal* se están usando cada vez más en la industria para establecer la corrección y encontrar los errores en *modelos de sistemas*; por ejemplo, la descripción de hardware, protocolos, programas distribuidos, etc. En particular, los verificadores de modelos lo hacen automáticamente para sistemas de estados finitos, pero están limitados debido al problema de la explosión de estados; y la *verificación formal interactiva*, el área de esta tesis, es necesaria.

El enfoque de la verificación automática trabaja sobre el sistema de transiciones del modelo, el cual define su semántica. Este sistema de transiciones tiene a menudo muchos estados, y siempre un tamaño grande comparado con el tamaño del modelo del sistema, el cual es siempre infinito. Estas consideraciones sugieren un enfoque de *verificación estática* como los de esta tesis, evitando los sistemas de transiciones, trabajando directamente sobre el modelo del sistema, en principio, la complejidad computacional tendría que ser menor. El enfoque estático de este trabajo se lleva a cabo sobre modelos de sistemas expresados en *notación imperativa* con *paralelismo* explícito, sentencias de *comunicaciones síncronas* y *variables de almacenamiento* locales.

Los *razonamientos de equivalencia* son muy empleados para números, matrices y otros campos. Sin embargo, para programas imperativos con paralelismo, comunicaciones y variables, aún teniendo la potencialidad de ser un método de verificación muy intuitivo, no han sido muy explorados. La *secuencialización formal* vía la *eliminación de comunicaciones* internas, el área de esta tesis, es una demostración basada en el razonamiento estático de equivalencias que, ya que disminuye la magnitud del vector de estados, puede complementar otros métodos de demostración. Se basa en la aplicación de un *conjunto de leyes*, apropiadas para tal propósito, como reducciones de reescritura del modelo del sistema. Éstas dependen de la noción de equivalencia y de las suposiciones de justicia.

Esta tesis contribuye a la casi inexplorada área de la eliminación de comunicaciones formal y secuencialización de modelos de sistema. Las leyes están definidas sobre una equivalencia débil: *equivalencia de interfaz*. La eliminación de comunicaciones está limitada a *modelos sin selecciones*, por ejemplo modelos en los cuales las comunicaciones internas no están dentro del ámbito de sentencias de selección. Aplicaciones interesantes existen dentro de este marco. Las leyes son válidas sólo para *justicia débil* o *sin justicia*. Ésta ha sido desarrollada siguiendo la semántica propuesta por *Manna y Pnueli* para sistemas reactivos [MP91, MP95]. Se han formulado las *condiciones de aplicabilidad* para las leyes de la propia eliminación de comunicaciones. Además, se propone un *procedimiento de construcción de demostraciones* para la eliminación de comunicaciones, el cual intenta aplicar automáticamente las leyes de la eliminación. También se ha diseñado un conjunto de *procedimientos de transformación*, los cuales garantizan que la transformación equivalente siempre corresponde a la aplicación de una secuencia de leyes. Debido a que la construcción de las demostraciones es impracticable, normalmente imposible, sin la ayuda de una herramienta, se ha desarrollado un *demostrador interactivo* para la construcción semiautomática de la secuencialización de modelos de sistemas y demostraciones de eliminación. Tanto los procedimientos de transformación como los de la eliminación de comunicaciones están integrados en la herramienta. Con la ayuda del demostrador se ha construido la demostración de secuencialización de un modelo, no trivial, de *procesador pipeline*. Para este ejemplo se ha logrado una reducción, respecto del modelo original, de la cota superior del número de estados de 2^{-672}.

A pesar de la enorme cantidad de esfuerzo dedicado al área, antes y durante esta tesis, todavía queda mucho trabajo para que la eliminación de comunicaciones y la secuencialización sea realmente un método práctico. Sin embargo los resultados de esta tesis han establecido los cimientos y han dado el estímulo necesario para continuar el esfuerzo.

ABSTRACT

Formal verification methods are increasingly being used in industry to establish the correctness of, and to find the flaws in, *system models*; for instance, descriptions of hardware, protocols, distributed programs, etc. In particular, model checking does that automatically for finite-state systems, but it is limited in scope due to the state explosion problem; and *interactive formal verification*, the broad area of this thesis, is needed.

Automatic verification approaches work on the transition system of the model, which defines its semantics. This transition system has often infinitely many states, and always a large size compared to the size of the system model, which is always finite. These considerations suggest that *static verification* approaches such as those of this thesis, avoiding the transition system, working directly on the system model would have less computational complexity, in principle. The static approach of this work is carried out on system models expressed in *imperative notations* with explicit *parallelism* and *synchronous communication* statements, and with local *storage variables*.

Equivalence reasoning is heavily used for numbers, matrices, and other fields. However, for imperative programs with parallelism, communications, and variables, although having the potentiality of being a very intuitive verification method, it has not been much explored. *Formal sequentialization* via internal *communication elimination*, the area of this thesis, is a static equivalence reasoning proof that, since it decreases the size of the state vector, could complement other proof methods. It is based on the application of *a set of laws*, suitable for that purpose, as rewriting reductions to a system model. These proofs need both proper communication elimination laws and auxiliary basic laws. These depend on the notion of equivalence and on the fairness assumptions.

This thesis contributes to the almost unexplored area of formal communication elimination and system model sequentialization. The laws are defined over a weak equivalence: *interface equivalence*. Communication elimination is confined to *selection-free models*, i.e. models none of whose inner communications are within the scope of selection statements. Interesting applications already exist within this framework. The laws are valid only with *weak fairness* or *no fairness*. It has been developed following the same semantics as *Manna and Pnueli* for reactive systems [MP91, MP95]. *Applicability conditions* for the proper communication elimination laws are derived. In addition, a communication elimination *proof construction procedure*, which attempts to apply the elimination laws automatically is proposed. A set of *transformation procedures*, guaranteeing that the equivalence transformation always corresponds to the application of a sequence of laws have been designed as well. Since the construction of elimination proofs is impractical, even impossible, without a tool, an *interactive prover* for semi-automatic construction of system model sequentialization and elimination proofs has been developed. Both transformation and communication elimination procedures are integrated within the tool. As a non-trivial example, a sequentialization proof of a *pipelined processor* model, has been constructed with the help of the prover. A reduction, with respect to the original model, of 2^{-672} on the upper bound on the number of states has been achieved in this example.

In spite of the huge amount of effort already devoted to the area, before and during this thesis, much work still needs to be done until communication elimination and sequentialization become a practical method. Nevertheless the results of this thesis have established its foundations and given the necessary encouragement for continuing the effort.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deep gratitude to my advisor, Prof. Miquel Bertran, for his enthusiasm, constant support and invaluable encouragement throughout all the thesis. I am indebted to him for the opportunity of working in an original and interesting research area.

I would also like to express my sincere gratitude to former director of *La Salle*, *Universitat Ramon Llull*, Prof. Daniel Cabedo, who passed away on June 2006, and to Prof. Josep Martí, then chief of studies, for their acceptance, patience, and economical support during the first half of my Ph.D. work.

I also thank the people of the *grup de recerca en sistemes distribuïts*, together with its head August Climent for the support of the work; also the *departament d'informàtica* of *La Salle*, Universitat Ramon Llull in general. Specially I am very grateful to Anna Villoslada and Gabriel Salvà for helping me in different aspects during this work.

I would like to extend the acknowledgment to the group of *General Systems Development* company: Albert Duran, Miquel Porta, Joan-Andreu Margalef and Román Duch. They always supported and helped me in the usage and maintenance of PADD/RALE, the distributed program development environment used in my thesis; and in many other ways during this research work.

I am really glad that Prof. Zohar Manna accepted my visit to his group at *Stanford University*, and for his invaluable comments along the thesis. Also many thanks to Tomás Uribe, Nikolaj Bjørner, Bernd Finkbeiner, and Henny Sipma, for introducing me to the temporal verification methodologies. Special thanks to Bernd Finkbeiner who followed and encouraged this work from its beginning. Also the criticism of Prof. Ricardo Peña at a certain stage of the work is appreciated.

The initial inspiration of the research leading into this thesis, occurred during the sabbatical stay of my thesis advisor at the Rutherford Appleton Laboratory, British Science and Engineering Research Council, invited by Dr. Robert Witty, also inventor of Dimensional Flowcharting, the tree-like notation from which PADD was born. Together with my advisor, I would like to acknowledge this and express my gratitude to Dr. Robert Witty.

I am also grateful to the coauthors of my communications and articles: Miquel Bertran, August Climent, Jordi Riera, and Miquel Nicolau, for their help while writing the papers, and to all the anonymous referees.

This research work would have not been possible without funding. In this regard, I would like to thank *General Systems Development* company, *La Salle*, and the partial support from the spanish *Ministerio de Educación y Ciencia* under the CICYT project TIN2006-14738-C02-02.

Last but not least, the final acknowledgement is addressed to my family. I would like to thank my parents Joan and Maria Dolors, and my brother Jordi for their unconditional support and patience. To them and to my friends I dedicate this thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF PROCEDURES

Chapter 1

INTRODUCTION

The work reported in this thesis is about formal static communication elimination from imperative programs and parallel program sequentialization, all via equivalence reasoning. These are relatively unexplored topics. Thus, this chapter defines and delimits the corresponding areas, and provides some motivation. It also introduces the broad fields to which these topics pertain. The base theory which is needed for the development of the proper thesis work is identified as well, but its review is left for the second and third chapters.

1.1 Motivation

Equivalence reasoning is extensively used for numbers and matrices. Research on equivalence reasoning in the area of concurrency and distribution has been carried out in the very specific field of process algebras and action systems, such as CCS [Mil80, Mil89], CSP [Hoa85] and ACP [BK84, BK85], where the equivalence is defined on state transition systems and computation trees. These models are very abstract, they are far from the intuitiveness of an imperative programming notation with variables and boolean conditions. Effort on equivalence reasoning at the more intuitive level of imperative program text is still needed for distributed, concurrent and reactive system models.

Imperative concurrent, reactive and distributed programs, with explicit parallelism and synchronous, handshaking, communications, are the framework of this thesis. With them both software and hardware system models can be constructed. This work contributes to the study of equivalence reasoning for these programs via communication elimination and sequentialization.

Equivalence reasoning for imperative concurrent and distributed programs, although having the potentiality of being a very intuitive verification activity, remains substantially unexplored within the broad field of interactive verification. Therefore, conceptually, it deserves further attention.

An impediment for the widespread use of verification in engineering is the needed expertise of the engineer in mathematical logic. Equivalence proofs remain at the imperative notation level and its transformation laws; thus removing obstacles for engineers. There is no need to go into invariants and predicate calculus. Mathematical reasoning is only needed to justify soundness of the laws and the equivalences. This has to be done only once, but not by the verifying engineer. In spite of not being a general verification approach, this intuitiveness and simplicity gives and additional motivation for studying the method and its application scope.

The widely researched area of *model checking* is affected by the state explosion problem, since the checkers operate on the semantic model: the finite transition system of the model program. Model checking algorithms have exponential complexity in the size of the program. The algorithms studied in this thesis for the automation of communication elimination and sequentialization proofs have polynomial complexity in the size of the program, as will be apparent. Although these proofs are not a general verification method, their complexity reduction makes their study appealing.

1.2 Imperative Notations with Synchronous Communications

Verification at the level of the source program is addressed in this work. Notation has been recognised as a very important element in general, and for verification in particular. Some of the pioneers of verification designed structured imperative notations like Pascal so that verification was more natural, or even possible. As a side comment, think about developing arithmetic in roman numerals. The notations used in this work are imperative with parallelism and synchronous communications and fall under the influence of Pascal.

Imperative notations with explicit parallelism and communication statements provide an intuitive, explicit, and complete framework to express distributed programs and system models with perspective and clarity. OCCAM [IL85, IL88, Jon87], ADA [Dep83, TDB+06], the *simple programming language* SPL of Manna and Pnueli [MP91, MP95], PROMELA of the SPIN model checker [Hol91], and the *shared-variable language*$^{++}$, SVL^{++}, in [dRdBH+01] are representatives of them.

To add motivation to study verification methods for imperative notations with parallel processes and synchronous communications, as this work does, it is interes-

ting to realize that recent approaches to improve operating systems and general concurrent program correctness and dependability, advocate for the use of concurrency in the form of parallel processes with local storage, communicating via rendez-vous; the notational framework of this work. [Lee06] is an example, proposing *coordination languages* for expressing this type of communication among sequential processes composed in parallel, as a way to avoid the more problematic shared storage concurrency. Also, in order to attain the above aims, new operating systems designed as microkernels, with operating system processes around them have been proposed in [THB06]. In an example of this approach, Minix3 [HBG+06], the microkernel offers rendez-vous as the only means of interprocess communication.

An unavoidable problem in system development is mapping: the transformation of the verified model into distributed software, or hardware, pertaining to the real world. Mapping, into distributed software systems, of imperative programs with parallelism and simple rendez-vous has been extensively treated in the literature for an even more general notation: multiparty interaction, the notation IP (*Interactive Processes*). The references [FF96, CRTC99, PCT04] are quite illustrative.

1.3 Formal Verification Methods

Formal methods are increasingly being used in industry to establish the correctness of system models and to find the flaws in them; for instance, descriptions of hardware and protocols. In particular, model checking [CE81, QS82, Hol91, MD93, CGP99, MOSS99, Mer01, BK08] does that automatically for finite-state systems. However, model checking is limited in scope due to the state explosion problem. Most practical system descriptions, notably those of software, are therefore not directly amenable to finite-state verification methods since they have very large or infinite state spaces. For such systems, interactive theorem proving, for instance with the PVS prover [OSRSC01a, SORSC01, OSRSC01b], or with STeP [BBC+95, BBMC+00], or with ACL2 [KM09], has so far been the only viable alternative; in spite of its use requiring manual effort and mathematical sophistication. Model checking often requires much interaction in practice, for example for arriving at an initial model suitable for checking. However, research on model checking for models with dynamic data structures is underway [dMGMJS07, dMGMS08]. New paradigms and methods that combine the ease of use of model checking with the power and flexibility of theorem proving are needed. Such hybrid techniques started to emerge [MBSU98, SUM99]. Anyway, interactive verification is unavoidable and needs further research; the effort reported in this dissertation contributes to it.

The formal methodology presented in this work is another interactive verification approach, named *Distributed Program Sequentialization* (DPS). The goal is to obtain a simpler model equivalent in some sense to the original distributed system. It is based on equivalence proofs using a suitable set of laws applied as reductions to

a program. An interactive prover tool is mandatory to carry out and store these proofs.

1.4 Static Verification versus State Explosion

Though there are many different approaches to automatic verification of programs, they are all limited by the space which is available on a given machine. Even small programs may have a significantly large state-space, so that verifying programs which implement solutions to realistic problems is difficult.

Automatic verification approaches work on the transition system of the program, defining the semantics of the distributed system. This transition system has often infinitely many states, and always a large size compared to the size of the program modeling the distributed system, which is always finite. The situation in which small programs correspond to exponentially large models is known as the state-explosion problems. In order to grasp the magnitude of this problem, suppose, for instance, that the number of variables of a program is n, each holding an integer of m bits. The number of states may then be of the order of $2^{n \times m}$, whereas the size of the program remains of an order close to n. The size of the control-flow graph is proportional to the number of statements in the program. The size of the transition system is much larger since it is also proportional to the sizes of the variable domains. It is the size of the transition systems which creates the difficulty in automatic verification.

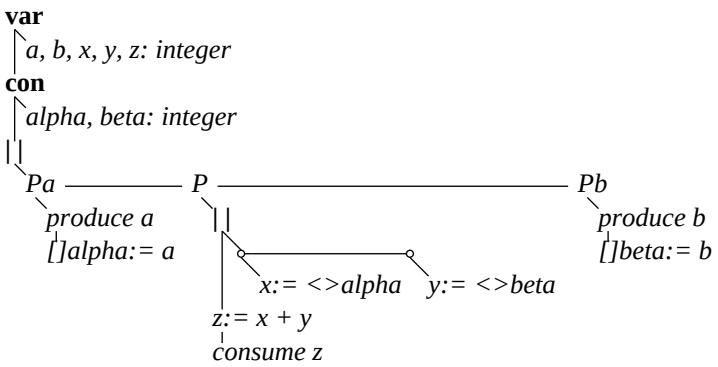
Static analysis is the process of examining the control-flow graph of the program (the system) to extract information on its semantics, without creating the semantic model. The syntactic model is significantly smaller than the transition system since it expands only the program counter and not the program variables, which are the source for the enormous size of the semantic model. These considerations suggest that static automatic verification approaches, avoiding the transition system, working directly on the program would have less computational complexity, in principle. Following this line of work, static analysis methods for state reduction [CGL94, KLM$^+$98, YG04] have been proposed as a step prior to model checking. They reduce the size of the transition system and hence the complexity of model checking. The verification method to be presented here falls within the broad category of static verification, thus benefiting from its computational complexity advantages. The size of the transition system would be reduced, indirectly, by working on the source program, as the work of this thesis does.

1.5 Communication Elimination and Equivalence Reasoning

Sequentialization via internal communication elimination [BBCN01] is a static equivalence reasoning proof. It is based on the application of a set of laws, suitable for that purpose, as reductions to a program. There are *proper* communication elimination laws, whose application removes communication statements, and auxiliary, or basic, laws. After communications have been removed, parallel processes are disjoint and can be reorganized with other laws, and redundant variables can also be eliminated. When all this has been possible, a sequential program is obtained.

$$\left[\begin{array}{ll} \textbf{local} \quad a,b,x,y,z & : \textbf{integer} \\ \textbf{local} \quad \alpha,\beta & : \textbf{channel of integer} \\ & \\ \quad P_a :: \left[\textbf{produce } a;\ \ \alpha \Leftarrow a;\right] & \\ \quad || & \\ \quad P :: \left[\ \left[\alpha \Rightarrow x\ ||\beta \Rightarrow y\right];\ \ \left[\begin{array}{l} z := x+y; \\ \textbf{consume } z; \end{array}\right]\right] & \\ \quad || & \\ \quad P_b :: \left[\textbf{produce } b;\ \ \beta \Leftarrow b;\right] & \end{array}\right]$$

As an illustration of communication elimination and sequentialization, consider the above SPL program, having top-level parallel or cooperating disjoint processes P_a, P and P_b. The inner process P is a concatenation whose first substatement is a parallel composition of two synchronous communication, receive, substatements over channels α and β. They match with synchronous communication, send, substatements, over the same respective channels, in processes P_a and P_b respectively. The same program expressed in PADD, another notation used in this work [BDP+09], would be:



The elimination of synchronous communications, via α and β, followed by sequentialization, would give the equivalent program:

$$\left[\begin{array}{l}\textbf{local} \quad a, b, z \quad : \textbf{integer} \\ \textbf{produce } a; \ \textbf{produce } b; \\ z := a + b; \\ \textbf{consume } z\end{array}\right]$$

This form reflects the essential function of the original program, and can be verified, much more easily, with sequential program verification methods. The work of this dissertation concentrates on the unexplored area of communication elimination and equivalence reasoning as illustrated by the above example. Other static transformation systems were proposed in [dFS98, SO99, Sch99], exploring communication elimination in frameworks different from imperative distributed programs with parallelism and synchronous rendez-vous communication.

Equivalence reasoning with communication elimination has applications in formal design. Actually, using it in the opposite direction, as communication introduction. This is commented in subsection 1.10.

1.6 Limitations of this Work

This work applies to a certain class of models only. But, in addition, the assumptions on the underlying scheduler are restricted to weak fairness or no fairness at all. Strong fairness is prohibited, since some of the laws are not valid under this assumption. Parallel statements in the model should be *disjoint*: no written variables being shared. Data is communicated among parallel processes via *synchronous message passing* only.

The results of this thesis are restricted to programs none of whose communication statements are under the scope of selections. Only sequential and parallelism operators are accepted, like the example above. Nevertheless interesting examples, like the pipeline processor in chapter 6, fall under this class. This limitation stems, on purpose, from a research plan. Obviously elimination of communications outside selection scopes is more tractable than general elimination. Nevertheless it is complex enough to be approached in isolation. The plan was to tackle the selection free case, taking its solution as a base, and then enter into the elimination of communications under selections. In fact, we are, as a joint effort outside the thesis scope, in an advanced stage with respect to the latter case.

The difficulties of elimination for general programs, where communications may be under selections, are illustrated in subsection 7.2.4.2. The complexity of analyzing the closely related problem of synchronization of concurrent programs has been treated in [Tay83, Ram00].

1.7 The Need for Laws in a Suitable Equivalence

The transformation overviewed above needs laws such as $P;\mathbf{nil} \approx P$, $P||\mathbf{nil} \approx P$, and associativity of parallelism, which are auxiliary since they do not eliminate any communication statement. It also requires a proper communication elimination law such as:

$$\left[\begin{array}{c} H^l; \\ \alpha \Leftarrow e\ ; \\ T^l \end{array}\right] \ || \ \left[\begin{array}{c} H^r; \\ \left[\ \alpha \Rightarrow u\ \ || \ P^r\ \right]; \\ T^r \end{array}\right] \ \equiv \ \left[\begin{array}{c} \left[\ H^l \ \ || \ \ H^r\ \right]; \\ \left[\ u := e\ \ || \ \ P^r\ \right]; \\ \left[\ T^l \ \ || \ \ T^r\ \right] \end{array}\right]$$

which is an instance of the communication elimination schema given in section 3.2.2. H^l and H^r statements do not communicate over channel α. The equivalence symbol is general and will be given concretion in section 2.8.

This law, applied from left to right, eliminates the communication statements over α, replacing their function with the assignment $u := e$. The auxiliary laws are necessary in order to transform the program into a form ready for structure matching with an elimination law.

The soundness of the laws depends on the notion of equivalence and on the fairness assumptions [MP91]. A set of laws for OCCAM was given in [RH88]. Although a simple communication elimination law was included, rather than communication elimination, the focus there was to obtain normal forms and to define the semantics of the notation. Communication closed layered systems are a special class of distributed systems, introduced in [EF82]. Some laws for them are given in [dRdBH$^+$01], in the framework of SVL^{++}. For instance, sequential-parallelism transformation and iteration unfolding transformation laws. The aim there was formal design by transformation, but no communication elimination law was reported. Communication elimination proofs in this thesis use the latter as auxiliary laws. Some laws for SPL are given in [MP91], with a very clear SPL semantics, based on fair transition systems (FTS), but none is for communication elimination. Thus, the work to be presented may be regarded as a concrete continuation line of all these works.

As a needed grounding work for this dissertation, a set of relations for communication elimination was given and proved sound in [BBCN01]. Its cardinality had to be unbounded and strong fairness had to be prohibited. Only weak fairness or no fairness at all are compatible with the soundness of these laws. In this earlier work, the notion of equivalence was assimilated to the congruence of [MP91], a very strong equivalence. This had the drawback of limiting the formulation of most communication elimination laws to unidirectional refinements. The need of working with a weaker equivalence, for all laws and avoiding the asymmetry of refinement relations, was outstanding.

1.8 Interface Equivalence and Substitution Rules

The notion of *interface equivalence* [BBC05b] is weaker than congruence, but strong enough to preserve the input/output relation of distributed programs and to lead to laws for communication elimination. It was also developed as a needed grounding work for this dissertation. There are many other equivalences in the literature, within process algebras [vG01], in the polychrony framework [GTL02], etc. It would be interesting to study interface equivalence in their perspective. A very interesting order among many equivalences has been reported in [dFEGR05, dFEVGR07, dFEGR09]. There, the weakest is *trace equivalence*. Interface equivalence is at the level of trace equivalence. In it all computations with the same intermediate results, but with different relative orders, are equivalent. Interface equivalence is justified in a semantics where each statement denotes a set of *interface behaviors*. These extend the notion of *reduced behavior* given in [MP91]. In order to capture the complete input/output relation, the former adds auxiliary variables, in addition to the usual data state variables, to record the values traversing synchronous channels. This makes explicit the fact that values may be input or output via channels, as well as via proper variables. All these notions were introduced since they were the minimal extensions needed while keeping this work within the framework Manna and Pnueli books. In this semantic context, the grounding work with streams introduced in [Bro97, Bro99, Bro01] is important, but in the concrete imperative program context of this work a new model, where both channel and variable values were explicitly taken into account and distinguished, was needed. The work on compatibility of components [dA03] has some relation also.

Important components of equivalence reasoning are a procedure reference unhiding rule and a rule for the substitution between procedure reference statements, to equivalent procedures. These allow proof decomposition. Conditions for the validity of such substitutions are also given and their justification outlined in [BBC05a] and in this work. Altogether, equivalence and substitution rules establish the necessary base theory for formal interface equivalence reasoning about distributed programs.

1.9 Elimination Procedures and Program Sequentialization Proofs

For the elimination of communications under any nested structure of parallelisms, an infinite set of communication elimination laws are required, [BBCN01]. Nevertheless, communication elimination procedures exist for the automatic application of the laws as reductions. Preliminary work towards the justification of a communication elimination procedure was undertaken in [BBCN01, BBC05b]. Proper communication elimination laws were mathematically justified in the new weaker equivalence in [BBC05a], and their complete set of applicability conditions was derived as part of this thesis. In addition, a communication elimination reduction procedure was proposed and justified, as part of this thesis also.

The inner communication-free program which results from the communication elimination procedure can be transformed interactively, with another set of laws, into a sequential program with less variables than the initial distributed program, but equivalent to it. The whole proof constitutes a *distributed program sequentialization* (DPS) proof. Since it decreases the size of the state vector, it could complement and be combined with other proof methods, such as model checking or interactive verification [BBMC$^+$00, KM97] as a succeeding step, reducing overall proof complexity. In many cases, only an equivalent purely sequential program has to be verified. Such is the case for the pipeline processor example presented in chapter 6. Nowadays combining different formal techniques is a promising direction for verifying complex hardware and software system models. Nevertheless, this line is not explored in this dissertation.

A DPS proof of a distributed fast Fourier transform was outlined in [BBCN01]. The result of another DPS proof of a pipelined processor model, carried out with the help of a tool which implements a communication elimination procedure, was reported in [BBC05b], without going into its steps. A very detailed report of the steps of the proof is given in the present work, which also analyzes the substantial state-space reduction obtained with it.

1.10 Formal Parallelization

The reported work is very relevant in the popular area of parallelization in formal design. Communication elimination and sequentialization proofs are reversible, since they are equivalence proofs. Consequently any sequentialization proof can be reinterpreted as a formal parallelization process. The laws used in this work would be applied then in the reverse direction; as a parallelism and communication introduction step.

References [SSB04, SJ00] provide examples in hardware / software partitioning. A sequential specification is transformed, introducing parallelism and communications into a model of a hardware implementation. Formal design is not treated in this thesis. It is an area of future work, section 7.2 of chapter 7.

At the beginning of our work [BBCN01] both communication elimination and introduction were envisaged. Although the work evolved with a bias to elimination and sequentialization, it could have evolved in the opposite direction. The adaptation of the procedures and tools of this dissertation to formal parallelization is proposed as future work.

1.11 Contributions and Plan of this Thesis

For self containment reasons, and as an introduction to the presentation of the main results of this thesis, chapter 2 treats notation and background notions which, although needed for the presentation, do not form part of this thesis, whose main contributions are the following:

- Formulation of *applicability conditions* for the communication elimination law schema, for selection-free programs. This is discussed in chapter 3 and was presented in [BBC05a].

- A communication elimination *proof construction procedure*, which attempts to apply the elimination laws automatically. This is treated in chapter 4. The elimination procedure is integrated within a prover tool.

- A set of *transformation procedures*, that guarantee that the equivalence transformation always corresponds to the application of a sequence of laws. They have been implemented to mechanize some tedious parts of DPS proofs. They are detailed in chapter 5, and are integrated within a prover tool.

- The design and development of an *interactive prover tool* for the semi-automatic construction of distributed program sequentialization proofs. The construction of these proofs is impractical, even impossible, without such a tool. It is described in chapter 5.

- A sequentialization *proof of a pipelined processor model*, carried out with the help of the tool, with the communication elimination procedure embedded in it. This proof example is detailed in chapter 6.

Throughout this thesis, communication elimination is confined to selection-free programs, i.e. programs none of whose inner communications are in the scope of selection statements. Interesting applications can already be attempted within this

framework. The thesis work has created the essential base for the needed extension, which is underway as continuation work.

Chapter 2

MODELING NOTATIONS AND GROUNDING NOTIONS

This chapter is on notation: syntax, semantics, and related grounding notions on which the proper dissertation work depends. The mathematical apparatus needed to justify communication elimination and DPS equivalence proofs in the notation, the main concern of this dissertation, integrates various notions, which should be defined with precision. Although this mathematical apparatus has been covered elsewhere, this chapter gives an overview, without going into proofs.

After introducing two syntactic forms of modeling notation, a common semantics is discussed: the fair transition systems of Manna and Pnueli with the extended notion of interface behavior. An equivalence based on interface behaviors, some basic laws for this equivalence, procedure unhiding/encapsulation rules, and procedure reference substitution rules constitute a minimal set of notions required to construct equivalence proofs for distributed system models. This chapter presents all these topics, which will prepare the reader to go into proper communication elimination laws, a procedure for their automatic application, the interactive prover tool embedding it, and complex DPS proofs. All these are left for other chapters.

2.1 A Tree-like Notation

2.1.1 Introduction

This section introduces informally the modeling notation, PADD [BDP+09], of the prover tool. It adopts an explicit tree form: *dimensional flowcharting*, proposed in [Wit77] and further elaborated into *dimensional design* [Wit81]. An algebraic formalization of dimensional design is given in [Ber88].

Both, models to be equivalence proved and the tool itself are expressed in PADD.

As a side comment, a modeling and simulation based distributed program development environment, RALE, exists for this notation and quite a number of interesting systems have been developed in it at the *General Systems Development* company [GSD]. The following are some of them: a simulator for an open air active noise cancellation system where both the acoustic field and the digital signal processing algorithms are modeled, a simulator of a packet switching communications network for a power utility providing telecontrol, IP-routing, file transfer, and other sorts of services, and many other systems. To give an idea of order of magnitude, the telecommunication network model has about 3000 processes and 5000 communication connections (to be explained later in this chapter).

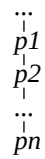
It should be stressed that, for this dissertation work, PADD is just another syntactic form for the variant of SPL which has been used. Its formal semantics is the same as the one described for such a variant later in this chapter.

2.1.2 Basic Statements

The basic statements are **skip**, **nil**, **stop**, the assignment $u := e$, and the communication statements, send and receive, to be introduced later. Storage variables are declared within a **var** construct.

2.1.3 Sequence

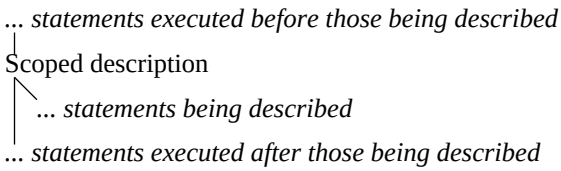
Sequential composition of statements, also referred to as *concatenation*, is expressed by connecting them with vertical edges.



The execution order starts with the upper statement, and continues down the list. Execution ends when the last statement in the list ends.

2.1.4 Scoped Descriptions

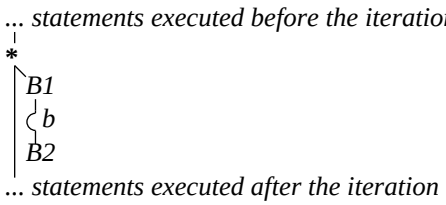
Comments are scoped descriptions. They consist of a line of text describing its lower diagonal tree, its scope.



Execution of a diagonally refined comment means execution of its diagonal subtree. When it ends, execution continues at the statements connected vertically under the description line. Within the notation, the diagonal edge denotes *refinement*. Scoped descriptions may be used recursively.

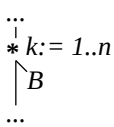
2.1.5 Sequential Iteration

A sequential conditional iteration consists of the symbol '*', denoting the fact that an iteration occurs at this point of a sequence, and the iteration body, standing at its diagonal lower subschema (its refinement).



The boolean expression b can stand at any point of the body. The semicircle at its left makes it more explicit. $B1$ and $B2$ are the parts of the body that execute before and after the boolean condition. Both are optional but at least one of them should be present.

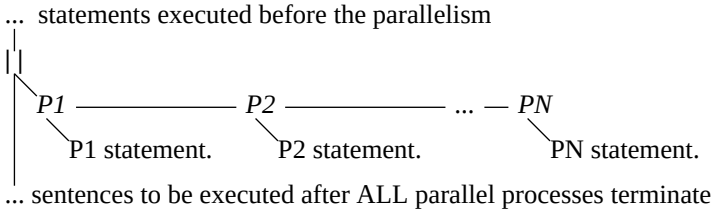
Repetition takes place while the boolean evaluates to true. Otherwise exit occurs, and execution continues with the process connected vertically under the iteration symbol '*'.



The usual indexed iteration, as shown above, is available also. B is the body of the iteration, whose execution is repeated.

2.1.6 Parallelism

Parallelism is always explicit. The following is the typical parallelism schema connecting N statements in parallel

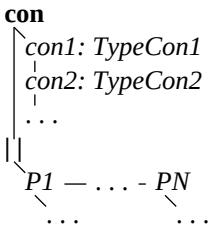


The execution of each of the statements corresponds to a process, which is a term expressing the dynamical aspect of a statement. Parallel composition of statements is also referred to as *cooperation*. It is defined with the parallel composition construct. The items in the composition have a heading label Pi. Labels are connected horizontally. The symbol '||' represents the whole parallelism within a sequence. The horizontal list of processes is its lower diagonal refinement, connected to it by the diagonal edge.

Execution starts by creating and starting all processes. The relative order of execution of their component subprocesses is unknown, in principle. Execution ends only when all process end, and continues with the sentence connected in sequence after the symbol '||'.

2.1.7 Connections

Connections are typed but memoryless points which enable synchronized half-duplex point to point communication between two parallel processes. They are declared before the parallelism operator, as in the following schema.



Connections will be used within the processes connected in parallel in order to send and receive values. Communication takes place when communication state-

ments are executed. This will be explained later. The **con** construct declares connections to be used within an algorithm. Their scope is the vertical subschema under the keyword **con**. They are unseen anywhere else.

A **con** construct with a parallel composition of processes communicating through the declared connections within its scope, defines the framework for a network of communicating processes.

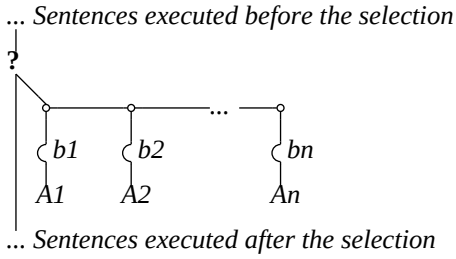
A connection should not be confused with an asynchronous channel, which consists of a queue of messages and which is modeled as a process in PADD. Another word for connection is *synchronous channel*.

2.1.8 Internal and External Connections

Throughout this work, connections are classified into *internal* and *external*. This classification is associated with procedures, and will be explained later when modular procedures are treated. Usually, the set of internal connections will be clear from the context, and will be denoted by I.

2.1.9 Selection

The selection construct connects its alternatives with horizontal lines, through the alternative headings (an structural dot, the 'o' character, in the current implementation). Each alternative A_i is headed by its boolean condition bi, preceded by a left parenthesis.



The alternatives are connected horizontally under the diagonal scope of the selection symbol '?'. The leftmost alternative whose condition evaluates to true is selected for execution. At any possible execution, at least one boolean condition should evaluate to true, otherwise this causes a run-time error. When **else** stands in place of the boolean condition of the rightmost alternative, the last condition is met always. When the else alternative does nothing a **nil** should be placed there. Execution of the selection ends when the selected alternative ends. It continues with

the next process connected vertically under the symbol '?'. A different semantics, *non-deterministic*, is followed throughout this work, where one alternative is selected non-deterministically among those whose guard evaluates to **true**.

2.1.10 Abstract Communication Pairs

The actual event of communicating a value via a synchronous channel is specified with communication operations within the algorithm of processes composed in parallel. There exist the send and receive operations associated to each type. They will be referred to as Abstract Communication Pair (ACP) in the sequel.

An ACP is associated to a type and its two complementary operations, send and receive, serve to communicate values of the type. The specific form in which the communication takes place is unknown to the programmer, this is the motivation for the word *abstract*.

Given $c : t$ and $v : t$, a connection and a variable of type t, the following syntax will be used for the receive and send operations respectively:

$$v :=\!\!<\!\!> c \quad and \quad []c := v$$

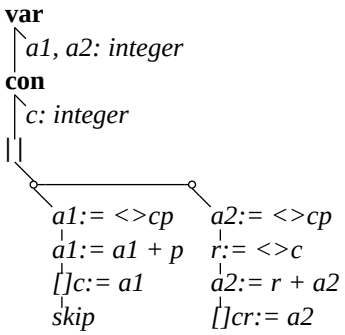
For *nil* typed connections, the syntax

$$<\!\!> c \quad and \quad []c$$

will be used. In this case, no value is passed, just two synchronizing signals.

Communication takes place without intermediate buffering, both processes synchronize at their communication operations. When both are ready to communicate, the value of type t is stored in the variable at the receive operation. Process execution continues in parallel. The first process that tries to communicate will wait for the other. While waiting, a processor will be free and the scheduler will assign it to some other process which is ready for execution.

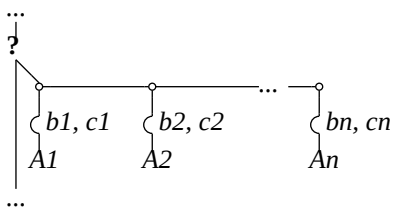
A simple example will illustrate the constructs presented so far:



Connections *cp* and *cr* communicate with other parallel processes; they are *external*. Variables *p* and *r* serve also to communicate values with others processes, but via storage. Connection *c* is *internal* since its scope is limited to the binary parallel statement. It enables communication between the two sub-process.

2.1.11 Communications Selection

Communications selection has alternatives connected in an horizontal list. The execution of only one alternative is determined by boolean conditions *bi* and communication operations (guards) *ci* located at alternative headings. The alternative to be executed is selected depending on the evaluation of conditions and the possibilities of communication with the implied neighbor processes. The logic disjunction of all the conditions should evaluate to true in all possible executions of the selection. Otherwise erroneous termination occurs.



A non-deterministic semantics will be adopted. The alternatives whose boolean guards *bi* evaluate to true are said to be *open*. Among the open alternatives, one alternative among those whose communication guard can communicate is chosen non-deterministically.

In the current implementation, the semantics is deterministic with left priority.

2.2 Textual Notation: Syntax and Related Notions

2.2.1 Introduction

Programs will also be expressed in a variant of SPL, which is general enough to express any practical program. Its syntax is presented now. Basically, the general selection statement of SPL has been restricted to only boolean and guarded communication selection forms. Also, in an initial attempt to obtain the above mentioned intuitive set of auxiliary equivalence relations, a *nil* statement has been introduced.

Any statement S may have explicit *pre* and *post labels*, ℓ and $\hat{\ell}$ respectively, as in $\ell; S; \hat{\ell}$. They serve statement and/or control location identification purposes. Control *locations*, to be further defined below, are classes in the label equivalence relation, since more than one label may denote the same location in a program.

2.2.2 Basic Statements

There are the same basic statements introduced above, subsections 2.1.2, 2.1.7 and 2.1.8. The two communication statements have the syntax $\alpha \Leftarrow e$ for the send, and $\alpha \Rightarrow u$ for the receive. This work is limited to synchronous channels α, which will be referred to as *channels*. In them both the sender and the receiver wait for each other before exchanging a value and continuing execution. The two communication statements will be referred to more simply as *communications*. Both channels and variables are declared globally before their usage.

2.2.3 Compound Statements

The rest of the notation is defined recursively. The *concatenation* statement is n-ary:

$$S_1; \cdots ; S_n$$

The iteration statements are **while** c **do** S , where c is a boolean expression, and the indefinite iteration

$$[\textbf{loop forever do } S] \stackrel{def}{=} [\textbf{while } true \textbf{ do } S].$$

The *cooperation* statement is also n-ary:

$$S_1 || \cdots || S_n$$

Its substatements S_j are the *top parallel statements* of the cooperation statement, which is the *least common ancestor* of them. It will be assumed throughout this work that the S_j's are disjoint, and that they communicate values through synchronous channels only.

The *regular selection* and the *communication selection* statements are non-deterministic and have, respectively, the forms

$$b_1; S_1 \textbf{ or} \cdots \textbf{or } b_n; S_n$$

and

$$b_1, c_1; S_1 \textbf{ or} \cdots \textbf{or } b_n, c_n; S_n$$

where the b_i's are boolean expressions referred to as *boolean guards*, and the c_i's are synchronous communication statements referred to as *communication guards*.

2.2.4 Related Notions

Following the references [MP91, MP95], Substatements S_i and S_j, $i \neq j$, of a cooperation are said to be *parallel*. Similarly for any pair of substatements, one in S_i and the other in S_j.

Also, if S' is a substatement of S, it is said that S is an *ancestor* of S'. Every statement is an ancestor of itself. A statement S is said to be a *common ancestor* of S_1 and S_2 if it is an ancestor of both. Also, S is said to be a *least common ancestor* (LCA) of statements S_1 and S_2 if it is a common ancestor of S_1 and S_2, and any other common ancestor $\tilde{S}$ of S_1 and S_2 is an ancestor of S. Every two statements have a unique least common ancestor.

Two statements are said to be *disjoint* when no variable written in one of them is either read or written in the other or vice versa. Two statements are also said to be *parallel* if their LCA is a cooperation statement that is different from both. This is equivalent to the definition given at the beginning of this subsection. Two communication substatements *match* if they are parallel, one of them is a send, and the other is a receive over the same channel. Two statements *communicate* if one of them has a communication substatement $com(\alpha)$ through channel α which has a matching communication substatement $c\bar{o}m(\alpha)$ in the other statement. If two substatements communicate, then they have to be parallel.

2.3 Modular Procedures in PADD

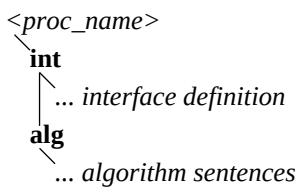
A modular procedure is a piece of a program which is given a name, and whose relation with the rest of it is explicitly defined within its interface. The reason for

the name *modular procedure* is to avoid confusion with the procedures of sequential programming languages, where parallelism and communication constructs are not allowed. The notion combines some elements of the SPL *module* [MP91, FMS98] and of *procedure*.

A modular procedure can be composed in parallel with other processes, which may have the form of references to other modular procedures. Connection (synchronous channel) names can be declared at the interface of a modular procedure. This is one of its differences with sequential procedures. In the following, the name *modular* will be dropped and modular procedures will be refered more simply as *procedures*. When used in a sequential program, modular procedures become just procedures.

Modular procedures may be invoked by *procedure reference* statements, which make explicit the names of all the interface channels and variables. Common variables are prohibited.

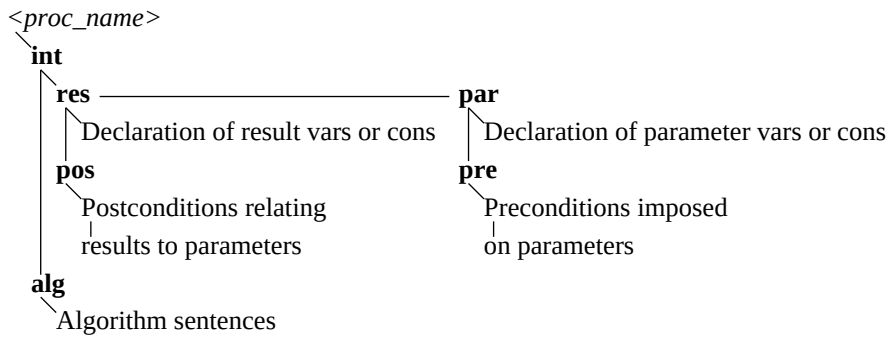
The algorithm (**alg**) or body and the interface (**int**) are the two only parts of a procedure. They stand as diagonal subtrees under the keywords **alg** and **int**, respectively.



The interface defines the relation of the procedure with the rest of the program. This is done in terms of variables and connections through which values (data) are interchanged. The algorithm defines the actions taken when the procedure is executed. Thus, the interface is static whereas the algorithm is dynamic.

2.3.1 Procedure Interface

Both communication connections to exchange values with the rest of the program, and variables that are used and/or computed by the procedure are declared in its interface, whose syntax is shown within the following schema.

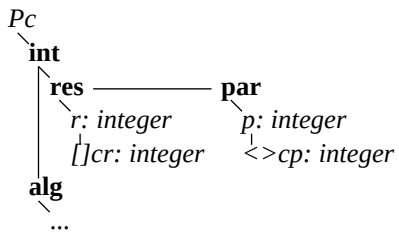


The interface of a procedure has parameter (**par**) and result (**res**) subtrees, where interface variables and connections are declared. Their scope is the whole algorithm section of the procedure. When a parameter variable has to be modified by the procedure, it should be declared as a result also. For the semantics, both parameter and result variables are passed by reference. Therefore, a change of a result variable carried out within the algorithm of a procedure is seen immediately by any parallel procedure having the same variable as a parameter.

Every variable which is updated by a procedure should be declared as a result. Forcing explicit interface declarations prevents hidden lateral effects. This has also advantages concerning self-documentation, and forces a healthy programming style, where structures and compound objects should be passed in order to decrease the number of parameters and results needed.

Another possible form of relation of the procedure with the rest of the program is the communication of values to parallel processes via connections. These communication connections form part of the interface and should be declared within it also. Output and input connections go in the result and parameter sections respectively. Input connections have names prefixed with the string "<>". Output connections have names prefixed with the string "[]". The usage of these connections takes place within the algorithm body.

As an example, the heading and interface of the example in page 19 could be the following:



2.3.2 Procedure Reference

A procedure reference starts with the list of actual results, followed by the assignment symbol ':=' and the procedure name. It ends with the list of actual parameters, within parenthesis, even when this list is empty. Their relative orders and typings should match the procedure interface declaration.

$$r1, r2, ... := \ Proc.Name(p1, p2, ...)$$

By forcing the explicit reference to all parameters and results there is no need to examine the procedure body in order to determine which objects are updated by the procedure execution, since only the actual results, stated in the procedure reference, may change. This makes easier the implementation of the procedure reference substitution rule.

2.4 Modular Procedures in the Textual Notation

2.4.1 Syntax

An example of procedure is given below. It is the same procedure of subsection 2.1.10. Its *procedure reference* stands at the left, and the *procedure body* at the right. The reference takes also the role of the procedure heading; defining the name, the parameters and the results with the semantics given above for the PADD notation.

$$(r, cr) ::= Pc(p, cp) :: \left[\begin{array}{l} \textbf{out } r : \textbf{integer} \\ \textbf{out } cr : \textbf{channel of integer} \\ \textbf{in } p : \textbf{integer} \\ \textbf{in } cp : \textbf{channel of integer} \\ \textbf{local } a1, a2 : \textbf{integer} \\ \textbf{local } c : \textbf{channel of integer} \\ \left[\begin{array}{l} cp \Rightarrow a1; \\ a1 := a1 + p; \\ c \Leftarrow a1; \\ \textbf{skip} \end{array}\right] \ || \ \left[\begin{array}{l} cp \Rightarrow a2; \\ c \Rightarrow r; \\ a2 := r + a2; \\ cr \Leftarrow a2 \end{array}\right] \end{array}\right]$$

The declaration of the interface is done in the four top statements of the body. The simplified notation $r ::= P(p)$ will be used for a procedure reference, where r and p stand for the result and parameter lists of the interface.

2.4.2 Procedure Reference Unhiding and Statement Hiding

The meaning of a statement, one of whose substatements is a procedure reference, is unchanged with the *replacement* of the reference substatement by the body of the referred procedure. This replacement consists of three steps

1. Renaming those internal variables and connections, of a copy of the referred procedure algorithm, that share their names with a variable or channel of the embedding statement.
2. Proper replacement of the reference substatement by the copy of the procedure algorithm, with the renamings.
3. Moving the procedure renamed algorithm declarations of variables, **var** section, and of connections, **con** section, to the embedded statement heading.

This equivalence implies that interface variables should be passed by reference at procedure invocation. In the equivalence prover, the *procedure reference unhiding* rule implements this transformation.

Encapsulation, or *hiding*, of a substatement as a procedure is the reverse of unhiding. A name and an interface set for the new procedure, which is to encapsulate the substatement, have to be given.

Encapsulation may hide connections making them internal to the new procedure. When a statement to be encapsulated has communication pairs over a given connection c, within some of its cooperation substatements, and some operation of a pair forms another pair with a matching, parallel, communication outside the statement to be encapsulated, there would be internal and external communication events with respect to the new procedure. This is forbidden, and has to be checked. Henceforth, encapsulation is preceded by the check of some conditions. It has the following operations:

1. Given a statement to be encapsulated and an interface set, check that
   (a) the interface set is formed with statement variables and connections only.
   (b) the statement variables and connections which are not in the proposed interface are used within the statement only. These are named *internal*.
   (c) no connection in the interface set is used for communications within the statement.
2. Move the internal variable and connection declarations, within the embedding statement, so that their scope is the statement to be encapsulated.
3. Form the new procedure. From the given name, interface set, and substatement to be encapsulated, with its just moved heading local declarations.

4. Replace the statement to be encapsulated, with its heading declarations, by the corresponding procedure reference substatement.

In the equivalence prover, the *statement hiding* rule implements this transformation.

2.5 Basic Notions for the Formal Semantics

The soundness of the laws, to be introduced later in this chapter, has been proved in the formal semantics of Manna and Pnueli books, based on fair transition systems (FTS), [MP91, MP95]. Some notions presented in these books are summarized in this section. Some extended notions, needed later in this work are built on top of them and will be introduced in another section. The rest of this section can be skipped if the reader is familiar with the topic.

The meaning of statements will be defined in terms of state transition systems. Then, the equivalence of statements will be based on the equivalence of their associated transition systems. Some basic notions are needed for that, and are introduced in this chapter without much elaboration. The reader is referred to [MP91, MP95].

A *fair transition system* (FTS) S is the tuple $\langle V, \Sigma, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$, where

- V is a finite set of system variables, expressible as $V = Y \cup \{\pi\}$, where π is the *control variable*, and $Y = \{y_1, \ldots, y_m\}$ is the set of *data variables*.
- Σ is the set of states.
- Θ, a satisfiable assertion, is the *initial condition* satisfied by all the initial states of the FTS.
- $\mathcal{T}$ is a finite set of *transitions*.
- $\mathcal{J}$ is the set of *just* (weakly fair) transitions.
- $\mathcal{C}$ is the set of *compassionate* (strongly fair) transitions.

The control variable π contains the values of the program counters of each parallel process, which are control locations, defined below.

A transition $\tau \in \mathcal{T}$ is a relation on pairs of states (s, s'). s' is a τ-*successor* of s. $\tau(s)$ denotes the set of successors of s in the relation. The relation will be specified by a *transition relation* $\rho_\tau(V, V')$, a first order formula which evaluates to true for each state pair of the transition. V and V' correspond to state variables evaluated at s and at s'. It has the general form

$$\rho_\tau(V, V') : E_\tau(V) \wedge D_\tau(V, V')$$

where conjunct E_τ depends only on the initial state s, and conjunct D_τ depends on both states s and s', and specifies the changes made on the variables by the transition. Transitions correspond to atomic actions associated to statements.

A transition is *enabled* or *disabled* on state s when $E_\tau(V)$ evaluates to true or false on s, respectively.

One of the transitions $\tau_I \in \mathcal{T}$ is the *idling* transition. It is such that $s' = s$ for every state s.

A *run* of a FTS is an infinite sequence of states

$$\sigma : \ s_0, \ s_1, \ s_2, \ \ldots$$

satisfying

- *initiation* ($s_0 \models \Theta$) and
- *consecution* ($s_{j+1} \in \tau(s_j)$).

A transition is **just** (*weakly fair*) if when it is continuously enabled in a run, then it is taken eventually in the same run. Consequently it will be taken an indefinite number of times as well.

A transition is **compassionate** (*strongly fair*) if when enabled an indefinite number of times in a run, it is taken eventually in the same run. Consequently it will be taken an indefinite number of times as well.

A *computation* of a FTS is a *run* all of whose transitions satisfy their corresponding fairness requirements, expressed by the sets $\mathcal{J}$ and $\mathcal{C}$.

In order to define a practical notion of equivalence between transition systems it is enough to consider observable parts. Then, a *reduced behavior* σ^r is obtained from a computation σ by retaining an *observable* part, relative to a set of observed variables $\mathcal{O}$, where $\pi \notin \mathcal{O}$, and eliminating from it stuttering steps (equivalent to idling transitions). In other words, deleting any state which equals its predecessor but not all its successors. $\mathcal{R}_\mathcal{O}(\mathcal{S})$ is the set of all reduced behaviors of a transition system S, with respect to the set $\mathcal{O}$ of observed variables.

Two transition systems S_1 and S_2 are equivalent relative to a set $\mathcal{O}$ of observed variables, denoted by $S_1 \sim S_2$, if $\mathcal{R}_\mathcal{O}(S_1) = \mathcal{R}_\mathcal{O}(S_2)$. A system S_c *refines* system S_a, written $S_c \sqsubseteq_\mathcal{O} S_a$, if every reduced behavior of S_c is also a reduced behavior of S_a.

A *program context* $P[\ \_\ ]$ is a program P one of whose statements corresponds to a hole to be filled-in with an arbitrary statement S. With some abuse of notation $P[S]$ will denote a program context, where S denotes the arbitrary statement placed in the hole.

Statement S_1 *refines* S_2, written $S_1 \sqsubseteq_{\mathcal{O}} S_2$, when for any program context $P[\cdot]$, any reduced behavior of $P[S_1]$ is also a reduced behavior of $P[S_2]$.

S_1 is *congruent* to S_2, written $S_1 \approx_{\mathcal{O}} S_2$, when $S_1 \sqsubseteq_{\mathcal{O}} S_2$ and $S_2 \sqsubseteq_{\mathcal{O}} S_1$. Some of the laws are congruence relations between statements.

2.6 Semantics of the Notation

2.6.1 Introduction

In order to define the equivalence used in the laws, it is mandatory to specify the precise meaning of the notation before. Since there are slight variations in SPL throughout the two framework books, and a variant is used in this work, a presentation is necessary. This will be undertaken in this section.

Once this is done, the specific FTS associated to a program will be clear; more specifically, the set of transitions, the runs and the behaviors. After this, the extended notions of *interface behavior* and *interface equivalence* will be introduced; just before the basic laws for interface equivalence are presented.

Before going into the semantics, which will detail the transitions associated to each statement of the SPL variant, some auxiliary notions are needed.

2.6.2 Auxiliary Notions

Following again Manna and Pnueli, a finite set of transitions, and a finite set of control locations is associated with each statement S. An equivalence relation $\sim_L$, defined on statement labels, will put together labels which denote the same control location.

A *location* is an equivalence class of the label relation $\sim_L$. The location corresponding to label ℓ will be denoted by $[\ell]$. It stands for the equivalence class containing ℓ and all the labels that are $\sim_L$ equivalent to ℓ. Usually, ℓ and $\hat{\ell}$ are the pre and post labels of a statement S, its pre and post control locations $[\ell]$ and $[\hat{\ell}]$ are also written as $pre(S)$ and $post(S)$ respectively.

The special variable π, introduced above, will range over sets of locations. Its value on a state denotes all the locations in the program that are active on that state. A state such that $[\ell] \in \pi$ will be referred to as an ℓ-state.

The predicates $pres(U)$ and $move(L, \hat{L})$

$$pres(U) : \textstyle\bigwedge_{u \in U}(u' = u) \qquad \text{and} \qquad move(L, \hat{L}) : L \subseteq \pi \wedge \pi' = (\pi - L) \cup \hat{L}$$

express preservation of the values of the variables in U, and movement of control from the set of control locations L to $\hat{L}$. The *pres* predicate specifies the data variables which are not changed by the transition. The *move* predicate specifies also the set L of control locations which should be active in order for the transition to be enabled, and the set $\hat{L}$ of locations that are active at the end of the transition.

Some notation shortcuts will be introduced in connection with the *move* predicate. As an example the expression $move(\ell, \hat{\ell})$ will be used instead of $move(\{[\ell]\}, \{[\hat{\ell}]\})$. A transition τ_ℓ whose transition relation ρ_ℓ is of the form

$$\rho_\ell : \quad move(L, \hat{L}) \wedge pres(Y)$$

will be referred to as a *skip-type* transition.

2.6.3 Formal Semantics

The table gives the semantics of some basic statements of the notation:

Statement	*Transition Relations*	*Fairness*	*Labels*
ℓ : **skip**; $\hat{\ell}$:	$\rho_\ell : \ move(\ell, \hat{\ell}) \wedge pres(Y)$	$\mathcal{J}$	
ℓ : **nil**; $\hat{\ell}$:			$\ell \sim_L \hat{\ell}$
ℓ : **stop**; $\hat{\ell}$:			
$\ell : \bar{u} := \bar{e}; \hat{\ell}$:	$\rho_\ell : \ move(\ell, \hat{\ell}) \wedge \bar{u}' = \bar{e} \wedge pres(Y - \{\bar{u}\})$	$\mathcal{J}$	
$\ell : \alpha \Leftarrow e; \hat{\ell}$: $m : \alpha \Rightarrow u; \hat{m}$:	$\rho_{<\ell,m>} : \ move(\{[\ell],[m]\}, \{[\hat{\ell}],[\hat{m}]\}) \wedge$ $u' = e \ \wedge \ pres(Y - \{u\})$	$\mathcal{J}$	

The skip statement involves a transition in the underlying fair transition system, but without any effect on the data variables. It moves control and preserves the values of all the variables.

The nil statement can be characterized by contributing no transition, but only the equivalence between its pre and post labels. It has been introduced in this work for convenience in algebraic manipulations.

The stop statement has neither transition nor label relation.

In the assignment statement the values of the variables after the transition, written as $\bar{u}'$, take the values of their corresponding expressions in $\bar{e}$. The values of the rest of the variables are preserved.

The last row of the table above states that the execution of a pair of matching communication statements is atomic and simultaneous. The effect is equivalent to

the assignment $\ell : u := e; \hat{\ell} :$. Notice that this transition is in the just set. This is required for the soundness of the laws as shown in [BBCN01].

The position of a pair of matching synchronous communication statements is such that the above joint transition could be enabled. For instance a send and a receive statement over the same channel may match but two send statements never do. As indicated in subsection 2.2.4 matching communication statements should be parallel.

Given a pair of matching synchronous communication statements, it is said that one *matches* the other. When a joint synchronous communication transition is taken in a computation it is said that a synchronous *communication event* takes place. Two synchronous communication events are *ordered* if they take place in the same order in any computation. For instance, when the four synchronous communication statements giving rise to two communication events are parallel, then the two communication events are not ordered.

Some of the semantics of the compound statements is given in the following table, to be completed in the paragraphs which follow it:

Statement	*Transition Relations*
$\ell : [\ell_1 : S_1; \hat{\ell_1} \ \ldots \ ; \ell_m : S_m; \hat{\ell_m}]; \hat{\ell} :$	
$\ell : [\ [\ell_1 : S_1; \hat{\ell_1}] \ \|\| \ldots \|\| \ [\ell_m : S_m; \hat{\ell_m}] \ ]; \hat{\ell} :$	$\rho^E_\ell : move(\{[\ell]\}, \{[\ell_1], \ldots, [\ell_m]\}) \wedge pres(Y)$ $\rho^X_\ell : move(\{[\hat{\ell_1}], \ldots, [\hat{\ell_m}]\}, \{[\hat{\ell}]\}) \wedge pres(Y)$
$\ell : [\ c_1; S_1$ **or** $\ \ldots \ $ **or** $c_m; S_m \ ]; \hat{\ell} :$	$\rho_i : move(\{\ell\}, \{\ell_i\}) \wedge c_i \wedge pres(Y)$
$\ell : [\ c_1, c(\alpha_1); S_1$ **or** $\ \ldots \ $ **or** $c_m, c(\alpha_m); S_m \ ]; \hat{\ell} :$	$\rho_{<i,n>} : move(\{\ell, n\}, \{\ell_i, \hat{n}\}) \wedge c_i \wedge u' = e \wedge pres(Y - \{u\})$

No transition is associated directly with the concatenation statement. All its transitions are associated with its children statements. The labels ℓ_i and $\hat{\ell_i}$, which are not represented explicitly, are the pre and post labels of substatements S_i. The label relations associated with the concatenation statement are $\hat{\ell_i} \sim_L \ell_{i+1}$ for $i = 1..m-1$, $\ell \sim_L \ell_1$, and $\hat{\ell} \sim_L \hat{\ell_m}$.

The cooperation statement has an entry and an exit transition, τ^E and τ^X associated with it. They are in the justice set $\mathcal{J}$. It also has the transitions associated with its substatements S_i.

Labels n and $\hat{n}$ of the communication selection transitions $\tau_{<i,n>}$ are the pre and post labels of the communication statements matching $c(\alpha_i)$; which form part of some statement parallel to the communication selection. These transitions correspond to synchronous communication events. Their fairness set is $\mathcal{J}$, in accordance with the entry for these joint transitions in the first table above. Pre labels ℓ_i of the communications $c(\alpha_i)$, and post labels $\hat{\ell_i}$ of substatements S_i are not shown explicitly in the table. The label relations associated with this statement are $\ell \sim_L \ell_1 \sim_L \ \ldots \ \sim_L \ell_m$ and $\hat{\ell} \sim_L \hat{\ell_1} \sim_L \ \ldots \ \sim_L \hat{\ell_m}$.

Two transitions τ and τ' are *competing* if both have the same initial location, and taking one of them disables the others. This is the case, for example, when they are directly associated with the same selection statement.

2.7 Interface Behaviors

The following notions are extensions of the semantics used by Manna and Pnueli. They are needed for the definition of interface equivalence, and the substitution rule. A much more elaborated account of this and the following section is available in [BBC05b]. The extensions are motivated by the explicit conservation, in the set of behaviors, of the input/output relation of a procedure, with independence of the intermediate computation orders and of the media through which values are passed; be it either storage or channels.

The set $\mathcal{O}_P$ of observed variables of a procedure Contains all proper variables in the interface of P, i.e. in the result and parameter lists, and an auxiliary *channel variable* for each channel in the interface.

A channel variable records, as a triplet, the values passed at each communication event, an integer mark reflecting their order for each channel, and an input/output mark (i, o). When the communication event is internal, a dot replaces either of these marks. For the procedure Pc of page 24, this set is $\mathcal{O} : \{r, p, cr, cp\}$, where here cr and cp are interpreted as auxiliary variables associated with each respective channel.

An *interface computation* records the changes of both the variables and the channels of a procedure body, a statement, during an execution. It has a row for each change and a column for each variable or channel. It is an extension of the notion of computation, adding to it columns for channels. Whereas a computation is a sequence of states only, an interface computation is a sequence of states where the values crossing channels are also recorded.

Groups of computations will be represented as *schemas*, which have *value variables*. Computations have just values (integers, booleans, etc.).

A possible *interface computation schema* of the Pc procedure, repeated below, is the following:

		r	p	cr	cp	$a1$	$a2$	c
0		x	$p1$	x	x	x	x	x
1	$cp \Rightarrow a1$	x	$p1$	x	$cp1, 1, i$	$cp1$	x	x
2	$cp \Rightarrow a2$	x	$p1$	x	$cp2, 2, i$	$cp1$	$cp2$	x
3	$a1 := a1 + p$	x	$p1$	x	$cp2, 2, i$	$cp1 + p1$	$cp2$	x
4	$c \Leftarrow a1 \| c \Rightarrow r$	$cp1 + p1$	$p1$	x	$cp2, 2, i$	$cp1 + p1$	$cp2$	$cp1 + p1, 1, \cdot$
5	$a2 := r + a2$	$cp1 + p1$	$p1$	x	$cp2, 2, i$	$cp1 + p1$	$cp1 + p1 + cp2$	$cp1 + p1, 1, \cdot$
6	$cr \Leftarrow a2$	$cp1 + p1$	$p1$	$cp1 + p1 + cp2, 1, o$	$cp2, 2, i$	$cp1 + p1$	$cp1 + p1 + cp2$	$cp1 + p1, 1, \cdot$

Table 2.1: An interface computation schema of the Pc procedure

$$(r, cr) ::= Pc(p, cp) :: \left[\begin{array}{l} \textbf{out } r : \textbf{integer} \\ \textbf{out } cr : \textbf{channel of integer} \\ \textbf{in } p : \textbf{integer} \\ \textbf{in } cp : \textbf{channel of integer} \\ \textbf{local } a1, a2 : \textbf{integer} \\ \textbf{local } c : \textbf{channel of integer} \\ \left[\begin{array}{l} cp \Rightarrow a1; \\ a1 := a1 + p; \\ c \Leftarrow a1; \\ \textbf{skip} \end{array}\right] \| \left[\begin{array}{l} cp \Rightarrow a2; \\ c \Rightarrow r; \\ a2 := r + a2; \\ cr \Leftarrow a2 \end{array}\right] \end{array}\right]$$

The same procedure, above, is used as example. It has no selection statements embedding communications, in accordance with the assumptions of this thesis. $p1$, $cp1$, $cp2$, etc., are value variables, whereas $a1$, $a2$, r, and p are program variables. cp, cr, and c are auxiliary channel variables. Giving integer values to $p1$, $cp1$, and $cp2$, specific computations would be obtained. Each row corresponds to the transition associated to the statement specified in the second column. The transition of row 4 is the joint transition of the synchronous communication over channel c. Observe that, an integer count is associated to each new value of a channel variable. In this case there is a *terminal* state, indexed with 6, repeating itself implicitly, resulting from idle transition firings. A proper computation (as in Manna and Pnueli) schema could be obtained from the above by deleting the cr, cp, c, and the two left columns. Then deleting any row which equals its predecessor but not all of its successors.

An interface behavior may be viewed as a trace of the execution of a procedure observed from outside, and including the values which traverse the channels. We assume that $\mathcal{O}_P$ is the procedure interface set, defined naturally from its interface declaration.

Definition 1 (Interface Behavior of a Procedure): The result of deleting, from an interface computation, all columns of variables not belonging to $\mathcal{O}_P$, and then deleting any row which equals its predecessor but not all of its successors.

Due to event counters, all channel events are represented by at least one row in an interface behavior. Therefore, consecutive events are not deleted when their

values are equal. This is necessary in order to retain the input/output relation of the procedure. An interface behavior has one row for each value change of a variable $v \in \mathcal{O}_P$. It has to be a result variable. A parameter variable never changes its value, unless it is also a result. Input and output channel variables exhibit value changes. The following interface behavior schema results from the above interface computation schema, table 2.1.

	r	p	cr	cp
0	x	$p1$	x	x
1	x	$p1$	x	$cp1, 1, i$
2	x	$p1$	x	$cp2, 2, i$
4	$cp1 + p1$	$p1$	x	$cp2, 2, i$
6	$cp1 + p1$	$p1$	$cp1 + p1 + cp2, 1, o$	$cp2, 2, i$

Table 2.2: Interface behaviour schema of the Pc procedure

Rows 3 and 5 have been deleted since they are equal to their predecessors 2 and 4 respectively, but distinct from all its successors. Suppose now that $cp1 = cp2$, then row 2 would not be deleted due to the distinct values of the counter field of the column of channel cp.

Definition 2 (Component of an Interface Behavior): The infinite list of values, a column, corresponding to a variable of the interface behavior. But having deleted any value in the list which equals its predecessor but not to all its successors.

There are both proper and channel variable components.

Definition 3 (Equivalence of Interface Behaviors): Two interface behaviors are equivalent when they share the same interface set, and any channel or variable component of one of them is identical to the homologous channel or variable component of the other.

It is important to observe that the relative order of value changes among different components of the interface behavior is lost. Equivalence only requires that each pair of homologous component lists be identical. This makes equivalent any computation order which shares the same intermediate results.

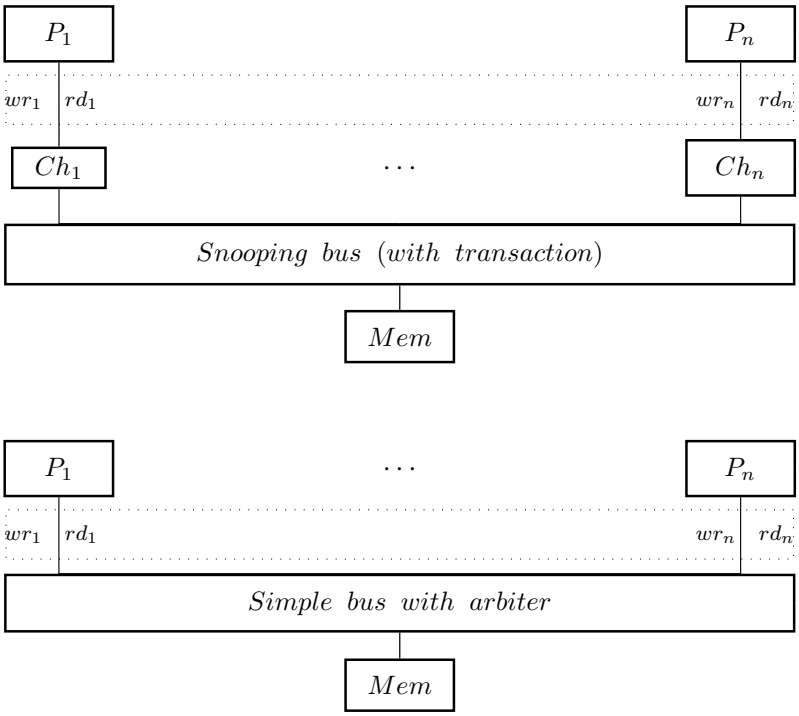
2.8 Interface Equivalence

2.8.1 The Notion

It is a weak equivalence, defined below, within the class of *trace equivalences*, the weakest equivalence given in [dFEGR05, dFEVGR07, dFEGR09]. In it, all computations with the same intermediate results for each variable, but with different relative orders among distinct variable value histories, are made equivalent. This is what was needed for the pipeline processor example, shown in chapter 6.

Definition 4 (Interface-equivalent Procedures): Two procedures P_1 and P_2 are interface equivalent with respect to their common interface set $\mathcal{O}$, written $P_1 =_{\mathcal{O}} P_2$, when any interface behavior of any of them is equivalent, as in definition 3, to an interface behavior of the other.

This notion would also be very appropriate for two important applications: proving consistency of a multiprocessor with caches [CS99], and for correctness proofs of concurrency control algorithms in distributed databases [BHG87]. In both cases it has to be shown that the value histories of certain variables are the same, irrespective of relative orders.



In the former application, the observer set would be formed with the variables of connections communicating each processor (P_i) with its cache (Ch_i), i.e. $\mathcal{O}$:$\{wr_1, rd_1, ..., wr_n, rd_n\}$; then the proof would have to show that the system with caches and snooping bus is interface equivalent to a system without caches but with a simple memory bus with its arbiter. They are the two systems in the figure above.

A stronger equivalence would be interface behavior set equivalence, *b-set equivalence* for short, where homologous behaviors are required to be identical. B-set equivalent procedures are always interface equivalent but not vice versa. The following is a procedure resulting from Pc of subsection 2.4 after elimination of internal channel c. It has the same interface set.

$$(r, cr) ::= Pnc(p, cp) :: \begin{bmatrix} [cp \Rightarrow a1 || cp \Rightarrow a2]; \\ r := a1 + p; a2 := r + a2; cr \Leftarrow a2 \end{bmatrix}$$

The reader may verify that each interface behavior of Pc is an interface behavior of Pnc and vice versa. Therefore, they are b-set equivalent $Pc =_{b-set\ \mathcal{O}} Pnc$. Consider also the procedures:

$$(r1, r2) ::= P1(cp1, cp2) :: \ \big[cp1 \Rightarrow r1; cp2 \Rightarrow r2\big]$$

$$(r1, r2) ::= P2(cp1, cp2) :: \ \big[cp2 \Rightarrow r2; cp1 \Rightarrow r1\big]$$

with the same interface set $\mathcal{O}$. Now the two procedures are not b-set equivalent, $P1 \neq_{b-set\ \mathcal{O}} P2$, but they are interface equivalent, $P1 =_{\mathcal{O}} P2$.

Deadlock, with the meaning that some parallel processes are waiting forever on a communication which no other parallel process will match, ends the value history of some variables. Then interface equivalence would consider equivalent two procedures, one of them with a deadlock and the other ending naturally. For instance:

$$[v ::= A()] \ =_{\{v\}} \ [v ::= B()]$$

where

$$v ::= A() :: \ \big[v := 1; \alpha \Rightarrow v; v := 2\big]$$

and

$$v ::= B() :: \ \big[v := 1\big]$$

where α is an internal connection of A, B terminates naturally whereas A terminates due to deadlock; its last substatement will never be executed.

2.8.2 Deadlock Introduction

The relative order of value changes in distinct components is neglected in interface equivalence. Therefore, substitution of a reference to a procedure by a reference to another procedure, interface equivalent to the first, may introduce deadlock. As an example, if the above $P1$ is parallel to a process which always offers an output via channel *cp1* before offering another output via *cp2*, and $P1$ is replaced by $P2$ in that program, deadlock is introduced, since the order of external communication offerings is the opposite in $P2$.

Therefore, when procedure references are substituted in an equivalence proof, the corresponding deadlock-freeness proof is mandatory. This proof is done, indirectly, by the communication elimination procedure below. See Theorem 4 and the communication elimination procedure of subsection 3.2.3 of chapter 3.

2.8.3 Substitution Rules

The first substitution to consider is the replacement of a procedure reference statement by the body of the procedure, as detailed in section 2.4.2.

Lemma 1 (Procedure Reference Unhiding): Let $S[r := P(p)]$ be a statement, one of whose substatements is the procedure reference $r := P(p)$, with interface set $\mathcal{O}_P$. Let $\mathcal{O}_S$ be the interface set formed with all variables and connections used in S. Let A denote the algorithm of procedure P, and A_{rel} be the algorithm after the relabelings required by the unhiding within S, and the moving of declarations. Then

$$S[r := P(p)] =_{\mathcal{O}} S[A_{rel}]$$

for any $\mathcal{O}$ such that $\mathcal{O} \subseteq \mathcal{O}_S$

A justification could be elaborated on the following considerations. Note first that $\mathcal{O}_P \subseteq \mathcal{O}_S$, and that the $\mathcal{O}_P$-components of $r := P(p)$ are identical to the homologous components of A_{rel}, since $\mathcal{O}_P$ proper variables are passed by reference at procedure invocation, and they are never relabeled; also their changes are determined by A_{rel} in the execution of one side and by A in the execution of the other; but the two A's are $\mathcal{O}_P$-equivalent. A similar argument applies to connection components in $\mathcal{O}_P$. Also, due to relabeling, the unhidden components of A_{rel} are never in $\mathcal{O}_S$.

Another essential step of equivalence reasoning is substitution between reference statements, to two interface equivalent procedures. It is used extensively in the proof of chapter 6. The set of observed variables $\mathcal{O}$ is defined by the referred procedure interface, which is the same for the two equivalent procedures. It is the following:

Lemma 2 (Procedure Reference Substitution): Let $S[r := A(p)]$ be a statement formed with concatenation, cooperation and selection statements, one of whose substatements is the procedure reference which has been highlighted; and with $\mathcal{O}_S$ the observed set formed with all variables and connections used in $S[r := A(p)]$. Let $S[r := A(p)]$ be deadlock-free, and $r := A(p)$ be disjoint with all its parallel substatements in $S[r := A(p)]$. Then, if

$$[r := A(p)] =_{\mathcal{O}} [r := B(p)],$$

and $S[r := B(p)]$ is deadlock-free,

$$S[r := B(p)] =_{\mathcal{P}} S[r := A(p)]$$

for any $\mathcal{P}$ such that $\mathcal{P} \subseteq \mathcal{O}_S$.

$\mathcal{O}$ is the interface set of both A and B, formed from variables and connections in the lists r and p. The deadlock-freeness requirement of $S[r := B(p)]$ is forced by the possibility that $r := A(p)$ has parallel statements in $S[r := A(p)]$ when $\mathcal{O}$ has connection variables.

Since there is the procedure reference unhiding rule stated in subsection 2.4.2 and the equivalence in lemma 1, one way to justify the procedure reference substitution rule is to reason that the bodies of the two equivalent procedures, with their pertinent relabelings, are interchangeable within the embedding statement $S[\_]$. In other words, that

$$S[A_{rel}] =_{\mathcal{O}_S} S[B_{rel}]$$

has to be established, where A_{rel} and B_{rel} are the relabeled algorithms of procedures A and B respectively. A justification could be elaborated on the following considerations. Due to the relabelings implicit in the unhidings, components of interface behaviors which are internal to the two equivalent procedures will have different names from those in $\mathcal{O}_S$. Since there is no deadlock for any behavior of left side, the portion corresponding to $\mathcal{O}$ of A_{rel} can be interpreted as corresponding to $\mathcal{O}$ of B_{rel} and, thus as a behavior of the right side.

2.9 Laws for Interface Equivalence

2.9.1 Introduction

A set of basic laws has resulted from the interface equivalence presented above. Although they do not eliminate communications, they are necessary to transform a

statement into a form where a proper communication elimination law can be applied. This subsection gives a summary of this set.

2.9.2 Repository of Laws

Laws for concatenation and then for cooperation are presented in the first two subsections. The laws hold only assuming weak fairness or no fairness. In other words, all transitions are in the justice set $\mathcal{J}$, no transition is in the compassion set $\mathcal{C}$. The remaining two subsections give laws used in DPS proofs, but after the communication elimination stage.

The soundness proofs of the basic laws given in subsections 2.9.2.1 and 2.9.2.2, in the semantic framework of the Manna and Pnueli books, is available in appendix A; together with the justification of the need of avoiding strong fairness. It contains an updated version of part of [BBCN01]. It does not belong to the thesis; it has been added to make this account more self-contained.

2.9.2.1 Concatenation

Law 1 (Concatenation with Nil):

$$\mathbf{nil}; S \approx S \qquad\qquad S; \mathbf{nil} \approx S$$

Law 2 (Concatenation with Skip):

$$\mathbf{skip}; S \approx S \qquad\qquad S; \mathbf{skip} \approx S$$

The two **skip** laws do not hold when strong fairness is assumed.

Law 3 (Concatenation Associativity):

$$S_1; \cdots; S_k; \cdots; S_l; \cdots; S_n \;\approx\; S_1; \cdots; [S_k; \cdots; S_l]; \cdots; S_n \;,$$

where k and l are integers such that $1 \leq k < l \leq n$.

Law 4 (Concatenation Commutativity): Let $p_m(k)$, where $k = 1..n$, denote the k-*th* integer of a permutation of the list $\langle 1, 2, \ldots, n\rangle$. Let the statements S_1 , $\cdots$, S_n be pairwise disjoint and without external communication statements with the exception of only one of them. Then:

$$S_1;\ \cdots;\ S_n \;=_{\mathcal{O}}\; S_{p_m(1)};\ \cdots;\ S_{p_m(n)}$$

2.9.2.2 Cooperation

Law 5 (Cooperation with Nil):

$$\mathbf{nil} \;||\; S \approx S \qquad\qquad S \;||\; \mathbf{nil} \approx S$$

Law 6 (Cooperation with Skip):

$$\mathbf{skip} \;||\; S \approx S \qquad\qquad S \;||\; \mathbf{skip} \approx S$$

Law 7 (Cooperation Commutativity):

$$S_1||\cdots||S_n \;\approx\; S_{p_m(1)}||\cdots||S_{p_m(n)}$$

where $p_m(j)$, for $j = 1..n$, denotes the *j-th* integer of the permuted list.

Law 8 (Cooperation Associativity):

$$[S_1||\cdots||S_k||\cdots||S_l||\cdots||S_n\,] \;\approx\; [\,S_1||\cdots||\;\; [\,S_k||\cdots||S_l\,]\;\;||\cdots||S_n\,]$$

Laws 5, 6 and 8 do not hold with strong fairness.

2.9.2.3 Cooperation and Concatenation

This and the law of law 10 are used in the communication closed layers framework [dRdBH$^+$01, EF82]. The following illustrates the type of laws that are employed after communication elimination in the sequentialization process.

Law 9 (Loop Forever Unfold):

$$[\,\mathbf{loop\ forever\ do}\ S\,] \;\approx\; [\,S;\mathbf{loop\ forever\ do}\ S]$$

Law 10 (Binary Cooperation and Concatenation): Let the statements S_1, S_2, S_3, and S_4 be non-communicating, have no external communication statement with the exception of at most one of them, and be pairwise disjoint, in the sense that no shared variable is written to. Then:

$$[S_1; S_3] \;||\; [S_2; S_4] \;=_{\mathcal{O}}\; [S_1||S_2];\;\; [S_3||S_4]$$

Law 11 (Cooperation and Concatenation): Let the statements $S_1, \ldots, S_n$ have no communication statements with the exception of at most one of them, and be pairwise disjoint, in the above sense. Then:

$$S_1||\cdots||S_n \quad =_{\mathcal{O}} \quad S_{p_m(1)};\ \cdots;\ S_{p_m(n)}$$

where $p_m(j)$, for $j = 1..n$, denotes the *j-th* integer of the permuted of the list.

2.9.2.4 Elimination of Redundant Variables and Statements

Law 12 (Dummy Assignment Elim-intro):

$$[r := r] \ \approx\ \mathbf{skip}$$

Law 13 (Variable and Assignment Elim-intro): Let v be a variable which is not in the observed set $\mathcal{O}$. Let $S_1(v)$ have only read references to v, and no assignment into any of the variables appearing within e, S_2 have no read reference to v, and be either the last statement within the scope of v or located just before a new assignment to v, with respect to the concatenation order of the program. Then

$$[\ v := e; S_1(v); S_2\ ] \quad =_{\mathcal{O}} \quad [\ S_1(e); S_2\ ]$$

The justification would go along the following line of thought. The assignment of a new value to v in the left hand side has no effect upon any reduced behavior since v is not in the observed set, its effect being only via $S_1(v)$. Due to the conditions imposed upon S_1 and S_2, the value of e in $S_1(e)$ will be the same as in the left hand side. Therefore, in going from one side to the other no variable in the observed set can change its value in any interface behavior.

The rest of this section collects special cases and generalizations which are directly used in the proof of chapter 6.

Law 14 (Simple Variable and Assignment Elim-intro): Let v not belong to the observed set $\mathcal{O}$. Let S_2 satisfy the conditions of law 13. Then

$$[\ v := e; v1 := e_1(v); S_2\ ] \quad =_{\mathcal{O}} \quad [\ v1 := e_1(e); S_2\ ]$$

Law 15 (Simpler Variable and Assignment Elim-intro): Let w and v be variables which are not in the observed set. Let S_1 have references to neither v nor w. Then

$$[\ v := e; S_1; w := v; S_2\ ] \quad =_{\mathcal{O}} \quad [\ w := e; S_1; S_2\ ]$$

Law 16 (Double Variable and Assignment Elim-intro): Let e_1 and e_2 have references to neither v_1 nor v_2. Then

$$[\ v_1 := e_1; v_2 := e_2; S_1(v_1, v_2); S_2\ ] \quad =_{\mathcal{O}} \quad [\ S_1(e_1, e_2); S_2\ ]$$

where $S_1(v_1, v_2)$ has only read references to v_1 and v_2, and S_2 has read references to neither v_1 nor v_2.

This would be derived by two applications of law 13. A more general equivalence follows.

Law 17 (Multiple Variable and Assignment Elim-intro): Let $\bar{v}$, $(v.v_1, ..., v.v_n)$, be a list of n variables, none of them being in the observed set $\mathcal{O}$. Let $S_1(\bar{v})$ have only read references to the variables in $\bar{v}$, and no assignments into any of the variables appearing in $\bar{e}$, S_2 have no read references to them, and be either the last statement within the scope of the variables in $\bar{v}$ or located just before a new multiple assignment to $\bar{v}$, with respect to the concatenation order of the program. Then

$$[\ (\bar{v}) := (\bar{e}); S_1(\bar{v}); S_2\ ] \quad =_{\mathcal{O}} \quad [\ S_1(\bar{e}); S_2\ ]$$

In the following variant, only some variables of the multiple assignment are eliminated.

Law 18 (Multiple variable and assignment partial elim-intro): Let $(v.v_1, ..., v.v_n)$, $(e_1, ..., e_n)$, $S_1(\cdot)$, and S_2 in law 17. Then

$$\begin{gathered}[\ (v.v_1, ..., v.v_i, ..., v.v_j, ..., v.v_n) := (e_1, ..., e_i, ..., e_j, ..., e_n);\ S_1(v.v_i, ..., v.v_j);\ S_2\ ] \\ =_{\mathcal{O}} \\ [\ (v.v_1, ..., v.v_{i-1}, v.v_{j+1}, ..., v.v_n) := (e_1, ..., e_{i-1}, e_{j+1}, ..., e_n);\ S_1(e_i, ..., e_j);\ S_2\ ]\end{gathered}$$

These laws are sufficient to prove the equivalences of the pipelined processor example of chapter 6 and other examples. Completeness has not been studied in this work.

2.10 Conclusion

As a prerequisite for the presentation of the results of this work, the chapter has covered the underlying notions, needed for mathematical justifications, and the notational framework. The notation for expressing distributed system models of hardware and software has been introduced. The presentation of the work is done in a version of SPL, the notation for reactive systems of the Manna and Pnueli books. The specific variant has been introduced in this chapter, since it has some additions

and changes. Another notation, PADD, was used in everyday work and has been introduced as well; both, systems to be transformed, and the formal transformation software tool are written in it. A reason for this notation is historical; another reason is its special tree-like form, which facilitated the understanding and grasp of the large statements which are generated in the inner steps of sequentialization proofs. Both notations are in the Pascal structured programming inheritance.

The notion of *modular procedure*, common to the two notations has been introduced. Proof decompositions in other chapters will be organized around them. The partition of the set of communication operations of a statement into the internal and the external classes stands as a basic assumption in this work. The *set of internal channels* will appear quite often. Modular procedures reflect this partition by declaring external channels in their interface section. External channels cannot be used for communications between parallel substatements in the procedure body. Internal channels are declared within the procedure body. It is believed that this partition does not restrict modeling power, but only imposes some healthy hierarchical structure.

The chapter has also summarized the semantics common to the two notations; the fair transition systems semantics of the Manna and Pnueli books with a little extension: instead of computations and reduced behaviors, *interface computations* and *interface behaviors* are introduced as extensions of the latter two; adding to them the values that are communicated through channels. Then, the interface set, for the behaviors, has auxiliary observed variables for each channel, in addition to the familiar observed data variables. The interface set has also been associated to the interface of a modular procedure in a natural way.

The interface behavior semantics was the ground layer for the justification of both *interface equivalence*, the equivalence in which the laws are formulated, the *procedure reference unhiding* rule, and the *procedure reference substitution rule*. These rules allow the hierarchical organization of proofs, and are used in the proof given in chapter 6.

A set of basic laws has been presented in the chapter as well. Soundness proofs are not included; they may be found in the referred works, where it is shown that many of them do not hold if strong fairness is assumed. This is a basic requirement for all the results in this dissertation.

Altogether the chapter has prepared the reader for the proper communication elimination laws and other elements of sequentialization proofs, which are covered in the following chapter.

Chapter 3

DISTRIBUTED PROGRAM SEQUENTIALIZATION PROOFS

For many applications, communication elimination proofs need a continuation, to simplify the resulting model. DPS proofs are, thus, the communication elimination proofs with a continuation to obtain an equivalent sequential system model. The notions needed to carry out these proofs are presented in this chapter. Different forms of DPS proofs, to be used in the processor example of chapter 6 are presented as well.

One of the contributions of this thesis is the formulation of a set of applicability conditions for the proper communication elimination laws. This chapter presents this topic also; first for bounded communication elimination statements and at the end for more general statements, within the framework of DPS.

3.1 Introduction

Distributed program sequentialization, DPS, is a three step proof procedure applied to a statement, S, that reduces a program with inner parallelism and internal communication statements to an equivalent purely sequential one.

The first step is carried out by a *communication elimination* reduction procedure, presented in chapter 4. When the procedure terminates successfully, the resulting interface equivalent form has parallelism between disjoint substatements but no internal communication statements. For instance, the following is a procedure resulting from Pc, section 2.4 of page 24, after the elimination of internal channel c.

$$(r, cr) ::= Pnc(p, cp) :: \begin{bmatrix} [cp \Rightarrow a1 || cp \Rightarrow a2]; \\ r := a1 + p; \\ a2 := r + a2; \\ cr \Leftarrow a2 \end{bmatrix}$$

It has the same interface set, and each interface behavior of Pc is an interface behavior of Pnc and vice versa, so $Pc =_{\mathcal{O}} Pnc$.

A DPS proof continues with a further step, *parallelism to concatenation transformation*. It is carried out applying permutation laws, such as those of law 11 of page 39, for transforming the parallel compositions of disjoint processes to interface equivalent sequential forms. A sequential program interface equivalent to the initial one is obtained.

The third and last step of DPS proofs is *redundant variable elimination*. State-vector reduction comes with this last step. Both, redundant variables and statements are eliminated; for instance by applying law 13 of page 40. The former usually come from communication buffers, of the original distributed system, which are no longer necessary after their inner communications have been eliminated.

The next section of this chapter presents the subclass of bounded communication statements, the ones that DPS deals with, and the laws and notions needed in the communication elimination step of DPS proofs. The extension of DPS proofs to some classes of non-BC statements is presented in another section at the end of the chapter.

3.2 Communication Elimination Laws and Algorithms

3.2.1 Preliminary Notions

The analysis of communication elimination is started below for some *bounded communication* (BC) statements. This and other required notions are introduced in this section.

Definition 5 (Bounded Communication Statement): A statement S is said to be of *bounded communication* if it meets the following requirements:

1. All its parallel substatements are disjoint, in the sense that they only read their shared data variables, should they have some.
2. Any internal communication is outside statically non-unfoldable iteration bodies.

Iteration with internal communications need to be unfolded to a sequential statement. This unfolding has to be done statically. Execution of a bounded communication statement generates only a finite number of communication events.

From now on, BC stands for *bounded communication*, and BCS stands for *bounded communication statement*. Also S and I denote such a statement and the set of its internal channels, respectively.

Also throughout this work, all parallel processes are assumed to be *disjoint*, in the sense that a variable written in one process is neither written nor read in any of its parallel processes.

Definition 6 (Communication Front): The *communication front* of S, written *ComFront(I,S)*, is the subset of minimal elements of the set of communication statements in its concatenation ordering.

Guards of communication selection statements may be in this set.

Definition 7 (Set of Competing Pairs): The set of *competing pairs* of S, written *CompPairs(I,S)* is, by definition,

$$\{\ (\ell, m) \mid \ell, m \in \mathit{ComFront}(I, S)\ \wedge\ \ell\ \ \mathit{matches}\ \ m\ \}$$

In other words, it is the set of all possible matching pairs formed with statements in *ComFront(I,S)*.

Lemma 3 (Non-Communicating Heading Statements): Let G be either a communication ℓ over $\alpha \in I$ or a communication selection statement in S one of whose guards is a communication statement ℓ over $\alpha \in I$, and $\ell \in \mathit{ComFront}(I, S)$. Let H be a statement in S which precedes G in its concatenation order. Then, H does not communicate with any substatement P of S which is parallel to G.

Definition 8 (Selection-free BC Statement): A selection-free BC statement is a BC statement all of whose internal communications are outside the scope of both selections and communication selections.

The execution of a selection-free BC statement generates a constant finite number of internal communication events. The analysis will be limited to these BC statements. For any of its matching pairs $(\ell, m) \in \mathit{CompPairs}(I, S)$, S always has a cooperation substatement which is the LCA of statements ℓ and m. G^l and G^r are the top statements, in this cooperation, corresponding to ℓ and m, respectively.

Lemma 4 (Standard Form of Pair-embedding Top Statements): Symbol x denotes both l and r, left and right.

- Let $(\ell, m) \in \mathit{CompPairs}(I, S)$, and α be its channel.

- Let G^x, either G^l or G^r, be the top statement, embedding either ℓ or m, of the LCA cooperation of ℓ and m.
- For $k = 0, 1, \ldots$, let T^x_k and P^x_k be bounded communication statements, in general with internal and external communications; and H^x_k be statements which do not have internal communications.
- Let G^x_0 be either one of the communication statements $\alpha \Leftarrow e$ and $\alpha \Rightarrow u$.
- Let $G^x_k = H^x_{k-1}; [G^x_{k-1} || P^x_{k-1}]; T^x_{k-1}$, for $k = 1, 2, \ldots$, be a sequence of statements.
- Then, S can be transformed into a congruent statement such that the embedding top parallel substatements G^x, for $x = l$ and $x = r$, have been replaced by a statement of the form of $G^x_{n_x}$ for some finite integers n_l and n_r, respectively.

Justification The reasoning is made for $x = l$, the other case would be similar. It is clear that the statement G^l_0 can be identified within G^l, as either one of its two possible forms in the lemma. Now, since S is selection-free and BC, G^l_0 can be neither within the scope of any selection statement nor within any proper alternative of a general communication selection statement. Hence, the LCA of G^l_0 is either a concatenation or a cooperation.

In the former case, P^l_0 is the nil statement, H^l_0 and T^l_0 correspond to the statements preceding and succeeding G^l_0 in the concatenation. Hence, G^l_1 can be identified as

$$G^l_1 = H^l_0; [G^l_0 || \textbf{ nil }]; \ T^l_0$$

In the latter case, where the LCA of G^l_0 is a cooperation, if one of its top parallel statements is the other G^r, then $n_l = 0$, $G^l = G^l_0$. Otherwise, P^l_0 corresponds to all the parallel statements, and G^l_1 can be identified as

$$G^l_1 = H^l_0; \ [G^l_0 || P^l_0]; \ T^l_0$$

where H^l_0 or T^l_0 may be **nil**. In the above cases, where G^l_1 has been identified within G^l, the process can be continued. The same reasoning made with G^l_0 can be applied now to G^l_1, either terminating or obtaining G^l_2. Hence, an inductive process can be followed. But this process cannot go on forever since G^l is of finite size. Therefore, it will stop at some $G^l_{n_l}$, after a finite number of iterations n_l, as the lemma states. Congruence with the initial G^l follows from the fact that all the nil statements can be introduced by some of the auxiliary laws, always congruences, cited in chapter 2 and justified in [BBCN01].

□

3.2.2 Elimination Laws for Selection-free BC Statements

The elimination of a single matching pair is considered first. The recursive elimination of all the internal communications of S will be considered later. The simplest case corresponds to $[\ \alpha \Leftarrow e \ ||\ \alpha \Rightarrow u\ ] \approx [u := e]$ which we identify with $[G_0^l \ ||\ G_0^r] \approx G_0$. The following shows a basic communication elimination:

Law 19 (Simple Communication Elimination): Let H^l and H^r be statements which do not have communication statements through synchronous channel α, and T^l and T^r be statements. Then

$$\left[\begin{array}{c} H^l; \\ \alpha \Leftarrow e; \\ T^l \end{array}\right] \ || \ \left[\begin{array}{c} H^r; \\ \alpha \Rightarrow u; \\ T^r \end{array}\right] =_{\mathcal{O}} \left[\begin{array}{c} [\ H^l \ ||\ H^r\ ]; \\ u := e; \\ [\ T^l \ ||\ T^r\ ] \end{array}\right]$$

A proof of the soundness of this law is given in appendix A. Actually, this law is a congruence. The channel variable of α should not belong to $\mathcal{O}$.

As it will be shown later, for the more complex forms the elimination law is defined for an arbitrary $k \geq 0$ as

$$\left[\begin{array}{c} H_k^l; \\ [\ G_k^l \ ||\ P_k^l\ ]; \\ T_k^l \end{array}\right] \ || \ \left[\begin{array}{c} H_k^r; \\ [\ G_k^r \ ||\ P_k^r\ ]; \\ T_k^r \end{array}\right] =_{\mathcal{O}} \left[\begin{array}{c} [\ H_k^l \ ||\ H_k^r\ ]; \\ [\ G_k \ ||\ P_k^l \ ||\ P_k^r\ ]; \\ [\ T_k^l \ ||\ T_k^r\ ] \end{array}\right]$$

where the H statements have no inner communication. When this equivalence is identified with $[\ G_{k+1}^l \ ||\ G_{k+1}^r] =_{\mathcal{O}} G_{k+1}$, a recursive definition of G_k^l, G_k^r, and G_k is obtained. For a given value of $k = k_0$, the corresponding law would be constructed recursively, applying the same equivalence to the inner G_k, which stands for $[G_k^l||G_k^r]$, for $k = k_0, k_0 - 1, \cdots, 1$. Finally, the last inner parallelism $[G_0^l||G_0^r]$ would be replaced by the corresponding right hand side G_0 of the basic congruence given earlier, and the law for $k = k_0$ would thus be obtained. There is a law for any finite integer $k = 0, 1, \cdots$ which may be applied as a reduction from left to right in order to eliminate a single communication pair.

Observe that some substatements, like T_k^l and P_k^r, are parallel in one side but not in the other. This disordering may introduce deadlock. Nevertheless, there are cases where deadlock is not introduced. For instance, for some communication closed layer systems. These systems, together with their laws, are treated in [dRdBH$^+$01], with a semantics different to the one used here. But the laws also hold in our semantics. The following is an example which we need later.

Lemma 5 (Communication-closed-layers): Let the statement pairs (A_1, B_2) and (A_2, B_1) be non-communicating, and $[B_1; A_1]$ be disjoint with $[B_2; A_2]$. Then

$$[\ [B_1; A_1]\ ||\ [B_2; A_2]\ ]\ =_{\mathcal{O}}\ [\ [B_1||B_2];\ [A_1||A_2]\ ]$$

and either both sides are deadlock-free or none of them is.

Justification The only statements which change their concatenation order relation are the pairs which do not communicate. Therefore deadlock can not be introduced, since processes can only wait for internal communications to occur. Also, the same pairs are disjoint as a consequence of the assumptions. This guarantees that variable components do not change. Hence, interface behaviors of both sides remain equivalent. See subsection 2.8.

□

Lemma 6 (G-statement Pairing Equivalence): Let all parallel statements below be disjoint and $\mathcal{O}$ be the union of all variables and channel variables in them, but excluding the variables of internal channels. Let H^l and H^r contain no communication statements over internal channels. Then

$$\left[\begin{array}{c} H^l; \\ [\ G^l\ ||\ P^l\ ]; \\ T^l \end{array}\right] \ ||\ \left[\begin{array}{c} H^r; \\ [\ G^r\ ||\ P^r\ ]; \\ T^r \end{array}\right] \ =_{\mathcal{O}}\ \left[\begin{array}{c} [\ H^l\ ||\ H^r\ ]; \\ [\ G^l\ ||\ G^r\ ||\ P^l\ ||\ P^r\ ]; \\ [\ T^l\ ||\ T^r\ ] \end{array}\right]$$

provided that the following statement pairs do not communicate: (P^l, T^r), (P^r, T^l), (G^l, T^r), (G^r, T^l) . Also, under the same conditions, either both sides are deadlock-free or none of them is.

Justification One of the changes of the concatenation order of the substatements of both sides of the equivalence is due to T^l, which is parallel to H^r, G^r, and P^r in the left but in concatenation with the same statements in the right. However, it remains parallel to T^r in both sides. A similar change takes place in relation to T^r. Due to this, the lemma follows by a two-fold application of lemma 5, the communication restrictions of our lemma, and the fact that the H statements do not have internal communications (see lemma 3).

□

The communication elimination law presented earlier, would be derived by the iterative application of the equivalence of lemma 6, from left to right starting at the outermost level $max(n_l, n_r)$ (see lemma 4) . For the moment, it can be assumed that $n_l = n_r$. The general case is treated after Theorem 5. The restrictions of lemma 6 should be fulfilled at each application. But in addition, in all the other applications, $[G^l||G^r]$ at the right hand side of the equivalence of lemma 6 is reduced to G with the same equivalence, applying it from left to right. Now, the substatements that change order, considered in the justification of lemma 6 above, have P^l and P^r, at the outer level, in parallel. This may be a further source of deadlock. The following lemma formulates the conditions for deadlock prevention in this new situation. Some notation is introduced before.

Definition 9 (Communication Precedence): Let C be a statement which is clear in a given context, and statements A and B be parallel to C. Then, the symbolism $cw(A) \leq cw(B)$ will mean that, within C and in the concatenation order, the communications with A precede or are unordered with all the communications with B.

Lemma 7 (Reduction of G-statement Parallelism): Let the equivalence of lemma 6 be represented as $\bar{G}^l||\bar{G}^r =_{\mathcal{O}} \bar{G}$, where the substatements of the three $\bar{G}$'s and the statements below satisfy the conditions stated in it. Then

$$\left[\begin{array}{l} \left[\ \bar{H}^l \ \|\ \bar{H}^r\ \right]; \\ \left[\ \bar{G}^l \ \|\ \bar{G}^r \ \|\ \bar{P}^l \ \|\ \bar{P}^r\ \right]; \\ \left[\ \bar{T}^l \ \|\ \bar{T}^r\ \right] \end{array}\right] =_{\mathcal{O}} \left[\begin{array}{l} \left[\ \bar{H}^l \ \|\ \bar{H}^r\ \right]; \\ \left[\ \bar{G} \ \|\ \bar{P}^l \ \|\ \bar{P}^r\ \right]; \\ \left[\ \bar{T}^l \ \|\ \bar{T}^r\ \right] \end{array}\right]$$

provided that, within $\bar{P}^l$ and $\bar{P}^r$,

$$cw(P^l) \leq cw(T^r)\ ,\ \ cw(P^r) \leq cw(T^l)\ ,\ \ cw(G^l) \leq cw(T^r)\ ,\ \ cw(G^r) \leq cw(T^l)$$

Also, under the same conditions, either both sides are deadlock-free or none of them is.

Justification In order to obtain the right hand side of the equivalence of this lemma, the equivalence of lemma 6 is applied to the inner parallelism between $\bar{G}^l$ and $\bar{G}^r$. The only statements which are parallel to $\bar{G}^l$ and $\bar{G}^r$ in the left hand side statement are $\bar{P}^l$ and $\bar{P}^r$. Also, they are the only ones which are parallel to $\bar{G}$ in the right hand side statement. But, in the G-statement pairing equivalence, the statements (P^l, T^r), (P^r, T^l), (G^l, T^r), and (G^r, T^l) are parallel in the l.h.s. but

concatenated in the above order in the r.h.s., therefore the communications with these statements within $\bar{P}^l$ and $\bar{P}^r$ must have the same order, should they exist. But this holds if the communication order restrictions of the lemma are fulfilled. Finally, the equivalence follows from lemma 6.

□

All the restrictions of lemma 6, that have to be fulfilled in the iterative application to $[G^l_{n+1}||G^r_{n+1}]$ of the equivalence in it, are gathered in the following

Theorem 1 (Non-communication Restrictions for Eliminability): A set of necessary conditions to be fulfilled by $[G^l_{n+1}||G^r_{n+1}]$ for the eliminability of its communication pair (ℓ, m) is that the following substatement pairs do not communicate

1. (P^l_i, T^r_k) and (P^r_i, T^l_k) for $k \in [0, n]$ and $i \in [0, k]$

2. (T^r_i, T^l_j) for $i, j \in [0, n]$, $i \neq j$

Justification In order to obtain the elimination law stated at the beginning of this section, the equivalence of lemma 6 is applied from left to right for $k = n$ first. At this outermost level, its non communication restrictions apply to the pairs (P^l_n, T^r_n), (G^l_n, T^r_n) and the two symmetric ones (P^r_n, T^l_n), (G^r_n, T^l_n). But G^l_n in the second pair can be split, for $n > 0$, into all of its substatements, giving the restrictions (P^l_i, T^r_n), (T^l_i, T^r_n), for $i = n-1, n-2, \cdots, 1, 0$. Together with the first pair, these can be reexpressed as

$$(P^l_i, T^r_n), \text{ for } i = 0, \cdots, n, \quad \text{and} \quad (T^l_i, T^r_n), \quad \text{for } i = 0, \cdots, n-1, \quad \text{for } n > 0$$

Proceeding similarly with the two symmetric restriction pairs (P^r_n, T^l_n),(G^r_n, T^l_n), the following additional restriction pairs are obtained

$$(P^r_i, T^l_n), \text{ for } i = 0, \cdots, n, \quad \text{and} \quad (T^r_i, T^l_n), \quad \text{for } i = 0, \cdots, n-1, \quad \text{for } n > 0$$

However, similar restrictions have to hold at all levels $k = n, n-1, \cdots, 1$ of application of the above reduction. But for $n = 0$ we have still the restrictions

$$(P^l_0, T^r_0) \ , \ (G^l_0, T^r_0) \ , \ (P^r_0, T^l_0) \ , \ (G^r_0, T^l_0)$$

Putting together all the P-T restrictions, we have that, for each $k = 0, 1, \cdots, n$ the following communication restrictions should hold (P^l_i, T^r_k), and (P^r_i, T^l_k), for

$i = 0, 1, \cdots, k$, which is restriction 1 of the lemma. Putting together all the T-T restrictions, we have: (T_i^l, T_k^r),(T_i^r, T_k^l), for $k = 1, \cdots, n$ and $i = 0, \cdots, k-1$. But this is equivalent to restriction 2 of the lemma. Restrictions (G_0^l, T_0^r) and (G_0^r, T_0^l) can be ignored since G_0^l and G_0^r, which form a matching pair, communicate among themselves only.

□

In a similar manner, all the restrictions of lemma 7 are gathered in the following.

Theorem 2 (Broad Communication Order Restrictions): A set of communication order restrictions to be fulfilled by $[G_{n+1}^l || G_{n+1}^r]$ for the eliminability of its communication pair (ℓ, m), without introducing deadlock, is that for $k \in [1, n]$ and $i \in [0, k-1]$, within P_k^l and P_k^r

$$cw(P_i^l) \leq cw(T_i^r)\ ,\ \ cw(P_i^r) \leq cw(T_i^l)\ ,\ \ cw(G_i^l) \leq cw(T_i^r)\ ,\ \ cw(G_i^r) \leq cw(T_i^l)$$

Justification We keep track of the communication order restrictions of lemma 7 in the recursive application to $[G_{n+1}^l || G_{n+1}^r]$ of the equivalence of lemma 6, as a reduction from left to right. Thus, concerning P_n^l and P_n^r , the second outermost application gives the restrictions

$$cw(P_{n-1}^l) \leq cw(T_{n-1}^r)\ ,\ \ cw(P_{n-1}^r) \leq cw(T_{n-1}^l)$$

and

$$cw(G_{n-1}^l) \leq cw(T_{n-1}^r)\ ,\ \ cw(G_{n-1}^r) \leq cw(T_{n-1}^l)$$

Similarly, the next outermost application gives restrictions on the communications of P_{n-1}^l and P_{n-1}^r but also on those of P_n^l and P_n^r , since the two latter statements are also parallel to P_{n-2} , T_{n-2} and to G_{n-2} , T_{n-2}. These restrictions on the communications within these four P statements are

$$cw(P_{n-2}^l) \leq cw(T_{n-2}^r)\ \ ,\ \ cw(P_{n-2}^r) \leq cw(T_{n-2}^l)\ \ ,\ \ cw(G_{n-2}^l) \leq cw(T_{n-2}^r)\ \ ,\ \ cw(G_{n-2}^r) \leq cw(T_{n-2}^l)$$

Within P_n^l and P_n^r only , and continuing until the last application, at $k = 1$, the following communication restrictions should hold: for $i \in [0, n-1]$,

$$cw(P_i^l) \leq cw(T_i^r)\ \ ,\ \ cw(P_i^r) \leq cw(T_i^l)\ \ ,\ \ cw(G_i^l) \leq cw(T_i^r)\ \ ,\ \ cw(G_i^r) \leq cw(T_i^l)$$

These conditions should also hold within P_k^l and P_k^r , for all $k \in [1, n]$ and $i \in [0, k-1]$, as the lemma states.

□

The set of restrictions of Theorem 2 to be fulfilled for the eliminability of the communication pair (ℓ, m), without introducing deadlock, can be reexpressed as in the next theorem:

Theorem 3 (Communication Order Restrictions for Eliminability): A set of communication order restrictions to be fulfilled by $[G^l_{n+1}||G^r_{n+1}]$ for the eliminability of its communication pair (ℓ, m), without introducing deadlock, is that for all $k \in [2, n]$, within P^l_k and P^r_k,

$$cw(P^l_j) \leq cw(T^r_i)\ ,\ \ cw(P^r_j) \leq cw(T^l_i)\ ,\ \ \text{for } i \in [0, k-1] \text{ and } \ j \in [0, i]$$

$$cw(T^l_j) \leq cw(T^r_i)\ ,\ \ cw(T^r_j) \leq cw(T^l_i)\ ,\ \ \text{for } i \in [1, k-1] \text{ and } \ j \in [0, i-1]$$

and for $k = 1$, within P^l_1 and P^r_1

$$cw(P^l_0) \leq cw(T^r_0)\ ,\ \ cw(P^r_0) \leq cw(T^l_0)$$

Justification The statements G^l_i and G^r_i in the last two conditions of Theorem 2 can be replaced by all their substatements Ps and Ts, with the exception of G^l_0 and G^r_0, obtaining the equivalent conditions: for $k \in [2, n]$ within P^l_k and P^r_k, and for $i \in [1, k-1]$ and $j \in [0, i-1]$

$$cw(P^l_j) \leq cw(T^r_i)\ ,\ \ cw(T^l_j) \leq cw(T^r_i)$$

$$cw(P^r_j) \leq cw(T^l_i)\ ,\ \ cw(T^r_j) \leq cw(T^l_i)$$

The case of $i = 0$ gives the restrictions $cw(G^l_0) \leq cw(T^r_0)$, $cw(G^r_0) \leq cw(T^l_0)$ within P^l_k and P^r_k for $k \in [2, n]$, which need not be included since (G^l_0, G^r_0) is a pair whose two communications communicate between themselves only.

We still have the restrictions for $k = 1$, within P^l_1 and P^r_1 : $cw(G^l_0) \leq cw(T^r_0)$, $cw(G^r_0) \leq cw(T^l_0)$, which can be removed by the same reason as before.

The above P-T restrictions can be put together with the P-T restrictions of Theorem 2, holding for $i \in [0, k-1]$. This results in the following conditions: for $k \in [2, n]$ within P^l_k and P^r_k,

$$cw(P^l_j) \leq cw(T^r_i),\ cw(P^r_j) \leq cw(T^l_i), \text{ for } i \in [0, k-1] \text{ and } j \in [0, i]$$

$$cw(T^l_j) \leq cw(T^r_i),\ cw(T^r_j) \leq cw(T^l_i), \text{ for } i \in [1, k-1] \text{ and } j \in [0, i-1]$$

and for $k = 1$, within P_1^l and P_1^r,

$$cw(G_0^l) \leq cw(T_0^r)\ ,\ \ cw(G_0^r) \leq cw(T_0^l)$$

$$cw(P_0^l) \leq cw(T_0^r)\ ,\ \ cw(P_0^r) \leq cw(T_0^l)$$

as the lemma states. The G-T restrictions are not in the lemma since they always hold, because G_0^l communicates only with G_0^r only and vice versa.

□

Theorem 4 (Elimination from a Standard Form Binary Cooperation): Let $S = [G_n^l||G_n^r]$, be selection-free, and its two top statements have the standard form given in lemma 4. Let $G_0 = [u := e]$, and for $k = 1, 2, \ldots$

$$G_k \ = \ [H_{k-1}^l||H_{k-1}^r]; [G_{k-1}||P_{k-1}^l||P_{k-1}^r]; [T_{k-1}^l||T_{k-1}^r]$$

Then $G_n =_{\mathcal{O}} [G_n^l||G_n^r]$, iff $[G_n^l||G_n^r]$ satisfies the conditions of Theorems 1 and 2. Either both sides are deadlock-free or none of them is.

Justification The equivalence is obtained applying the following steps:

1. Recursive application of the equivalence of lemma 6, starting at $[G_n^l||G_n^r]$ as above, until the following statement is obtained

$$\left[\begin{array}{c}
\left[\ H_{n-1}^l \ ||\ H_{n-1}^r\ \right]; \\
\left[\begin{array}{c}
\cdots \\
\left[\begin{array}{l}
\left[\ H_0^l \ ||\ H_0^r\ \right]; \\
\left[\ [\ G_0^l \ ||\ G_0^r\ ] \ ||\ P_0^l \ ||\ P_0^r\ \right]; \\
\left[\ T_0^l \ ||\ T_0^r\ \right]
\end{array}\right] \ ||\ \cdots \\
\cdots
\end{array}\right]; \\
\left[\ T_{n-1}^l \ ||\ T_{n-1}^r\ \right]
\end{array}\right]$$

2. Application, to its inner statement $[G_0^l||G_0^r]$, of the congruence $[G_0^l||G_0^r] \approx G_0$, as a reduction from left to right.

Thus, the equivalence $G_n =_{\mathcal{O}} [G_n^l||G_n^r]$ is a direct consequence of lemma 6 and the congruence of step 2. In the present scenario of disjoint processes communicating only via synchronous communications, the only possible cause of deadlock is waiting at communications that can never take place. This can only happen with communications within substatements that change from being parallel in $[G_n^l||G_n^r]$ to being concatenation ordered in G_n. The possible situations are captured by Theorems 1 and 2. Deadlock-freeness follows from the satisfaction of the conditions stated in them. In any cases S may be deadlock-free but some of the applicability conditions fail.

□

3.2.3 Elimination Algorithm for Selection-free BC Statements

Given a general selection-free BCS with a non-empty set of competing pairs, the elimination of any pair is feasible under the conditions of the following

Theorem 5 (Elimination from a Selection-free BCS): Let $p = (\ell, m)$ be one of the pairs in *CompPairs(I,S)*, and the top statements of the LCA parallelism of ℓ and m be G^l and G^r. Then S can be transformed into an interface equivalent statement without p if the standard forms of order $n = max(n_l, n_r)$ of the two top statements satisfy the conditions of Theorems 1 and 2.

In general, the orders n_l and n_r of the standard forms of G^l and G^r will not be equal. Then, if $n_l > n_r$ we make $n = n_l$ and construct G_n^r by inserting $n_l - n_r$ layers of nil H, P, and T statements immediately around $G_{n^r}^r$. One proceeds similarly in the opposite case. The insertion can be done in other ways, but we have chosen the outermost one, which preserves the input statement form as it is.

Justification Due to commutativity and associativity of parallelism, S can always be transformed into $S[G^l||G^r]$, where the binary parallelism of the embedding top statements has been isolated. Then $S[G^l||G^r] =_{\mathcal{O}} S[G_n^l||G_n^r] =_{\mathcal{O}} S[G_n]$ by lemma 4, Theorem 4, and monotonicity of interface equivalence.

□

Assuming that the conditions of Theorems 1 and 2 hold, the term *Elim*$\{(\ell, m)$,$S\}$ will represent the statement resulting after elimination of (ℓ, m) from S. Thus, the result of the above theorem may be written as $S =_{\mathcal{O}}$ *Elim*$\{(\ell, m)$,$S\}$.

Lemma 8 (Elimination Commutativity of Disjoint Pairs): Let p_1 and p_2 be two disjoint competing pairs of S. Then

$$Elim\{p_2, Elim\{p_1, S\}\} =_{\mathcal{O}} Elim\{p_1, Elim\{p_2, S\}\}$$

Justification One has that $S =_{\mathcal{O}} Elim\{p_1, S\}$ and $S =_{\mathcal{O}} Elim\{p_2, S\}$. But, for the same reason

$$Elim\{p_2, S\} =_{\mathcal{O}} Elim\{p_2, Elim\{p_1, S\}\} \text{ and}$$
$$Elim\{p_1, S\} =_{\mathcal{O}} Elim\{p_1, Elim\{p_2, S\}\}$$

The desired result follows, since the left hand sides of the last two equivalences are both equivalent to S. □

Lemma 9 (Elimination of a Set of Disjoint Competing Pairs): Let n_{cp} be the cardinality of *CompPairs(I,S)*, all of whose pairs $cp_i, i = 1, \cdots, n_{cp}$ are disjoint. Then,

$$S =_{\mathcal{O}} Elim\{cp_1, Elim\{cp_2, \cdots, Elim\{cp_{n_{cp}}, S\}\cdots\}\}$$
$$=_{\mathcal{O}} Elim\{cp_{p(1)}, Elim\{cp_{p(2)}, \cdots, Elim\{cp_{p(n_{cp})}, S\}\cdots\}\}$$

where $< p(1), \cdots, p(n_{cp}) >$ is any permutation of $< 1, \cdots, n_{cp} >$.

Justification This follows by linear induction. The base case, where $n_{cp} = 2$ holds by lemma 8. For the induction step, assume that the result is true for $n_{cp} = k$, then $S =_{\mathcal{O}} Elim\{cp_{p(1)}, Elim\{cp_{p(2)}, \cdots, Elim\{cp_{p(k)}, S\}\cdots\}\}$ for any permutation $< p(1), \cdots, p(k) >$ of the first k integers. But any permutation of the first $k+1$ integers can be obtained from a suitable permutation of the first k integers by inserting the integer $k+1$ at a convenient position l. Also,

$$Elim\{cp_{(k+1)}, S\} =_{\mathcal{O}} S$$
$$=_{\mathcal{O}} Elim\{cp_{(k+1)}, Elim\{cp_{p(1)}, Elim\{cp_{p(2)}, \cdots, Elim\{cp_{p(k)}, S\}\cdots\}\}\}$$

After some applications of lemma 8, $cp_{(k+1)}$ can be moved to the l-th position. □

Assuming that all the pairs are mutually disjoint, the following communication elimination algorithm is a consequence of the above results:

failure := F
while ¬*failure* ∧ { S has a competing pair p}
  do (*failure,S*):= $PElim(p, S)$;
if ¬*failure*
  then if *ComFront*(I, S)=∅
    then terminate with success
    else terminate with deadlock
  else terminate with failure

Procedure *PElim* is the extension of *Elim* which checks applicability conditions. It transforms G^l and G^r into standard form, as in the proof of lemma 4. After structure matching and application of the law, nil statements are eliminated with the basic congruences. When a true boolean result is returned, the applicability conditions were not satisfied. When the loop terminates without failure, *Comp-Pairs(I,S)* of the final statement is empty. When at the same time there is still some communication left in the communication front, this indicates that no match can be found for it. Then the initial statement is not deadlock-free.

3.3 Extensions of DPS

3.3.1 DPS for Non-BC Statements. The Fundamental Proof

There exist many types of non-BC statements, where communications appear within indefinite loops. Attention will be confined only in the following very common structure: $S = [S_1 || \cdots || S_m]$, where the S_k's are of the form $S_k =$ **loop forever do** B_k. The B_k's are BC statements. Since they have communication statements and appear within indefinite iterations, the whole statement is non BC.

Assume that the loop of each top substatement S_k is unfolded n_k times, thus obtaining the statement

$$S_u = [B_1^{n_1}; S_1 || \cdots || B_m^{n_m}; S_m]$$

where the $B_k^{n_k}$'s stand for the concatenation of n_k copies of B_k : $B_k; \ldots ; B_k$.

DPS can be applied to S_u partially, only considering its internal communications in the $B_k^{n_k}$ statements. Assume that we succeed and obtain $B; E$, where B has no internal communication but the ending statement E is non-BC, it may have both parallelism and inner communication. Assume also that $B; E$ is also reduced by DPS, partially as before, to $B; B; E$. Then, as a consequence of linear induction,

$$S =_{\mathcal{O}} [B^n; E]$$

for any finite integer n, where B^n is both inner parallelism and communication free. In the frequent case where the first elimination yields $B; S$, i.e. $E = S$, then $S =_{\mathcal{O}}$ **loop forever do** B and the right hand side statement has no inner communication. In many practical systems this occurs already for $n_k = 1$; $k = 1, \ldots , m$.

3.3.2 Hierarchical Proof Organization Around Procedures

Without loss of generality, statement S of last subsection may be regarded as the body of a procedure P_p. The top statements of S may also be embedded within procedures P_s^i, for $i = 1, \ldots, m$, and internal channel declarations added. Under this formulation, the body of $r := P_p(p)$ is of the form

$$DI\ [[r^1 := P_s^1(p^1)]|| \cdots ||[r^m := P_s^m(p^m)]]$$

where DI is the declaration of the set of *internal channels*, and the $r^i := P_s^i(p^i)$ are sequential procedures. Their interface channels belong either to I, the set of internal channels, or to the interface channels of P_p. As usual, the r's and the p's stand for the result and parameter lists of procedure interfaces. With this new formulation, the fundamental proof establishes the equivalence

$$[r := P_s(p)]\ =_{\mathcal{O}}\ [r := P_p(p)]$$

between two procedures, with the same interface, the above $r := P_p(p)$ and $r := P_s(p)$, referred to as *sequential*, whose body is of the form $B^n; E$ as in last subsection.

By the substitution rule, the equivalence

$$DI\ [[r^1 := P_s^1(p^1)]|| \cdots ||[r^m := P_s^m(p^m)]]\ =_{\mathcal{O}}\ P_s$$

is also established. P_s stands for the body of the sequential procedure.

3.3.3 Hierarchical DPS Proofs

Its objective is to establish an equivalence similar to the one of the fundamental proof. However, in this new situation, the body of the starting procedure $r := P_p(p)$ is of the form

$$DI\ [[r^1 := P_p^1(p^1)]|| \cdots ||[r^m := P_p^m(p^m)]]$$

where the P_p^i's have inner parallelism and communication as well, thus expressing a hierarchy of parallelism. The proof steps are the following:

Step 1 Establish the equivalences $[r^i := P_p^i(p^i)]\ =_{\mathcal{O}_i}\ [r^i := P_s^i(p^i)]$ for $i = 1, \ldots, m$, applying the fundamental proof m times. Then, the obtained P_s^i's are sequential, as defined above.

Step 2 If Step 1 succeeds, apply the established equivalences to replace the parallel by the sequential procedure references in the original program, thus obtaining a program of the form needed for the application of the fundamental proof.

Step 3 Apply again the fundamental proof to the resulting program, obtaining an equivalent sequential form P_s.

Lemma 10 (Hierarchical Proof): When all the steps of the hierarchical sequentialization proof succeed, the following holds

$$DI\ [[r^1 := P_p^1(p^1)]\ ||\cdots||\ [r^m := P_p^m(p^m)]]\ =_{\mathcal{O}}\ P_s$$

Justification Step 2 is justified by the rule of substitution of equivalent procedure references of lemma 2, provided that Step 3 succeeds, thus showing that the original and resulting statements are deadlock-free. Step 3 can be applied since the result of Step 2 is of the form required by the fundamental proof.

□

3.3.4 Hierarchical DPS Proofs with Channel Hiding

Hierarchical proofs are simplified if channels are grouped, and the group can be hidden under a newly defined abstract channel, to represent the group. This can be done when the sends and receives of the channels in a group appear in the same pattern everywhere in the program. For instance, they are everywhere composed in sequence and in the same order. Although many other different patterns may be considered, without loss of generality this possibility will be assumed.

In this situation, a new channel α can be introduced to represent the group. Its type has to be a structure, product, with as many components as channels α_i in the group, typed as their corresponding channels. The global correspondence is $\alpha \longleftrightarrow (\alpha_1, \ldots, \alpha_m)$, which defines the *hide* and the *unhide* functions, and implies the correspondences

$$[\alpha \Leftarrow v] \longleftrightarrow [\alpha_1 \Leftarrow v.v_1; \cdots; \alpha_m \Leftarrow v.v_m]$$

and

$$[\alpha \Rightarrow v] \longleftrightarrow\ [\alpha_1 \Rightarrow v.v_1; \cdots; \alpha_m \Rightarrow v.v_m]$$

for the send and receive statements. The type of v is the cartesian product of the types of the α_i's, i.e. $v.v_i$ is of the type of α_i. Notice that the two sides of the correspondences are functionally equivalent. In both the same product value is received and stored into, or retrieved and sent from, v.

The hierarchical proof with channel hiding is a variant of the hierarchical proof. At a certain stage of it, some channels are grouped and hidden under a new abstract channel, as explained above. The body $r := P_p(p)$ of the original parallel procedure of the proof will be denoted now as

$$DI\ [[\bar{r}^1 := \bar{P}_p^1(\bar{p}^1)]||| \cdots |||[\bar{r}^n := \bar{P}_p^n(\bar{p}^n)]]$$

Also, $\bar{r}^i := \bar{P}_s^i(\bar{p}^i)$ will denote the result of the fundamental proof applied to $\bar{r}^i := \bar{P}_p^i(\bar{p}^i)$. The superbars denote unhidden procedure interface lists and bodies. However, distinctive of this variant is the fact that groups of internal channels, with the correspondences for sends and receives defined above, can be formed in the result and parameter lists, $\bar{r}^i$ and $\bar{p}^i$, of all the $\bar{P}_s^i$'s. For this to be so, the following should hold:

1. If a channel belongs to a group and it appears in the result and parameter lists of the same or of two distinct procedure interfaces, then it can be made to belong to the same group in all these interface lists.

2. The send and receive statements of a group of channels appear in the same pattern within the bodies of procedures having the group in their interfaces. As mentioned above, it is assumed that they are always composed in sequence and in the same order.

Under these circumstances, hide and unhide functions introduced above in this subsection can be established for all procedures $\bar{r}^i := \bar{P}_s^i(\bar{p}^i)$, of the original parallel composition statement.

Hierarchical proof with channel hiding

It starts with the statement:

$$DI\ [[\bar{r}^1 := \bar{P}_p^1(\bar{p}^1)]||| \cdots |||[\bar{r}^n := \bar{P}_p^n(\bar{p}^n)]]$$

Its steps are the following:

Step 1 Obtain $\bar{r}^i := \bar{P}_s^i(\bar{p}^i)$ from $\bar{r}^i := \bar{P}_p^i(\bar{p}^i)$ via the fundamental proof, for all i.

Step 2 Obtain the statement:

$$DI\ [[\bar{r}^1 := \bar{P}_s^1(\bar{p}^1)]||\cdots|||[\bar{r}^n := \bar{P}_s^n(\bar{p}^n)]]$$

from the initial statement, via substitution using the equivalences resulting from Step 1.

Step 3 Define the channel hiding functions which will transform the $\bar{r}^i := \bar{P}_s^i(\bar{p}^i)$ into the $r^i := P_s^i(p^i)$. Apply them at once to the statement of Step 2, to obtain:

$$DI\ [[r^1 := P_s^1(p^1)]||\cdots|||[r^n := P_s^n(p^n)]]$$

Step 4 Apply the fundamental proof to the last statement, obtaining a sequential form P_s.

Lemma 11 (Hierarchical Proof with Channel Hiding): When the steps of the hierarchical proof with channel hiding succeed, the following equivalence holds

$$DI\ [[\bar{r}^1 := \bar{P}_p^1(\bar{p}^1)]||\cdots|||[\bar{r}^n := \bar{P}_p^n(\bar{p}^n)]]\ =_{\bar{\mathcal{O}}}\ P_s$$

Justification The statement resulting from Step 2 is equivalent to the initial one due to the procedure reference substitution rule of lemma 2. Since the channel hiding transformation preserves equivalence, as reasoned above in this subsection, the statement resulting from Step 3 is equivalent to the one resulting from Step 2. This is also so since no hidden channel belongs to the interface of the original statement. The equivalence of the lemma follows from the success of the fundamental proof in Step 4. □

3.4 Conclusion

The notion of bounded communication (BC) statement has been introduced in this chapter. The execution of a BC statement gives a finite number of inner communication events. BC statements, whose inner communication operations are outside the scope of selection substatements, form the base class for which communication elimination has been studied. Extension of communication elimination and sequentialization proofs to some non-BC statements, whose execution generates an infinite number of inner communication events, has been treated at the end of this chapter.

The communication elimination algorithm evolves by eliminating matching communication pairs whose elements are in the *communication front* of the statement, the set of minimal elements in its communication operation order. The set of *competing pairs*, as the set of matching pairs taken from the communication front, has been introduced in the chapter together with the communication front.

The precise formulation of *applicability conditions* for the proper communication elimination laws is one of the contributions of this dissertation. It has been presented in this chapter. There is a linear infinite set of laws conforming to the possible structures of nested parallelisms encountered in models of general distributed programs and hardware. Applicability conditions have an iterative structure reflecting the layers of nested parallelisms; they restrict inner communication ordering and location in substatements and warrant that deadlock is not introduced when applying the law; since some parallelism has to be lost in this application.

The chapter has also reviewed important elements of sequentialization proofs. Among them, a communication elimination proof construction algorithm has been studied. It applies both basic and proper communication elimination laws intending to obtain a statement free of inner communications. It has been mathematically justified and its collateral deadlock analysis capabilities have been addressed. The topics of hierarchical proof organization and channel hiding, needed in the pipelined processor proof of chapter 6, have been covered in the chapter as well.

Chapter 4

A COMMUNICATION ELIMINATION REDUCTION PROCEDURE

A specific communication elimination reduction procedure for bounded communication statements is described in this chapter. It is based on the communication elimination laws introduced in the previous chapter. The elimination takes place only when the applicability conditions of Theorems 1 and 2 are satisfied. The chapter provides enough detail to help understand the corresponding programs within the prover. A communication elimination example is detailed at the end.

4.1 Introduction

The communication elimination reduction procedure is detailed step by step. The elimination of a single matching pair of communication statements from a binary cooperation statements is treated first. This gives a detailed implementation to what was referred to as $Elim\{(\ell, m),\ S\}$ in section 3.2.3. The extension to n-ary cooperation statements is given at the end of this chapter. The chapter adds algorithmic details to the schema given in chapter 3.

Before explaining and detailing the communication elimination procedure, some notions are reviewed first. In chapter 3 the notion of *bounded communication statements* has been introduced. These are selection-free and loop-free statements, whose substatements are concatenations, parallelisms and basic substatements only. From now on in the text, bounded communication statements will be referred also as *the input program*. S will denote such a statement.

Moreover the following requirements must hold:

- BC statements are free of scoped descriptions or comments. These are not necessary for equivalence reasoning purposes.

- All variable and synchronous channel declarations are outside the scope of BC statements. Any declaration within the bounded statement must be moved outside of it. If necessary, variable renaming must be applied.

When the bounded statement satisfies the above definitions, called *well formed*, the communication elimination reduction can be applied.

An important requirement of the elimination reduction is that any transformation of the input BC statement be carried out by application of laws only. These laws should have been justified beforehand by mathematical arguments. A repository stores the laws. The procedure transforms the input program only via the APPLY procedure, introduced later in this chapter, which transforms a part of the program applying a law of the repository. The program, a part indicator, and the law are passed as parameters. Each law can be applied from left to right or vice-versa, and for each application a set of applicability conditions has to hold. These are also checked by procedure APPLY.

Within the elimination procedure, the following notation is used sometimes:

$$\left\{ [\ A\ ] => [\ B\ ] \right\}$$

It denotes the application of the transformation to A whose result is B. The symbol $=>$ corresponds to an explicit application of either a law or a set of laws or some *transformation procedures*. The *transformation procedures*, which apply only a set of laws or lemmas, are detailed in chapter 5.

Basically the communication elimination procedure removes a matching pair of communication statements, (ℓ, m), from S, a bounded statement. Both ℓ and m are locations within S that identify the communication statements. The matching pair belongs to $CompPairs(I, S)$, where I is the set of *internal* channels of S. S can always be rewritten so that $S :: [S^l(\ell)||S^r(m)]$. This is the binary parallelism that matches the elimination law. The reasons for this are given in the justification of Theorem 5 of chapter 3. $S^l(\ell)$ and $S^r(m)$ are the top statements of the LCA of both, which contain statements ℓ and m respectively.

Some nomenclature relating to the top statements $S^l(\ell)$ and $S^r(m)$ is introduced. The integer k^l stands for k of $S^l(\ell)$. Similarly for k^r.

Definition 10 (Structural Order): Given a bounded statement S, the *structural order* of its substatement $S^l(\ell)$ is the integer k^l that counts recursively the number of G^l_k statement levels within $S^l(\ell)$, as in law 19 of page 47.

Similarly for $S^r(m)$ and k^r.

Before explaining the procedure some more concepts are defined:

Definition 11 (Immediate Parallel Statement): Given a bounded statement S and a statement S_0 within S, S_0 has an immediate parallel statement if there exists any statement in parallel with it with the same LCA.

For instance, $S :: [S_0 \parallel S_1]$, S_1 is in parallel with S_0 and both have the same LCA, then S_1 is an immediate parallel of S_0.

S_0 and S_1 are immediate parallel statements if both are top statements of the same cooperation.

Definition 12 (Immediate Sequence Predecessor): Given a bounded statement S and a statement S_0 within S, S_0 has an immediate sequence predecessor if there exists a preceding statement in sequence with it.

For example $S :: [S_1;\ S_0]$.

Definition 13 (Immediate Sequence Successor): Given a bounded statement S and a statement S_0 within S, S_0 has an immediate sequence successor if there exists any succeeding statement in sequence with it: $S :: [S_0;\ S_1]$

4.2 Binary Communication Elimination

The goal of this procedure, shown in next page, is to eliminate a matching pair of communication statements in $CompPairs(I, S)$ from a binary cooperation statement S by the application of Theorem 5. S is a bounded communication statement. Whenever this is not possible no transformation takes place and the procedure reports a failure. Otherwise it returns as output a bounded communication statement S'.

In STEP1 of the procedure the orders of the two parallel statements are obtained. Once the procedure has constructed the top level statements in STEP2, the communication elimination law, with the following general form,

$$\left[\begin{array}{c} H_k^l; \\ [\ G_k^l \ \|\ P_k^l\ ]; \\ T_k^l \end{array}\right] \parallel \left[\begin{array}{c} H_k^r; \\ [\ G_k^r \ \|\ P_k^r\ ]; \\ T_k^r \end{array}\right] =_{\mathcal{O}} \left[\begin{array}{c} [\ H_k^l \ \|\ H_k^r\ ]; \\ [\ G_k \ \|\ P_k^l \ \|\ P_k^r\ ]; \\ [\ T_k^l \ \|\ T_k^r\ ] \end{array}\right]$$

can be applied, if the applicability conditions hold, obtaining the target statement S', STEP3, where the matching pair (ℓ, m) has been removed. Next subsections describe each step in detail.

Procedure BIN-COMELI − binary communication elimination

Input: $S :: [S^l(\ell)||S^r(m)]$ a bounded statement, and $p = (\ell, m)$ a matching communication pair, with communication statement ℓ within $S^l(\ell)$ and communication statement m within $S^r(m)$.

Output: S' a bounded statement equivalent to S, where p has been eliminated, or a failure indication.

STEP1: Determine the orders k^l and k^r of $S^l(\ell)$ and $S^r(m)$, respectively.

STEP2: Construct, via law applications, top level statements $G^l_n(\ell)$ and $G^r_n(m)$ equivalent to $S^l(\ell)$ and $S^r(m)$, respectively, and whose structures match G^l_k and G^r_k of chapter 3 for $k = n$, the maximum of the two orders, k^l and k^r.

STEP3: Proper application, from left to right, of the law of Theorem 4 for order n to $[G^l_n(\ell)||G^r_n(m)]$ to obtain S'.

STEP4: Elimination of redundant *nil* statements from S' by applying simple laws.

STEP5: Elimination of redundant sequence and parallel associations introduced in the H^x_k, P^x_k and T^x_k statements.

4.2.1 Determine the Orders

This corresponds to a preliminary processing of the input statement S, needed for the transformation to the standard forms of law 19 of page 47.

Procedure STEP1 − determine the orders of $S^l(\ell)$ and $S^r(m)$

Input: $S :: [S^l(\ell)||S^r(m)]$ a bounded statement, and $p = (\ell, m)$ a matching communication pair.

Output: Integers k^l, k^r, and n.

$k^l :=$ STRUCTORDER(S, ℓ)

$k^r :=$ STRUCTORDER(S, m)

n := max(k^l, k^r)

Basically the procedure collects structural information of S, by identifying the orders k^l and k^r of the statements $S^l(\ell)$ and $S^r(m)$. Procedure STRUCTORDER of page 70 calculates the orders.

Observe that the top parallel statements, G^x, contain some characteristic locations that help the procedure to determine the correct order for any input form. These are also locations of statement S. These are the following (see figure on top of page 69):

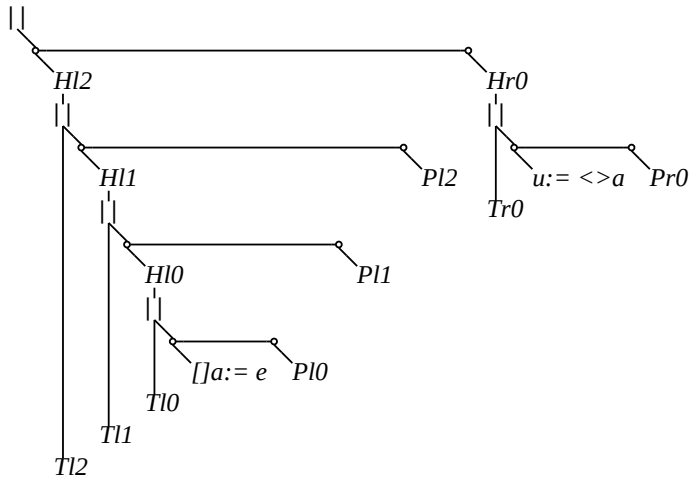
- ℓ_{LCA}: It corresponds to the initial location of the binary cooperation statement, $\ell_{LCA} : [S^l(\ell)||S^r(m)]$.
- $\ell_{G_k^x}$: Initial location of G_k^x statements, where $x = l$ or r, and $k = 0, 1, ..., n$. G statements are those that contain either ℓ or m. For instance the inner location, $\ell_{G_0^l}$, is related to ℓ, of the matching pair. If ℓ denotes a pure communication statement then $\ell_{G_0^l} = \ell$.

  The existence of the input communication pair, (ℓ, m), guarantees that at least $\ell_{G_0^x}$ exists.
- $\ell_{P_k^x}$: Initial location of the P_k^x statements, where $x = l$ or r, and $k = 0, 1, ..., n$. The location identifies the P statements that are in parallel with a G statement.
- $\ell_{GP_k^x}$: Initial location of the $[G_k^x||P_k^x]$ statements, where $x = l$ or r, and $k = 0, 1, ..., n$.
- $\ell_{H_k^x}$: Initial location of the H_k^x statements, where $x = l$ or r, and $k = 0, 1, ..., n$. The H statements can easily be identified as the immediate sequence predecessors of $\ell_{GP_k^x}$. In case that the predecessors be a sequence composition such as $S_1^x; \cdots; S_m^x$, $\ell_{H_k^x}$ identifies the whole composition.
- $\ell_{T_k^x}$: Initial location of the tail statements, T_k^x, where $x = l$ or r, and $k = 0, 1, ..., n$. This is identified as the immediate sequence successor of $\ell_{GP_k^x}$. As above $\ell_{T_k^x}$ identifies any sequential composition that follows in sequence the $[G_k^x||P_k^x]$ statements.

These locations are identifiable if the top statements are in their normal form as in lemma 4 of page 45. In case of H, P and T statements do not exist then some of the above locations are undetermined.

Example This example illustrates the locations on a binary cooperation S. The left-hand side contains the base statement $G_0^l = [\ \alpha \Leftarrow e\ ]$ and the right-hand side $G_0^r = [\ \alpha \Rightarrow u\ ]$, where α is the synchronous channel.

The following shows the S statement in PADD notation:



and in SPL notation:

$$\left[\begin{array}{l} \left[\begin{array}{l} H_2^l; \\ \left[\begin{array}{l} \left[\begin{array}{l} H_1^l; \\ \left[\begin{array}{l} \left[\begin{array}{l} H_0^l; \\ [\ \alpha \Leftarrow e \ \ \| \ \ P_0^l\ ]; \\ T_0^l \end{array}\right] \ \| \ P_1^l \end{array}\right]; \\ T_1^l \end{array}\right] \ \| \ P_2^l \end{array}\right]; \\ T_2^l \end{array}\right] \ \| \ \left[\begin{array}{l} H_0^r; \\ [\ \alpha \Rightarrow u \ \ \| \ \ P_0^r\ ]; \\ T_0^r \end{array}\right] \end{array}\right]$$

Next figure shows all the locations within S, in SPL notation:

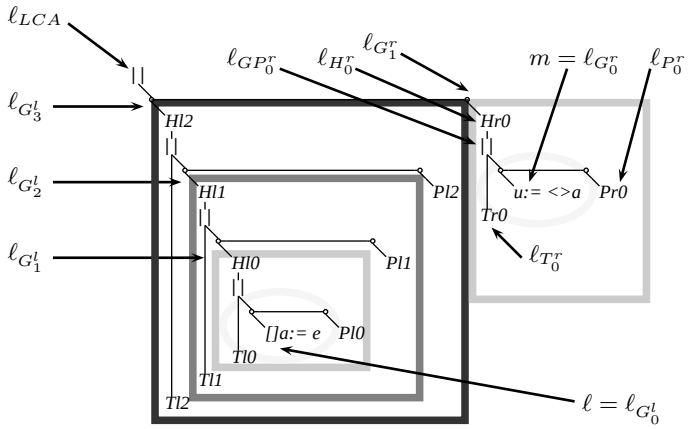
$$
\ell_{LCA}:
\left[
\begin{array}{l}
\ell_{H_2^l}: H_2^l; \\
\ell_{G_3^l}: \left[
\begin{array}{l}
\ell_{GP_2^l}: \left[
\begin{array}{l}
\ell_{G_2^l}: \left[
\begin{array}{l}
\ell_{GP_1^l}: \left[
\begin{array}{l}
\ell_{G_1^l}: \left[
\begin{array}{l}
\ell_{GP_0^l}: \left[\ell: \alpha \Leftarrow e \quad || \quad \ell_{P_0^l}: P_0^l \right]; \\
\ell_{T_0^l}: T_0^l
\end{array}
\right] \quad || \quad \ell_{P_1^l}: P_1^l
\end{array}
\right]; \\
\ell_{T_1^l}: T_1^l
\end{array}
\right] \quad || \quad \ell_{P_2^l}: P_2^l
\end{array}
\right]; \\
\ell_{T_2^l}: T_2^l
\end{array}
\right]
\end{array}
\right]
$$

$$
||
$$

$$
\ell_{G_1^r}: \left[
\begin{array}{l}
\ell_{H_0^r}: H_0^r; \\
\ell_{GP_0^r}: \left[m: \alpha \Rightarrow u \quad || \quad \ell_{P_0^r}: P_0^r \right]; \\
\ell_{T_0^r}: T_0^r
\end{array}
\right]
$$

Observe that $\ell_{G_0^l}$ is actually ℓ.

Next figure shows the locations on the S statement in PADD notation, only some of them are printed out:



This example can be expressed as $S :: [G^l_{k^l} || G^r_{k^r}]$, but it does not match the definition of the top statement form needed in the Theorem 5. Notice that $k^l \neq k^r$. Since locations $\ell_{G^l_3}$ and $\ell_{G^l_1}$ can be identified, both structural orders, respectively as $k^l = 3$ and $k^r = 1$, can be obtained. ⌟

The following procedure computes the structural order of S^x, which is either $S^l(\ell)$ or $S^r(m)$, within a bounded statement S. Basically it searches ancestors ascending from G^x_0, where $x = l$ or r. Each ancestor is assigned to an auxiliary statement, S_G, which initially denotes the basic G^x_0 statement.

Procedure STRUCTORDER − determine the structural order

Input: $S :: [\ell_{LCA} : [S^l(\ell) || S^r(m)]]$, and $[\ell_{G^x_0} : G^x_0]$, where x can be either l or r.
Output: An integer o.

$o := 0$
let $S_G :: G^x_0$

ℓ_1 **while** *least ancestor of* $S_G \neq S$ **do**
ℓ_2 **if** S_G *has immediate parallel statement* **then**
let $S_G ::$ *least ancestor* of S_G
$o = o + 1$
ℓ_3 **if** S_G *has immediate sequence predecessor* **then**
let $S_G ::$ *least ancestor* of S_G
else
ℓ_4 **if** S_G *has immediate sequence successor* **then**
let $S_G ::$ *least ancestor* of S_G
else
ℓ_5 **if** S_G *has immediate sequence predecessor* **then**
let $S_G ::$ *least ancestor* of S_G
$o := o + 1$
else
ℓ_6 **if** S_G *has immediate sequence successor* **then**
let $S_G ::$ *least ancestor* of S_G
$o := o + 1$

Checking for the presence of immediate parallel or sequential statements of S_G and identifying the locations shown before, the order can be calculated. The G_k^x within S are seldom a complete top statement, such as $[H_k^x;\ [G_k^x \mid\mid P_k^x];\ T_k^x]$, they are in one of the following only possible combinations of G_k^x, H_k^x, P_k^x, and T_k^x:

- *form* 1 $[G_k^x \mid\mid P_k^x]$
- *form* 2 $[H_k^x;\ [G_k^x \mid\mid P_k^x]]$
- *form* 3 $[[G_k^x \mid\mid P_k^x];\ T_k^x]$
- *form* 4 $[H_k^x;\ G_k^x]$
- *form* 5 $[G_k^x;\ T_k^x]$

Note that H_k^x and T_k^x match any sequential composition, and P_k^x any cooperation statement. In general the procedure only verifies the relationship between G_k^x and their neighbor statements. The construction of the standard forms G_k^x is done in the next step. The procedure, by considering the above possible forms 1 to 5 at a level, essentially counts the number of nested parallelisms embedding ℓ or m.

At the beginning S_G is G_0^x. When S_G has an immediate parallel statement, the condition at ℓ_2 is true. This case matches the form 1, $[G_k^x \mid\mid P_k^x]$, a G statement has a P statement in parallel. The procedure continues by checking the presence of an immediate sequence predecessor, line ℓ_3, if the condition is true the new S_G matches the form 2, $[H_k^x;\ [G_k^x \mid\mid P_k^x]]$. Similarly for the successors, line ℓ_4, and the form 3, $[[G_k^x \mid\mid P_k^x];\ T_k^x]$. Note that the forms $[G_k^x \mid\mid P_k^x]$, $[H_k^x;\ [G_k^x \mid\mid P_k^x]]$, and $[[G_k^x \mid\mid P_k^x];\ T_k^x]$ has the same structural order, then variable o is incremented once.

When the condition at ℓ_2 is false, S_G does not have an immediate parallel statement, the forms checked at lines ℓ_5 and ℓ_6 match respectively $[H_k^x;\ G_k^x]$ and $[G_k^x;\ T_k^x]$.

The procedure ends when the least ancestor of S_G, line ℓ_1, is S, this means that the procedure has already traversed the whole S, and the variable o contains the number of levels of S.

The general form, $[H_k^x;\ [G_k^x \mid\mid P_k^x];\ T_k^x]$, is not checked directly for the procedure. It has the same order as $[H_k^x;\ [G_k^x \mid\mid P_k^x]]$, checked in lines ℓ_2 and ℓ_3. Checking the presence of T_k^x is skipped since it does not give us further information about the order. In general the number of possible G_k^x forms found establishes the structural order of S^x.

Example The example illustrates how the order o is calculated. The inputs of the procedure are the following: $S ::$

$$\ell_{LCA}: \quad \left[\quad \left[\quad \left[\begin{array}{l} S_0;\ S_1;\ S_2; \\ [G_0^l \ ||\ S_3] \end{array} \right] \ || \ \left[\ S_4 \ \right] \right] \ || \ \left[\ G_0^r \right] \quad \right]$$

and $\ell_{G_0^l} : G_0^l$.

Initially $S_G :: G_0^l$. The *while* condition, line ℓ_1, is satisfied, and the procedure checks for the presence of immediate parallel and sequential statements. Since S_3 is in parallel with G_0^l, checked at line ℓ_2, then the new S_G is $[G_0^l \ || \ S_3]$, which is the least ancestor of G_0^l. The order o is incremented by 1, $o = 1$.

Next, line ℓ_3 checks the presence of immediate sequence predecessor statements. In our example they are $[S_0;\ S_1;\ S_2;]$, thus S_G becomes its least ancestor, now it is the following: $\left[\begin{array}{l} S_0;\ S_1;\ S_2; \\ [G_0^l \ ||\ S_3] \end{array} \right]$. The procedure loops and the while condition, line ℓ_1, is satisfied again. At line ℓ_2, S_4 is the immediate parallel statement, and S_G becomes:

$$\left[\quad \left[\begin{array}{l} S_0;\ S_1;\ S_2; \\ [G_0^l \ ||\ S_3] \end{array} \right] \ || \ \left[\ S_4 \right] \quad \right]$$

and $o = 2$.

The next loop the procedure ends due to the least ancestor of S_G is S.

Computational complexity of STEP1

Basically procedure STEP1 calls twice STRUCTORDER. Since the latter procedure traverses once all levels of S^x, the computational order is $O(n)$, where n is the structural order of S.

$$C_{\text{STEP1}} = O(n)$$

4.2.2 Construct Top Level Statements

The next procedure transforms $S^l(\ell)$ and $S^r(m)$ to the standard forms G_n^l and G_n^r, as in law 19 of page 47, where $n = max(k^l, k^r)$. After the transformation, each G_k^x within the output S' statement of procedure STEP2 has the form $[H_{k-1}^x; [G_{k-1}^x || P_{k-1}^x]; T_{k-1}^x]$. This guarantees that S' is ready to match the communication elimination law for order n of Theorem 4 of chapter 3, to be applied in the STEP3. In the following $N_1, N_2, N_3 \geq 1$.

Procedure STEP2 − put top level statements in the standard form

Input: $S :: [S^l(\ell) \;||\; S^r(m)]$, $p = (\ell, m)$ as in procedure BIN-COMELI, and, k^l, k^r and n from procedure STEP1.

Output: S' a bounded statement equivalent to S or a failure indication.

for $x := l$ *and* r **do**
  $level := 1$
  let S_G denote G_0^x within S
  while $level \leq n$ **do**
    if $level \leq k^x$ **then**
      if S_G *within* S *does not have immediate parallel statement* **then**
        if S_G *within* S *does not have an immediate sequence successor* **then**
t_1 transform $\left\{ [S_G] => \begin{bmatrix} [\ S_G \;||\; \textbf{nil}\ ]; \\ \textbf{nil} \end{bmatrix} \right\}$
        else
t_2 transform $\left\{ \begin{bmatrix} S_G; \\ S_1; \ldots; S_{N_1} \end{bmatrix} => \begin{bmatrix} [\ S_G \;||\; \textbf{nil}\ ]; \\ [S_1; \ldots; S_{N_1}] \end{bmatrix} \right\}$
      else
        if S_G *within* S *does not have an immediate sequence successor* **then**
t_3 transform $\left\{ \begin{array}{l} [S_{P_1}||\cdots||S_{P_i}||S_G||S_{P_{i+2}}||\cdots||S_{P_{N_2}}] => \\ \quad \begin{bmatrix} [\ S_G \;||\; [S_{P_1}||\cdots||S_{P_{N_2}}]\ ]; \\ \textbf{nil} \end{bmatrix} \end{array} \right\}$
        else
t_4 transform $\left\{ \begin{array}{l} \begin{bmatrix} [\ S_{P_1}||\cdots||S_{P_i}||S_G||S_{P_{i+2}}||\cdots||S_{P_{N_2}}\ ]; \\ S_1; \ldots; S_{N_1} \end{bmatrix} => \\ \quad \begin{bmatrix} [\ S_G \;||\; [S_{P_1}||\cdots||S_{P_{N_2}}]\ ]; \\ [S_1; \ldots; S_{N_1}] \end{bmatrix} \end{array} \right\}$
        let S_G denote the resulting form, after the transformation.
        if S_G *within* S *does not have an immediate sequence predecessor* **then**
t_5 transform $\left\{ [S_G] => [\ \textbf{nil};\ S_G] \right\}$
        else
t_6 transform $\left\{ [S_1; \ldots; S_{N_3}; S_G] => [\ [S_1; \ldots; S_{N_3}];\ S_G] \right\}$
      else
t_7 transform $\left\{ [S_G] => \begin{bmatrix} \textbf{nil}; \\ [\ S_G \;||\; \textbf{nil}\ ]; \\ \textbf{nil} \end{bmatrix} \right\}$
      let S_G denote the resulting form, after the transformation.
      $level := level + 1$
  let $S_n^{x'}$ denote S_G, whose structural order is n.
t_8 $S' :: [S_n^{l'}||S_n^{r'}]$

Some parts of the procedure, shown in the previous page, labeled as t_x, are explained in more detail now:

Transformation t_1
In line t_1 the following transformation is applied:

$$\left\{ [S_G] => \left[\begin{array}{l} [\ S_G \ ||\ \mathbf{nil}\ ]; \\ \mathbf{nil} \end{array} \right] \right\}$$

This applies only basic laws showed in section 2.9 of chapter 2. First, the congruence $\{S \approx S \ ||\ \mathbf{nil}\}$ is applied to S_G, then $\{S \approx S; \mathbf{nil}\}$ is applied to the resulting form. The transformation guarantees that the output form is a $[[G||P];T]$ form, where $P = T = \mathbf{nil}$.

Transformation t_2
The transformation applied in line t_2, with $N_1 \geq 1$, is:

$$\left\{ \begin{bmatrix} S_G; \\ S_1; \ldots ; S_{N_1} \end{bmatrix} => \left[\begin{array}{l} [\ S_G \ ||\ \mathbf{nil}\ ]; \\ [S_1; \ldots ; S_{N_1}] \end{array} \right] \right\}$$

The first law is the same as above, then sequence associativity is applied, to $S_1; \cdots ; S_N$, obtaining a $[[G||P];T]$ form, where P is **nil**, and T is $[S_1; \cdots ; S_{N_1}]$.

Transformation t_4
This transformation:

$$\left\{ \begin{bmatrix} [\ S_{P_1}|| \cdots ||S_{P_i}||S_G||S_{P_{i+2}}|| \cdots ||S_{P_{N_2}}\ ]; \\ S_1; \ldots ; S_{N_1} \end{bmatrix} => \begin{bmatrix} [\ S_G \ ||\ [S_{P_1}|| \cdots ||S_{P_{N_2}}]\ ]; \\ [S_1; \ldots ; S_{N_1}] \end{bmatrix} \right\}$$

applies sequentially the following:

- Parallelism Permutation, due to commutativity:

$$\left[S_{P_1}|| \cdots ||S_{P_i}||S_G||S_{P_{i+2}}|| \cdots ||S_{P_{N_2}}\right] => \left[S_G||S_{P_1}|| \cdots ||S_{P_i}||S_{P_{i+2}}|| \cdots ||S_{P_{N_2}}\right]$$

  The permutation procedure is detailed in section 5.4.2.2 of chapter 5. S_G is permuted to the beginning of the cooperation statement.

- Cooperation Associativity (from section 5.4.2.3):

$$\left[S_G||S_{P_1}|| \cdots ||S_{P_i}||S_{P_{i+2}}|| \cdots ||S_{P_{N_2}}\right] => \left[S_G \ ||\ [S_{P_1}|| \cdots ||S_{P_{N_2}}]\right]$$

- Concatenation Associativity (from section 5.4.2.7):

$$\begin{bmatrix} S; \\ S_1; \ldots; S_{N_3} \end{bmatrix} => \begin{bmatrix} S; \\ [S_1; \ldots; S_{N_3}] \end{bmatrix}$$

Computational complexity of STEP2

For each level of S, two transformations are applied at most. The complexity of them could be, for the worst case, $O(N_1 + N_3)$ or $O(N_2 + N_3)$. The number st of statements could be taken as an upper bound for the N_i's. Hence

$$C_{\text{STEP2}} = O(n \times st)$$

where n is the structural order of S.

4.2.3 Application of Elimination from a Binary Cooperation

The proper communication elimination laws given in section 3.2.2 of chapter 3, are applied in this step of the procedure to the program resulting from last step.

As an example the following shows the communication elimination law of order 2.

$$\begin{bmatrix} H_1^l; \\ \begin{bmatrix} \begin{bmatrix} H_0^l; \\ [\ \alpha \Leftarrow e \ \ || \ \ P_0^l\ ]; \\ T_0^l \end{bmatrix} || \ P_1^l \end{bmatrix}; \\ T_1^l \end{bmatrix} \ || \ \begin{bmatrix} H_1^r; \\ \begin{bmatrix} \begin{bmatrix} H_0^r; \\ [\ \alpha \Rightarrow u \ \ || \ \ P_0^r\ ]; \\ T_0^r \end{bmatrix} || \ P_1^r \end{bmatrix}; \\ T_1^r \end{bmatrix}$$

$$=_{\mathcal{O}} \begin{bmatrix} [\ H_1^l \ || \ H_1^r\ ]; \\ \begin{bmatrix} \begin{bmatrix} [\ H_0^l \ || \ H_0^r\ ]; \\ [\ u := e \ || \ P_0^l \ || \ P_0^r\ ]; \\ [\ T_0^l \ || \ T_0^r\ ] \end{bmatrix} || \ P_1^l \ || \ P_1^r \end{bmatrix}; \\ [\ T_1^l \ || \ T_1^r\ ] \end{bmatrix}$$

From chapter 3 the applicability conditions have to be checked before application of the law. There where two types of them: checking that two substatements are communicating, and checking the order of communications of two statements with respect to a third one. The next procedures carry out those checks now:

❑ ***Procedure CommStat***

Procedure COMMSTAT verifies whether or not two statements communicate. They communicate if a matching communication pair is found within them and are parallel. The next procedure checks that one of the two statements has an output and the other an input over the same channel, and vice versa. They communicate if the intersection of the input and output communication statement sets, InpSt and OutSt respectively (detailed in section 5.3.3.2), is not empty, as the following shows:

Procedure COMMSTAT − communicating statements

Input: A, B are statements in parallel.
Output: A boolean b. b is $true$ if A and B communicate.

$b := (\mathsf{InpSt}(A) \cap \mathsf{OutSt}(B) \neq \emptyset) \bigvee (\mathsf{OutSt}(A) \cap \mathsf{InpSt}(B) \neq \emptyset)$

❑ ***Procedure CommPrecede***

It computes the expressions $cw(A) < cw(B)$ within P, introduced in definition 9 of chapter 3. The procedure is the following:

Procedure COMMPRECEDE − communication order precedence restriction

Input: P, A, B statements from any G statement.
Output: A boolean b. b is $true$ if $cw(A) < cw(B)$ within P.

```
b := true
if {COMMSTAT(P, A) ∧ COMMSTAT(P, B)} then
    S_PA :: set of matching pairs found in (P, A)
    S_PB :: set of matching pairs found in (P, B)
    foreach element x of S_PB do
        if x precedes within P any communication pair of S_PA then
            b := false
p1          exit
```

The procedure exits with failure, location p_1, if any communication of (P, B) precedes a communication of (P, A). The precedence between the above sets is determined by checking within P, statement by statement and in its concatenation order, that any element of S_{PB} does not come before any element of S_{PA}, otherwise the procedure ends and the boolean variable b becomes $false$.

The overall procedure STEP3 checks the applicability conditions first. Only when all are satisfied the elimination law is applied. Notice that, due to the current implementation the index $(n+1)$ of the applicability conditions of Theorems 1 and 2, pages 50 and 51 respectively, has been changed to (n) in the procedure which is the following:

Procedure STEP3 − proper communication elimination

Input: $S :: [S_n^l(\ell) \mid\mid S_n^r(m)]$ the output of procedure STEP2, and n.

Output: S' a bounded statement equivalent to S, where p has been eliminated, or a failure indication.

if ℓ *is an input communication statement* **then**

e_1 **let** S' be the result of the transformation

$$\left\{ \quad \left[S_n^l(\ell) \mid\mid S_n^r(m)\right] \quad => \quad \left[S_n^r(m) \mid\mid S_n^l(\ell)\right] \right\}$$

else let $S' :: S$

let the P's, T's and G's are the substatement of S' in the rest.

e_2 **if** $\{\bigvee$COMMSTAT(P_i^l, T_k^r), *for* $k \in [0, n-1]$ *and* $i \in [0, k]$ $\}$ **then**
exit with failure, some (P_i^l, T_k^r) communicate

e_3 **if** $\{\bigvee$COMMSTAT(P_i^r, T_k^l), *for* $k \in [0, n-1]$ *and* $i \in [0, k]$ $\}$ **then**
exit with failure, some (P_i^r, T_k^l) communicate

e_4 **if** $\{\bigvee$COMMSTAT(T_i^l, T_j^r), *for* $i, j \in [0, n-1]$, $i \neq j$ $\}$ **then**
exit with failure, some (T_i^l, T_j^r) communicate

e_5 **if** $\{\bigvee$ [$\neg$COMMPRECEDE$(P_k^l, P_i^l, T_i^r) \vee \neg$COMMPRECEDE$(P_k^l, P_i^r, T_i^l) \vee$
$\neg$COMMPRECEDE$(P_k^l, G_i^l, T_i^r) \vee \neg$COMMPRECEDE$(P_k^l, G_i^r, T_i^l)$],
for $k \in [1, n-1]$ *and* $i \in [0, k-1]$ $\}$ **then**
exit with failure, communication order restriction for P_k^l not satisfied

e_6 **if** $\{\bigvee$ [$\neg$COMMPRECEDE$(P_k^r, P_i^l, T_i^r) \vee \neg$COMMPRECEDE$(P_k^r, P_i^r, T_i^l) \vee$
$\neg$COMMPRECEDE$(P_k^r, G_i^l, T_i^r) \vee \neg$COMMPRECEDE$(P_k^r, G_i^r, T_i^l)$],
for $k \in [1, n-1]$ *and* $i \in [0, k-1]$ $\}$ **then**
exit with failure, communication order restriction for P_k^r not satisfied

e_7 $(failure, S')$ :: APPLY$(S', bincomeli_n, LftToRght)$

The procedure applies first a basic transformation needed for the matching of the communication elimination law. Next the applicability conditions are checked, if they hold the communication elimination can take place, otherwise the procedure exits with failure. The parts of the procedure, labeled as e_x, are detailed next.

Transformation e_1
In case the communication statement ℓ be an input communication statement, then the binary cooperation commutativity law is applied, moving the output communication statement to the left. This is necessary due to the definition of Theorem 5, where G_0^l and G_0^r must contain $\alpha \Leftarrow e$ and $\alpha \Rightarrow u$ respectively. Now S' matches perfectly the communication law for order n.

Applicability conditions e_2 **and** e_3
The statement pairs (P^l, T^r) and (P^r, T^l) should not communicate. First the pair (P^l, T^r) is checked. The next schema shows how these statements are in parallel in the left hand side of the interface equivalence but after applying the reduction they are in sequence. Since they do not communicate, no deadlock can be introduced in the r.h.s.

$$\left[\begin{array}{c} H^l; \\ \left[\ G^l \ \|\ P^l\ \right]; \\ T^l \end{array}\right] \| \left[\begin{array}{c} H^r; \\ \left[\ G^r \ \|\ P^r\ \right]; \\ T^r \end{array}\right] =_{\mathcal{O}} \left[\begin{array}{c} \left[\ H^l \ \|\ H^r\ \right]; \\ \left[\ G \ \|\ P^l \ \|\ P^r\ \right]; \\ \left[\ T^l \ \|\ T^r\ \right] \end{array}\right]$$

For example consider the following schema. It represents the r.h.s of the communication law for order 2. P_1^l and T_0^r statements can communicate since they are in parallel before and after the reduction.

$$\left[\begin{array}{c} \left[\ H_1^l \ \|\ H_1^r\ \right]; \\ \left[\begin{array}{c} \left[\begin{array}{c} \left[\ H_0^l \ \|\ H_0^r\ \right]; \\ \left[\ G_0 \ \|\ P_0^l \ \|\ P_0^r\ \right]; \\ \left[\ T_0^l \ \|\ T_0^r\ \right] \end{array}\right] \| \left[\ P_1^l \ \|\ P_1^r\ \right] \end{array}\right]; \\ \left[\ T_1^l \ \|\ T_1^r\ \right] \end{array}\right]$$

Theorem 1 of page 50 expresses this condition as:

- The pairs (P_i^l, T_k^r) do not communicate, for $k \in [0, n-1]$ and $i \in [0, k]$

The procedure calls COMMSTAT to verify whether or not P^l and T^r communicate. The two statements communicate if a matching communication pair is found within them. COMMSTAT has been detailed earlier in this section.

Example Given the communication elimination law of order 3, the procedure checks that the following pairs do not communicate: (P_0^l, T_0^r), (P_0^l, T_1^r), (P_1^l, T_1^r), (P_0^l, T_2^r), (P_1^l, T_2^r), and (P_2^l, T_2^r). If any conditions do not hold then the procedure stops and exits reporting a failure. ▟

Similarly for the pairs (P^r, T^l), where from Theorem 1:

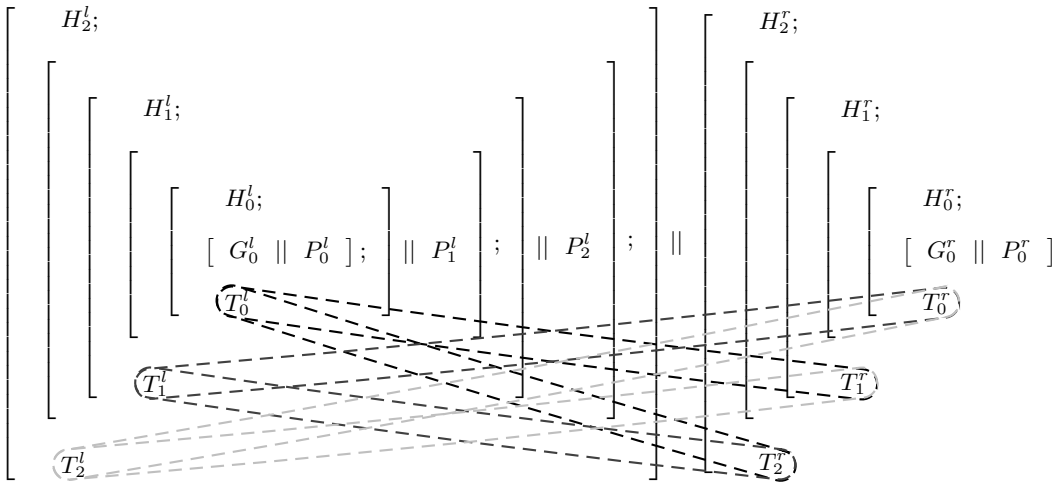
- The pairs (P_i^r, T_k^l) do not communicate, for $k \in [0, n-1]$ and $i \in [0, k]$

Applicability conditions e_4
The procedure checks the following expression, from Theorem 1:

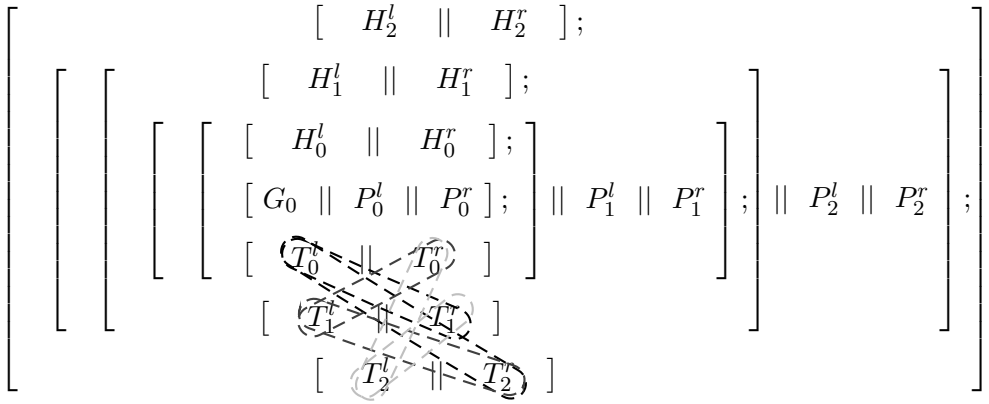
- The pairs (T_i^l, T_j^r) do not communicate, for $i, j \in [0, n-1]$, $i \neq j$

Note that statements with index $i = j$ can communicate since they remain in parallel after applying the communication elimination law.

Example The next figures show the elimination law of order 3, and how the pairs: (T_0^l, T_1^r), (T_0^l, T_2^r), (T_1^l, T_0^r), (T_1^l, T_2^r), (T_2^l, T_0^r), and (T_2^l, T_1^r), are in parallel in the l.h.s.:

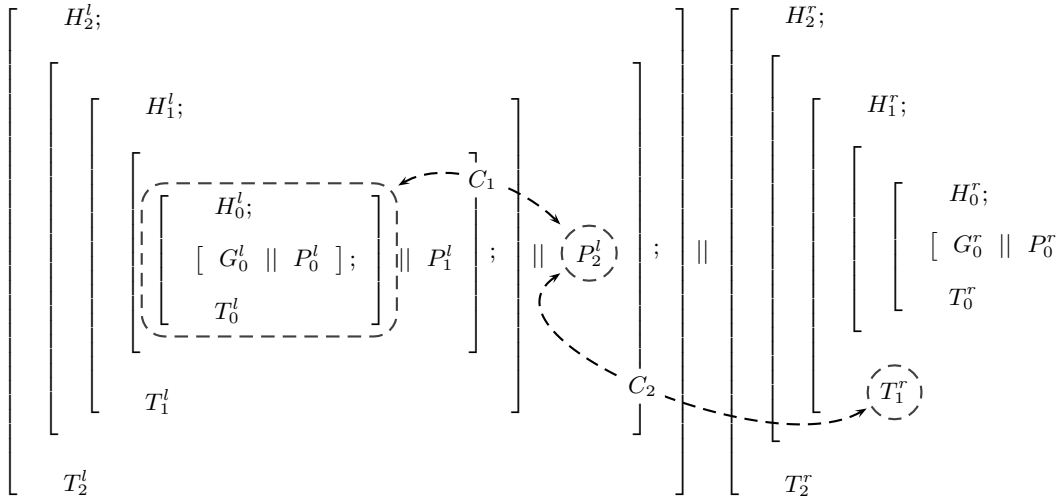


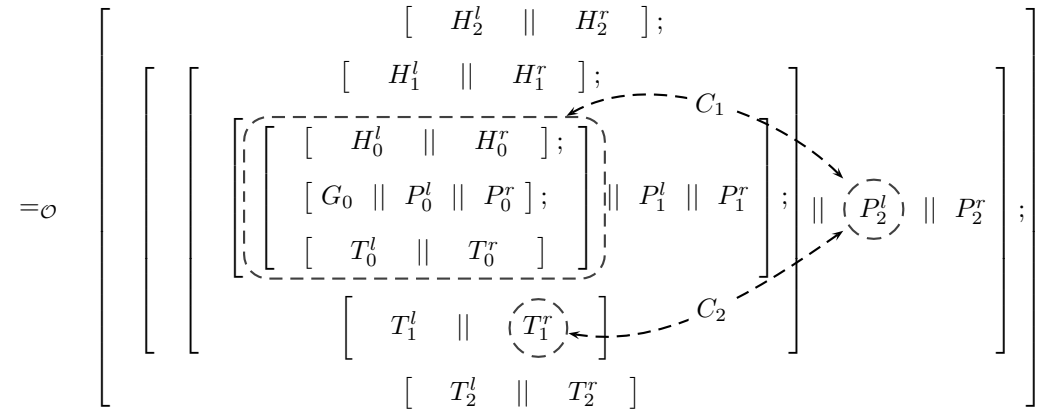
and in sequence in r.h.s.:



Communication order restriction e_5 **and** e_6
These restrictions are also conditions to be fulfilled before applying the elimination law, these are explained in Theorem 2 of page 51. The procedure checks the communications precedence for each P^l and P^r.

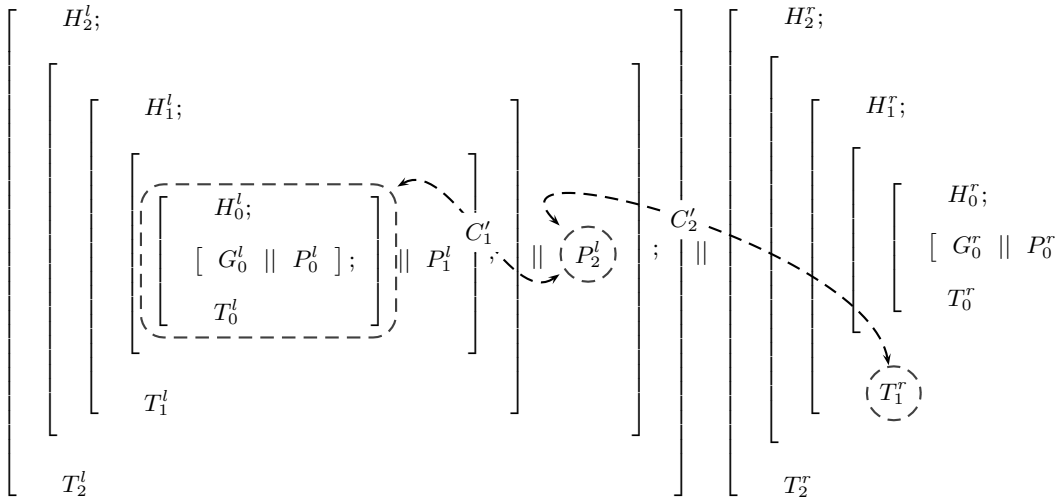
Example Given the law of order 3, observe the communications within P_2^l with G_1^l and T_1^r in both sides of the equivalence:

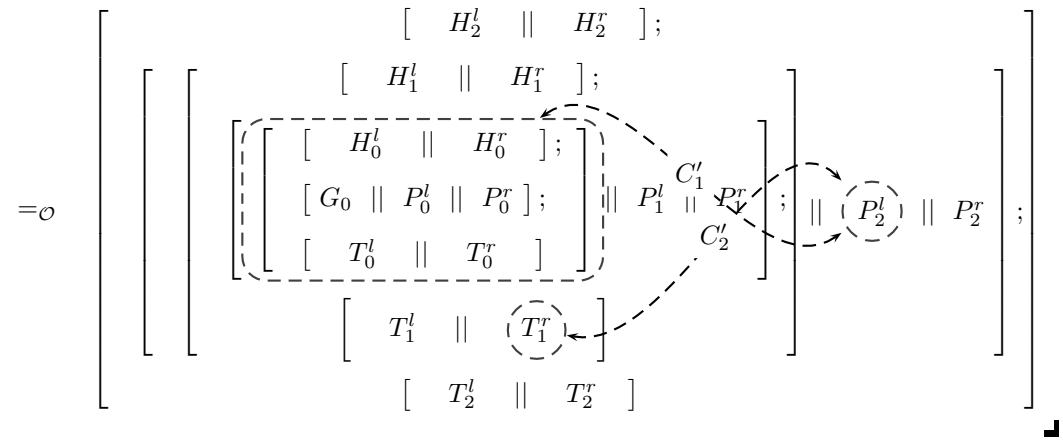




Labels C_1 and C_2 denote the communication statements within P_2^l with G_1^l and T_1^r respectively. In the example the communication C_1 always precedes C_2 either in the l.h.s. or r.h.s, thus no deadlock can be introduced after applying the elimination reduction.

Rewriting the example changing the order of the communications within P_2^l, the next schema is obtained. Now in the l.h.s. C_2' precedes C_1'. Statements G_1^l and T_1^r are in parallel with P_2^l and both communication events can take place. In the r.h.s. G_1^l and T_1^r are not in parallel, therefore if C_2' precedes C_1' then a deadlock would be introduced in case the elimination transformation took place. Since the restrictions are not satisfied the communication elimination reduction can not be applied.





In general, the set of communication order restrictions to be fulfilled are:

- Within P_k^l for $k \in [1, n-1]$ and $i \in [0, k-1]$,

$$cw(P_i^l) < cw(T_i^r) \ \bigwedge \ cw(P_i^r) < cw(T_i^l) \ \bigwedge \ cw(G_i^l) < cw(T_i^r) \ \bigwedge \ cw(G_i^r) < cw(T_i^l)$$

- Also for P_k^r for $k \in [1, n-1]$

The expression $cw(A) < cw(B)$ is explained in definition 9 of page 49, and it is computed by procedure COMMPRECEDE introduced in the beginning of this section.

Transformation e_7
Finally the proper communication elimination law for n can be applied, whose mnemonic is $bincomeli_n$, to S' by calling procedure APPLY, explained in the next chapter 5:

$$(failure, S') :: \text{APPLY}(S', \ bincomeli_n, \ LftToRght)$$

where law $bincomeli_n$ is applied from l.h.s. to r.h.s., as parameter $LftToRght$ indicates.

When the procedure terminates without failure, indicated by boolean $failure$, a bounded communication statement S', interface equivalent to S, is obtained, and the matching communication pair (ℓ, m) has been eliminated.

Computational complexity of STEP3

Procedure STEP3 computes the applicability conditions by calling procedures COMMSTAT and COMMPRECEDE, traversing all n levels of S $n-1$ times.

$$C_{\text{STEP3}} = O(n^2)$$

where n is the structural order of S.

4.2.4 Remove Nil Statements

This step rearranges the statement S', obtained in STEP3, by eliminating the redundant *nil* statements introduced before applying the communication elimination law. S' has the structure of the r.h.s. of the proper communication elimination law given in the subsection 3.2.2.

In previous steps the *nil*'s are used to construct the G statements, for matching with the l.h.s. of the communication elimination law. Now they are no longer needed. The procedure applies the simple laws to reduce them.

Procedure STEP4, shown in the next page, operates at each level of the S_n statement, starting at the inner most one. First it removes *nil*'s from statements $[G_k||P_k^l||P_x^r]$. Then, it eliminates the *nil*'s from the tail statements $[T_k^l||T_k^r]$, and finally simplifies the heading part $[H_k^l||H_k^r]$. Most of the laws used in this procedure are the ones applied in the STEP2 and detailed in subsection 2.9.2 of chapter 2. Here, they are all applied from right to left.

Some steps of procedure STEP4, label as n_x, are the following:

Transformation n_1
In this step the following transformation is applied:

$$\left\{ \quad [\ G_k\ ||\ \mathbf{nil}\ ||\ \mathbf{nil}\ ] \Rightarrow [\ G_k\ ] \right\}$$

It corresponds to double application of law 5 of page 39 of chapter 2.

Transformation n_4
This transformation is derived in two steps as follows:

$$\left\{ \quad \left[\begin{array}{c} [\ S_{GP}\ ]; \\ [\ \mathbf{nil}\ ||\ \mathbf{nil}\ ] \end{array} \right] \Rightarrow \left[\begin{array}{c} S_{GP}; \\ \mathbf{nil} \end{array} \right] \Rightarrow [\ S_{GP}\ ] \right\}$$

$\{S \approx S\ ||\ \mathbf{nil}\}$ .....

$\{S \approx S; \mathbf{nil}\}$ .....

The steps use Law 5 and Law 1 respectively.

Transformation n_8
Applying $\{S \approx \mathbf{nil}\ ||\ S\ \}$, obtaining:

$$\left\{ \quad \left[\begin{array}{c} [\ \mathbf{nil}\ ||\ H_k^r\ ]; \\ [\ S_{GP}\ ] \end{array} \right] \Rightarrow [\ H_k^l;\ S_{GPT}\ ] \right\}$$

Procedure STEP4 − elimination of redundant *nil* statements

Input: $S :: [S_n]$ the output of procedure STEP3, and n from STEP1.

Output: S' a bounded statement equivalent to S.

for $k := 0$ ***to*** n **do**

  if *The two immediate parallel statements of* G_k *are* ***nil*** **then**

n_1 apply $\left\{ \quad [G_k \parallel \textbf{nil} \parallel \textbf{nil}\,] \Rightarrow [G_k\,] \right\}$

  else

    if *statement* P_k^l *is* ***nil*** **then**

n_2 apply $\left\{ \quad [G_k \parallel \textbf{nil} \parallel P_k^r\,] \Rightarrow [G_k \parallel P_k^r\,] \right\}$

    else

      if *statement* P_k^r *is* ***nil*** **then**

n_3 apply $\left\{ \quad [G_k \parallel P_k^l \parallel \textbf{nil}\,] \Rightarrow [G_k \parallel P_k^l\,] \right\}$

  let S_{GP} denote the resulting form, after the transformation.

  if *The immediate sequence successor of* S_{GP} *is a* [***nil*** || ***nil***] *statement* **then**

n_4 apply $\left\{ \quad \left[\begin{array}{c}[\,S_{GP}\,];\\ [\,\textbf{nil} \parallel \textbf{nil}\,]\end{array}\right] \Rightarrow [S_{GP}\,] \right\}$

  else

    if *statement* T_k^l *is* ***nil*** **then**

n_5 apply $\left\{ \quad \left[\begin{array}{c}[\,S_{GP}\,];\\ [\,\textbf{nil} \parallel T_k^r\,]\end{array}\right] \Rightarrow [S_{GP};\ T_k^r\,] \right\}$

    else

      if *statement* T_k^r *is* ***nil*** **then**

n_6 apply $\left\{ \quad \left[\begin{array}{c}[\,S_{GP}\,];\\ [\,T_k^l \parallel \textbf{nil}\,]\end{array}\right] \Rightarrow [S_{GP};\ T_k^l\,] \right\}$

  let S_{GPT} denote the resulting form, after the transformation.

  if *The immediate sequence predecessor of* S_{GPT} *is a* [***nil*** || ***nil***] *statement* **then**

n_7 apply $\left\{ \quad \left[\begin{array}{c}[\,\textbf{nil} \parallel \textbf{nil}\,];\\ [\,S_{GPT}\,]\end{array}\right] \Rightarrow [S_{GPT}\,] \right\}$

  else

    if *statement* H_k^l *is* ***nil*** **then**

n_8 apply $\left\{ \quad \left[\begin{array}{c}[\,\textbf{nil} \parallel H_k^r\,];\\ [\,S_{GPT}\,]\end{array}\right] \Rightarrow [H_k^r;\ S_{GPT}\,] \right\}$

    else

      if *statement* H_k^r *is* ***nil*** **then**

n_9 apply $\left\{ \quad \left[\begin{array}{c}[\,H_k^l \parallel \textbf{nil}\,];\\ [\,S_{GPT}\,]\end{array}\right] \Rightarrow [H_k^l;\ S_{GPT}\,] \right\}$

S' :: the resulting form, after removing all ***nil*** statements.

Computational complexity of STEP4

For each level of S, three simple transformations are applied at most. Then,

$$C_{\text{STEP4}} = O(n)$$

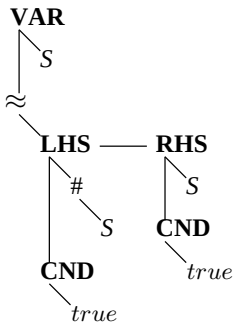
where n is the structural order of S.

4.2.5 Remove Sequence and Parallelism Associations

The output of STEP4 still contains some redundant statement associations. These have been introduced in procedure STEP2 to construct the appropriate G statements.

Procedure STEP5 is shown in the next page. First, line a_1, it reduces the association of a sequence of contiguous substatements into non-associated substatements. This is done applying procedure CONCATFLAT of subsection 5.4.2.8 of next chapter 5. The sequence association has been introduced in lines t_2 and t_5 of procedure STEP2. This was necessary to arrange the H's and T's for matching with the communication elimination law.

In PADD notation, substatements can be associated under the scope of a comment (a line starting with a '#'), called scoped descriptions and introduced in chapter 2. The sequential association removal law applied by CONCATFLAT, can be expressed in tree-like PADD notation as follows:



The law notation is introduced in section 5.3 of chapter 5. Here is shown only as an example.

The procedure traverses in preorder through the internal representation, remember that the PADD notation is a ternary tree representation, searching for sequential associations and applying the concatenation flattening transformation, which removes it, and obtaining a new bounded statement S' equivalent to S.

The procedure continues by reducing the parallelism associations, line a_2. These also have been introduced at line t_3 of STEP2. Applying procedure COOPFLAT, detailed in subsection 5.4.2.5 of chapter 5, they are reduced. Usually these associations are located within $[G_k \; ||P_k^l \; || \; P_k^r]$ statements as a result of applying the communication elimination law.

Procedure STEP5 − elimination of redundant statement associations

Input: S the output of procedure STEP4.
Output: S' a bounded statement equivalent to S.

let $S' :: S$

foreach *sequence association,* S_S*, found in preorder within* S' **do**
a_1 $\quad S' :: \text{CONCATFLAT}(S_S, S')$

foreach *parallelism association,* S_P*, found in preorder within* S' **do**
a_2 $\quad S' :: \text{COOPFLAT}(S_P, S')$

Observe that the substatements within S_P, line a_2, can be associated as cooperation substatements. For instance, $S_P :: \; [S_1 \; || \cdots || \; S_m]$ where $S_1 :: [S_1' \; || \cdots || \; S_m']$. At line a_2 COOPFLAT visits all inner cooperation substatements of S_P, and returns an S' equivalent to S, where the parallelisms have been flattened.

Computational complexity of STEP5

Since S is the output of the communications elimination transformation, it contains at most two sequence associations and at most one parallelism association per level, with $O(st)$ complexity each, as pages 135 and 130 of chapter 5 show. Then,

$$C_{\text{STEP5}} = O(n \times st)$$

where n is the structural order of S.

4.2.6 Overall Computational Complexity

The next table 4.1 summarizes the computational complexity of each step of procedure BIN-COMELI.

Since binary communication elimination procedure, BIN-COMELI, is a composition of all the above steps, the higher degree of them determines its complexity. Then, $C_{\text{BIN-COMELI}} = O(n^2 + n \times st)$

Procedure	Complexity
STEP1 – determine the orders of $S^l(\ell)$ and $S^r(m)$	$O(n)$
STEP2 – put top level statements in the standard form	$O(n \times st)$
STEP3 – proper communication elimination	$O(n^2)$
STEP4 – elimination of redundant *nil* statements	$O(n)$
STEP5 – elimination of redundant statement associations	$O(n \times st)$

Table 4.1: Computation complexity of BIN-COMELI

4.3 A Communication Elimination Example

This section describes in detail how the communication elimination reduction is applied to a specific statement S. All the intermediate forms of each step of the elimination are shown, these are the ones obtained after applying either a law or a transformation.

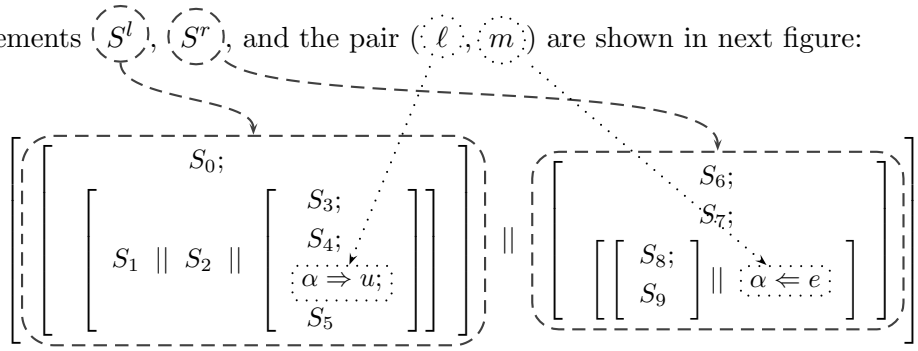
Given the following bounded statement S, procedure BIN-COMELI is applied to eliminate the matching pair (ℓ, m):

$$S :: \left[\left[\left[\begin{array}{c} S_0; \\ \left[\begin{array}{ccc} S_1 & || & S_2 \end{array} || \left[\begin{array}{c} S_3; \\ S_4; \\ \alpha \Rightarrow u; \\ S_5 \end{array}\right]\right] \end{array}\right]\right] || \left[\begin{array}{c} S_6; \\ S_7; \\ \left[\left[\begin{array}{c} S_8; \\ S_9 \end{array}\right] || \ \alpha \Leftarrow e\right] \end{array}\right]\right]$$

The inputs of procedure BIN-COMELI, detailed in section 4.2, are:

- A statement S, which denotes a binary parallel composition $[S^l(\ell)||S^r(m)]$
- A matching communication pair $p = (\ell, m)$

Substatements S^l, S^r, and the pair (ℓ, m) are shown in next figure:



❑ STEP1 - Determine the orders.

The procedure computes the structural order of statements S^l and S^r by calling twice procedure STRUCTORDER of page 70. Its execution trace is the following:

- S^l: Initially $S_G :: G_0^l$, where $G_0^l :: \alpha \Rightarrow u$. The *while* condition is satisfied, line ℓ_1 of the procedure, since the ancestor of S_G, $\left[\begin{array}{c} S_3; \\ S_4; \\ \alpha \Rightarrow u; \\ S_5 \end{array}\right]$ is different from S.

  Next, the procedure checks the presence of immediate statements of S_G. As it has only sequence predecessors and successors, ℓ_2 is skipped. At ℓ_5, S_3 and S_4 are the sequence predecessor statements, then variable $o = 1$ and S_G becomes its least ancestor:

  $$* \quad S_G :: \left[\begin{array}{c} S_3; \\ S_4; \\ \alpha \Rightarrow u; \\ S_5 \end{array}\right]$$

  In the next iteration S_G has immediate parallel statements, $[S_1||S_2]$, checked at ℓ_2, then the structural order is increased, $o = 2$, and:

  $$* \quad S_G :: \left[\begin{array}{c} S_1 \;||\; S_2 \;||\; \left[\begin{array}{c} S_3; \\ S_4; \\ \alpha \Rightarrow u; \\ S_5 \end{array}\right] \end{array}\right]$$

  Next, line ℓ_3, S_0 is the immediate sequence predecessor and the new S_G is:

  $$* \quad S_G :: \left[\begin{array}{c} S_0; \\ \left[\begin{array}{c} S_1 \;||\; S_2 \;||\; \left[\begin{array}{c} S_3; \\ S_4; \\ \alpha \Rightarrow u; \\ S_5 \end{array}\right] \end{array}\right] \end{array}\right]$$

  Variable o does not change, and the procedure ends due to the least ancestor of $S_G = S$.

- S^r: Initially $S_G :: G_0^r$, where $G_0^r :: \alpha \Leftarrow e$. S_G has immediate parallel statements, $[S_8; S_9]$, ℓ_2, $o = 1$ and:

  $$* \quad S_G :: \left[\left[\begin{array}{c} S_8; \\ S_9 \end{array} \right] \,||\, \alpha \Leftarrow e \right]$$

  In the last step, ℓ_3:

  $$* \quad S_G :: \left[\begin{array}{c} S_6; \\ S_7; \\ \left[\left[\begin{array}{c} S_8; \\ S_9 \end{array} \right] \,||\, \alpha \Leftarrow e \right] \end{array} \right]$$

  o remains unchanged. The procedure exits and $o = 1$.

Since $k^l = 2$, and $k^r = 1$, then $n = 2$ and the matching communication elimination law is of order 2.

❑ STEP2 - Construct top level statements.

After collecting the information about S, the construction of statements G, procedure of page 73, takes place as follows:

- Initially S_G denotes $G_0^l :: [\, \alpha \Rightarrow u \,]$.
  - Arranging S^l for $level = 1$.
    - S_G has not parallel statements, but has immediate sequence successors, then the transformation t_2 is applied:

      $$\left\{ \begin{bmatrix} S_G; \\ S_1; \ldots; S_N \end{bmatrix} => \begin{bmatrix} [\, S_G \,||\, \mathbf{nil} \,]; \\ [S_1; \ldots; S_N] \end{bmatrix} \right\}$$

      obtaining:

      $$\left\{ \left[\begin{array}{c} \alpha \Rightarrow u; \\ S_5 \end{array} \right] => \left[\begin{array}{l} [\, \alpha \Rightarrow u \,||\, \mathbf{nil} \,]; \\ S_5 \end{array} \right] \right\}$$

      The concatenation $[S_1; \ldots; S_N]$ matches S_5, where $N = 1$. In this case, the sequence associativity is not applied.

* The resulting form is: $S_G :: \left[\begin{array}{l} [\ \alpha \Rightarrow u \ ||\ \mathbf{nil}\ ]; \\ S_5 \end{array}\right]$

* Next step applies the transformation t_6:

$$\left\{\ \ [S_1; \ldots ; S_N; S_G] => [\ [S_1; \ldots ; S_N];\ S_G]\ \right\}$$

to S_G, obtaining:

$$\left\{\ \ \left[\begin{array}{l} S_3; \\ S_4; \\ [\ \alpha \Rightarrow u \ ||\ \mathbf{nil}\ ]; \\ S_5 \end{array}\right] => \left[\begin{array}{c} \left[\begin{array}{l} S_3; \\ S_4 \end{array}\right]; \\ [\ \alpha \Rightarrow u \ ||\ \mathbf{nil}\ ]; \\ S_5 \end{array}\right]\ \right\}$$

The concatenation $[S_1; \ldots ; S_N]$ matches $[S_3; S_4]$, where $N = 2$, and the sequence associativity is applied.

* Now S_G is $\left[\begin{array}{c} \left[\begin{array}{l} S_3; \\ S_4 \end{array}\right]; \\ [\ \alpha \Rightarrow u \ ||\ \mathbf{nil}\ ]; \\ S_5 \end{array}\right]$ which corresponds to G_1^l.

* *level* is incremented by 1, $level = 2$.

◦ Arranging S^l for level 2.

* Transformation t_3:

$$\left\{\ \ [S_{P_1}||\cdots||S_{P_i}||S_G||S_{P_{i+2}}||\cdots||S_{P_N}] => \left[\begin{array}{l} [\ S_G\ ||\ [S_{P_1}||\cdots||S_{P_N}]\ ]; \\ \mathbf{nil} \end{array}\right]\ \right\}$$

is applied obtaining:

$$\left\{\ \ [\ S_1\ ||\ S_2\ ||\ S_G\ ] => \left[\begin{array}{l} [\ S_G\ ||\ [\ S_1\ ||\ S_2\ ]\ ]; \\ \mathbf{nil} \end{array}\right]\ \right\}$$

* Next, transformation t_6 is applied. Note that concatenation $[S_1; \ldots ; S_N]$ matches S_0, where $N = 1$, hence sequence associativity is not applied.

* The resulting form, which corresponds to G_2^l, is:

$$S_G :: \left[\begin{array}{c} S_0; \\ \left[\begin{array}{c} \left[\begin{array}{c} \left[\begin{array}{c} S_3; \\ S_4 \end{array}\right]; \\ [\ \alpha \Rightarrow u \ ||\ \mathbf{nil}\ ]; \end{array}\right] \ ||\ [\ \ S_1\ \ ||\ \ S_2\ ] \\ S_5 \end{array}\right]; \\ \mathbf{nil} \end{array}\right]$$

* *level* is incremented by 1, $level = 3$, exiting from the **while** loop.

- At this point $S_n^{l'}$ is a standard top statement of order 2.

- Next iteration constructs $S_n^{r'}$. Initially S_G denotes $G_0^r :: [\ \alpha \Leftarrow e\ ]$.

  - Arranging S^r for $level = 1$:

    * Transformation t_3 is applied:

$$\left\{ \begin{array}{c} [S_{P_1}||\cdots||S_{P_i}||S_G||S_{P_{i+2}}||\cdots||S_{P_N}] => \left[\begin{array}{l} [\ S_G\ ||\ [S_{P_1}||\cdots||S_{P_N}]\ ]; \\ \mathbf{nil} \end{array}\right] \end{array} \right\}$$

      to $\left[\begin{array}{l} \left[\begin{array}{l} S_8; \\ S_9 \end{array}\right]; \ S_G \end{array}\right]$, obtaining:

$$\left\{ \begin{array}{c} \left[\begin{array}{c} \left[\begin{array}{l} S_8; \\ S_9 \end{array}\right] \ ||\ \alpha \Leftarrow e \end{array}\right] => \left[\begin{array}{c} \left[\begin{array}{c} \alpha \Leftarrow e\ \ ||\ \left[\begin{array}{l} S_8; \\ S_9 \end{array}\right] \end{array}\right]; \\ \mathbf{nil} \end{array}\right] \end{array} \right\}$$

    * S_G becomes $\left[\begin{array}{c} \left[\begin{array}{c} \alpha \Leftarrow e\ \ ||\ \left[\begin{array}{l} S_8; \\ S_9 \end{array}\right] \end{array}\right]; \\ \mathbf{nil} \end{array}\right]$

    * Applying transformation t_6, sequence association, to S_6 and S_7:

$$\left\{ \begin{bmatrix} S_6; \\ S_7; \\ \left[\ \alpha \Leftarrow e \ \|\ \left[\ \begin{matrix} S_8; \\ S_9 \end{matrix}\ \right]\right]; \\ \textbf{nil} \end{bmatrix} => \begin{bmatrix} \begin{bmatrix} S_6; \\ S_7 \end{bmatrix}; \\ \left[\ \alpha \Leftarrow e \ \|\ \left[\ \begin{matrix} S_8; \\ S_9 \end{matrix}\ \right]\right]; \\ \textbf{nil} \end{bmatrix} \right\}$$

* The resulting form is $S_G :: \begin{bmatrix} \begin{bmatrix} S_6; \\ S_7 \end{bmatrix}; \\ \left[\ \alpha \Leftarrow e \ \|\ \left[\ \begin{matrix} S_8; \\ S_9 \end{matrix}\ \right]\right]; \\ \textbf{nil} \end{bmatrix}$ which corresponds to G_1^r.

* *level* is incremented by 1, $level = 2$.

○ Arranging S^r for level 2.

* Transformation t_7:

$$\left\{ [S_G] => \begin{bmatrix} \textbf{nil}; \\ [\ S_G \ \|\ \textbf{nil}\ ]; \\ \textbf{nil} \end{bmatrix} \right\}$$

Note that S_G has neither immediate sequence successors nor predecessors.
After applying t_7, the following is obtained:

$$\left\{ \begin{bmatrix} \begin{bmatrix} S_6; \\ S_7 \end{bmatrix}; \\ \left[\ \alpha \Leftarrow e \ \|\ \left[\ \begin{matrix} S_8; \\ S_9 \end{matrix}\ \right]\right]; \\ \textbf{nil} \end{bmatrix} => \begin{bmatrix} \textbf{nil}; \\ \left[\ \begin{bmatrix} \begin{bmatrix} S_6; \\ S_7 \end{bmatrix}; \\ \left[\ \alpha \Leftarrow e \ \|\ \left[\ \begin{matrix} S_8; \\ S_9 \end{matrix}\ \right]\right]; \\ \textbf{nil} \end{bmatrix} \| \textbf{nil}\ \right]; \\ \textbf{nil} \end{bmatrix} \right\}$$

* Obtaining S_G, which corresponds to G_2^r:

$$\left[\begin{array}{c} \\ \left[\left[\left[\begin{array}{c} \left[\begin{array}{c} S_6; \\ S_7 \end{array}\right]; \\ \left[\begin{array}{c} \alpha \Leftarrow e \ \| \left[\begin{array}{c} S_8; \\ S_9 \end{array}\right] \end{array}\right]; \\ \mathbf{nil} \end{array}\right] \| \mathbf{nil}\right]; \\ \\ \end{array}\right.$$

$$\left.\begin{array}{c} \mathbf{nil}; \\ \\ \mathbf{nil} \end{array}\right]$$

$*$ *level* is incremented by 1, then $level = 3$ and the procedure exits.

- At the end, t_8, the output S' is the following:

$$S' :: \left[\begin{array}{l} \left[\begin{array}{c} S_0; \\ \left[\left[\begin{array}{c} \left[\begin{array}{c} \left[\begin{array}{c} S_3; \\ S_4 \end{array}\right]; \\ [\ \alpha \Rightarrow u \ \| \ \mathbf{nil}\ ]; \\ S_5 \end{array}\right] \| [\ S_1 \ \| \ S_2\ ] \end{array}\right]; \\ \mathbf{nil} \end{array}\right] \\ \\ \| \\ \\ \left[\begin{array}{c} \\ \left[\left[\left[\begin{array}{c} \left[\begin{array}{c} S_6; \\ S_7 \end{array}\right]; \\ \left[\begin{array}{c} \alpha \Leftarrow e \ \| \left[\begin{array}{c} S_8; \\ S_9 \end{array}\right] \end{array}\right]; \\ \mathbf{nil} \end{array}\right] \| \mathbf{nil}\right]; \\ \\ \end{array}\right. \\ \left.\begin{array}{c} \mathbf{nil}; \\ \\ \mathbf{nil} \end{array}\right] \end{array}\right]$$

After constructing the G top statements, both statements have the same structural order.

❑ STEP3 - Communication elimination. The trace of the execution of STEP3 is the following:

- Step e_1 of procedure STEP3 checks whether the communication statement denoted by ℓ is an input or an output. In the example, G_0^l denotes an input, then the cooperation commutativity law is applied:

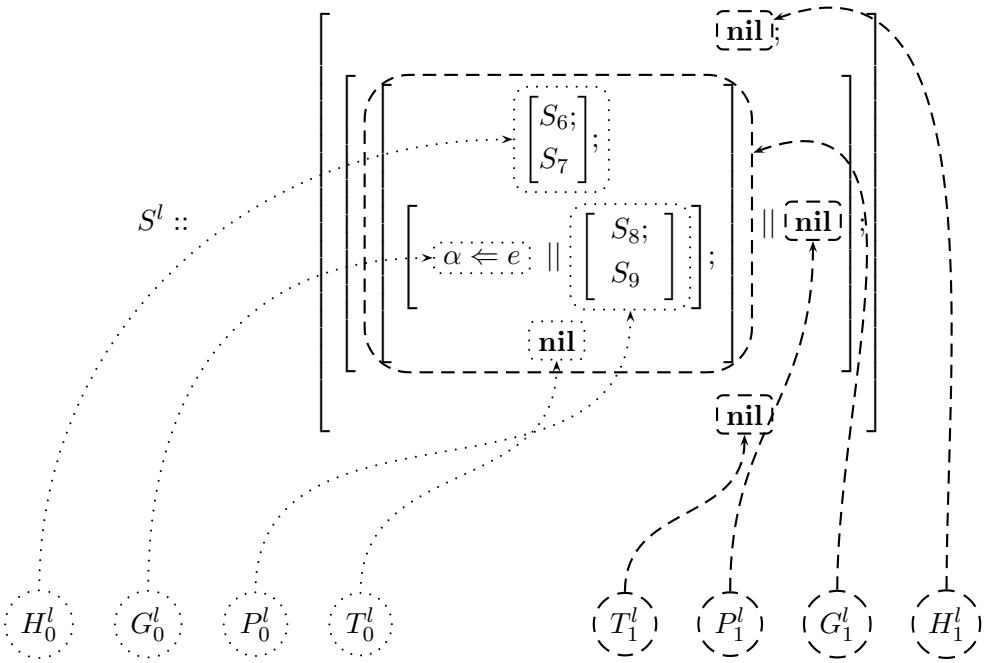
$$\left\{ \quad \left[S_n^l(\ell) \;||\; S_n^r(m)\right] \quad => \quad \left[S_n^r(m) \;||\; S_n^l(\ell)\right] \right\}$$

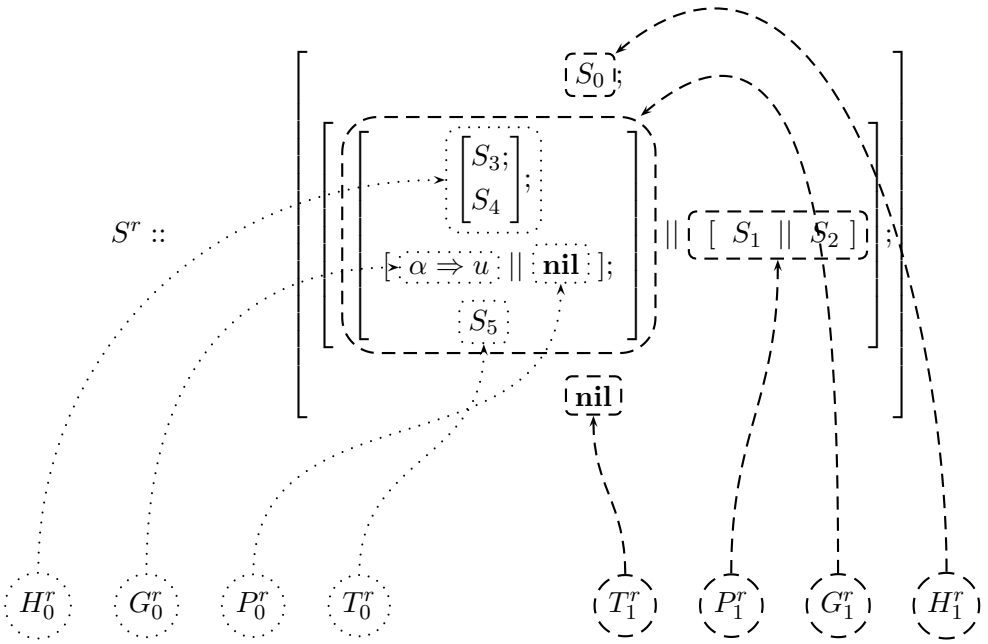
the resulting form S is:

$$\left[\begin{array}{l}
\left[\begin{array}{c} \\ \mathbf{nil}; \end{array}\right] \\
\left[\begin{array}{l}
\left[\begin{array}{l}
\left[\begin{array}{c} \\ \left[\begin{array}{c} S_6; \\ S_7 \end{array}\right]; \end{array}\right] \\
\left[\begin{array}{l} \alpha \Leftarrow e \;||\; \left[\begin{array}{c} S_8; \\ S_9 \end{array}\right] \end{array}\right]; \\
\mathbf{nil}
\end{array}\right] \;||\; \mathbf{nil}\; ; \\
\mathbf{nil}
\end{array}\right] \\
\;||\; \\
\left[\begin{array}{c}
S_0; \\
\left[\begin{array}{l}
\left[\begin{array}{l}
\left[\begin{array}{c} S_3; \\ S_4 \end{array}\right]; \\
[\; \alpha \Rightarrow u \;||\; \mathbf{nil} \;]; \\
S_5
\end{array}\right] \;||\; [\; S_1 \;||\; S_2 \;]\; ; \\
\mathbf{nil}
\end{array}\right]
\end{array}\right]
\end{array}\right]$$

Now S matches perfectly the communication elimination law of order 2.

- Next schemas show all substatements that take part in the applicability conditions of the communication elimination law:


$S^l ::$
nil ;
S_6;
S_7
;
$\alpha \Leftarrow e$
||
S_8;
S_9
;
nil
||
nil
;
nil
H_0^l
G_0^l
P_0^l
T_0^l
T_1^l
P_1^l
G_1^l
H_1^l


$S^r ::$
S_0 ;
S_3;
S_4
;
$\vdash \alpha \Rightarrow u$
||
nil
];
S_5
||
[S_1 || S_2]
;
nil
H_0^r
G_0^r
P_0^r
T_0^r
T_1^r
P_1^r
G_1^r
H_1^r

- Before applying the elimination, the procedure verifies the applicability conditions.
  The first one is e_2: the pairs (P_i^l, T_k^r) do not communicate, for $k \in [0, n-1]$ and $i \in [0, k]$. Detailing the pairs:
  - The pair (P_0^l, T_0^r):: $\left(\quad [S_5] , \quad \begin{bmatrix} S_8; \\ S_9 \end{bmatrix} \right)$ does not communicate. Since our example is generic, S_5, S_8, and S_9 are not detailed, hence the above pairs can not be evaluated.
    For instance, suppose the case: $\left(\quad [\gamma \Leftarrow a] , \quad \begin{bmatrix} \gamma \Rightarrow b; \\ b := b+1 \end{bmatrix} \right)$, where $S_5 :: [\gamma \Leftarrow a]$, $S_8 :: [\gamma \Rightarrow b]$, and $S_9 :: [b := b+1]$.
    The synchronous channel γ establishes a communication between P_0^l and T_0^r, in this case the applicability condition becomes false.
  - (P_0^l, T_1^r):: $\left(\quad [\mathbf{nil}] , \quad \begin{bmatrix} S_8; \\ S_9 \end{bmatrix} \right)$. Since T_1^r is **nil**, the pair does not communicate.
  - $(P_1^l, T_1^r) :: ([\mathbf{nil}], [\mathbf{nil}])$ does not communicate.
- Applicability condition e_3: the pairs (P_i^r, T_k^l) do not communicate for $k \in [0, n-1]$ and $i \in [0, k]$.
  - $(P_0^r, T_0^l) :: ([\mathbf{nil}], [\mathbf{nil}])$ does not communicate.
  - $(P_0^r, T_1^l) :: ([\mathbf{nil}], [\mathbf{nil}])$ does not communicate.
  - $(P_1^r, T_1^l) :: ([\mathbf{nil}], [S_1 \parallel S_2])$ does not communicate.
- Applicability condition e_4: the pairs (T_i^l, T_j^r) do not communicate, for $i, j \in [0, n-1]$, $i \neq j$. The pairs are:
  - (T_0^l, T_1^r):: $([\mathbf{nil}], [\mathbf{nil}])$ does not communicate.
  - (T_1^l, T_0^r):: $([\mathbf{nil}], [S_5])$ does not communicate.
- Communication order restriction e_5: within P_k^l for $k \in [1, n-1]$ and $i \in [0, k-1]$,

$$cw(P_i^l) < cw(T_i^r) \ \bigwedge \ cw(P_i^r) < cw(T_i^l) \ \bigwedge \ cw(G_i^l) < cw(T_i^r) \ \bigwedge \ cw(G_i^r) < cw(T_i^l)$$

  The restrictions respect to P_1^l are:

$$cw(P_0^l) < cw(T_0^r) \ \bigwedge \ cw(P_0^r) < cw(T_0^l) \ \bigwedge \ cw(G_0^l) < cw(T_0^r) \ \bigwedge \ cw(G_0^r) < cw(T_0^l)$$

- Communication order restriction e_6: within P_k^r for $k \in [1, n-1]$ and $i \in [0, k-1]$,

$$cw(P_i^l) < cw(T_i^r) \ \bigwedge \ cw(P_i^r) < cw(T_i^l) \ \bigwedge \ cw(G_i^l) < cw(T_i^r) \ \bigwedge \ cw(G_i^r) < cw(T_i^l)$$

  The restrictions respect to P_1^r are:

$$cw(P_0^l) < cw(T_0^r) \ \bigwedge \ cw(P_0^r) < cw(T_0^l) \bigwedge \ cw(G_0^l) < cw(T_0^r) \ \bigwedge \ cw(G_0^r) < cw(T_0^l)$$

- The communication elimination law for order 2 is applied at step e_7, the obtained form, S', is the following:

$$S' :: \left[\begin{array}{c} \left[\ \mathbf{nil} \ \| \ S_0 \ \right]; \\ \left[\begin{array}{c} \left[\begin{array}{c} \left[\begin{array}{c} \left[\begin{array}{c} S_6; \\ S_7 \end{array} \right] \ \| \ \left[\begin{array}{c} S_3; \\ S_4 \end{array} \right] \end{array} \right]; \\ \left[\ u := e \ \| \ \left[\begin{array}{c} S_8; \\ S_9 \end{array} \right] \ \| \ \mathbf{nil} \ \right]; \\ \left[\ \mathbf{nil} \ \| \ S_5 \ \right] \end{array} \right] \| \ \mathbf{nil} \ \| \ [\ S_1 \ \| \ S_2 \ ] \end{array} \right]; \\ \left[\ \mathbf{nil} \ \| \ \mathbf{nil} \ \right] \end{array} \right]$$

❑ STEP4 - Elimination of redundant *nil* statements. Statement S' contains some redundant *nil* statements introduced in STEP2. These are removed as follows:

- For $k = 0$:
  - ∗ Applying the reduction n_3:

$$\left\{ \ \ [\ G_0 \ \| \ P_0^l \ \| \ \mathbf{nil} \ ] \Rightarrow [\ G_0 \ \| \ P_0^l \ ] \ \right\}$$

  one obtains:

$$\left\{ \ \left[\ u := e \ \| \ \left[\begin{array}{c} S_8; \\ S_9 \end{array} \right] \ \| \ \mathbf{nil} \ \right] \Rightarrow \left[\ u := e \ \| \ \left[\begin{array}{c} S_8; \\ S_9 \end{array} \right] \ \right] \ \right\}$$

* S_{GP} is the above resulting form $\left[\begin{array}{c} u := e \end{array} \;||\; \left[\begin{array}{c} S_8; \\ S_9 \end{array}\right] \right]$

* Applying the reduction n_5:

$$\left\{ \quad \left[\begin{array}{c} [\ S_{GP}\ ]; \\ [\ \mathbf{nil}\ ||\ T_0^r\ ] \end{array}\right] => [\ S_{GP};\ T_0^r\ ] \right\}$$

obtaining:

$$\left\{ \quad \left[\begin{array}{c} \left[\begin{array}{c} u := e \end{array} \;||\; \left[\begin{array}{c} S_8; \\ S_9 \end{array}\right] \right]; \\ [\ \mathbf{nil}\ ||\ S_5\ ] \end{array}\right] => \left[\begin{array}{c} \left[\begin{array}{c} u := e \end{array} \;||\; \left[\begin{array}{c} S_8; \\ S_9 \end{array}\right] \right]; \\ S_5 \end{array}\right] \right\}$$

- For $k = 1$:

  * Applying the reduction n_2:

$$\left\{ \quad [\ G_1\ ||\ \mathbf{nil}\ ||\ P_1^r\ ] => [\ G_1\ ||\ P_1^r\ ] \right\}$$

one obtains:

$$\left\{ \quad \left[\begin{array}{c} \left[\begin{array}{c} \left[\begin{array}{c} \left[\begin{array}{c} S_6; \\ S_7 \end{array}\right] \;||\; \left[\begin{array}{c} S_3; \\ S_4 \end{array}\right] \end{array}\right]; \\ \left[\begin{array}{c} u := e \end{array} \;||\; \left[\begin{array}{c} S_8; \\ S_9 \end{array}\right] \right]; \\ S_5 \end{array}\right] \;||\; \mathbf{nil} \;||\; [\ S_1\ ||\ S_2\ ] \end{array}\right] => \right.$$

$$\left. \left[\begin{array}{c} \left[\begin{array}{c} \left[\begin{array}{c} \left[\begin{array}{c} S_6; \\ S_7 \end{array}\right] \;||\; \left[\begin{array}{c} S_3; \\ S_4 \end{array}\right] \end{array}\right]; \\ \left[\begin{array}{c} u := e \end{array} \;||\; \left[\begin{array}{c} S_8; \\ S_9 \end{array}\right] \right]; \\ S_5 \end{array}\right] \;||\; [\ S_1\ ||\ S_2\ ] \end{array}\right] \right\}$$

* Now S_{GP} denotes the above resulting form.

* Reduction n_4:

$$\left\{ \quad \left[\begin{array}{c} [\ S_{GP}\ ]; \\ [\ \mathbf{nil} \parallel \mathbf{nil}\ ] \end{array} \right] => [\ S_{GP}\ ] \right\}$$

after applying it, the following is obtained:

$$\left\{ \quad \left[\begin{array}{c} \left[\begin{array}{c} \left[\begin{array}{c} \left[\begin{array}{c} \left[\begin{array}{c} S_6; \\ S_7 \end{array} \right] \parallel \left[\begin{array}{c} S_3; \\ S_4 \end{array} \right] \end{array} \right]; \\ \left[\begin{array}{c} u := e \parallel \left[\begin{array}{c} S_8; \\ S_9 \end{array} \right] \end{array} \right]; \\ S_5 \end{array} \right] \parallel [\ S_1 \parallel S_2\ ] \end{array} \right]; \\ [\ \mathbf{nil} \parallel \mathbf{nil}\ ] \end{array} \right] =>$$

$$\left[\begin{array}{c} \left[\begin{array}{c} \left[\begin{array}{c} \left[\begin{array}{c} S_6; \\ S_7 \end{array} \right] \parallel \left[\begin{array}{c} S_3; \\ S_4 \end{array} \right] \end{array} \right]; \\ \left[\begin{array}{c} u := e \parallel \left[\begin{array}{c} S_8; \\ S_9 \end{array} \right] \end{array} \right]; \\ S_5 \end{array} \right] \parallel [\ S_1 \parallel S_2\ ] \end{array} \right] \quad \right\}$$

* S_{GPT} is assigned to the above resulting form.

* Applying the reduction n_8:

$$\left\{ \quad \left[\begin{array}{c} [\ \mathbf{nil} \parallel H_1^r\ ]; \\ [\ S_{GPT}\ ] \end{array} \right] => [\ H_1^r;\ S_{GPT}\ ] \right\}$$

one obtains:

$$\left\{ \left[\left[\left[\begin{array}{c} [\,\mathbf{nil} \parallel S_0\,]; \\ \left[\left[\begin{bmatrix} S_6; \\ S_7 \end{bmatrix} \parallel \begin{bmatrix} S_3; \\ S_4 \end{bmatrix} \right]; \\ \left[u := e \parallel \begin{bmatrix} S_8; \\ S_9 \end{bmatrix} \right]; \\ S_5 \end{array} \right] \parallel [\, S_1 \parallel S_2 \,] \end{array} \right] ; \right] \right] \Rightarrow \right.$$

$$\left. \left[\begin{array}{c} S_0; \\ \left[\left[\begin{bmatrix} S_6; \\ S_7 \end{bmatrix} \parallel \begin{bmatrix} S_3; \\ S_4 \end{bmatrix} \right]; \\ \left[u := e \parallel \begin{bmatrix} S_8; \\ S_9 \end{bmatrix} \right]; \\ S_5 \end{array} \right] \parallel [\, S_1 \parallel S_2 \,] \end{array} \right] \right\}$$

- Finally $S' ::$

$$\left[\begin{array}{c} S_0; \\ \left[\left[\begin{bmatrix} S_6; \\ S_7 \end{bmatrix} \parallel \begin{bmatrix} S_3; \\ S_4 \end{bmatrix} \right]; \\ \left[u := e \parallel \begin{bmatrix} S_8; \\ S_9 \end{bmatrix} \right]; \\ S_5 \end{array} \right] \parallel [\, S_1 \parallel S_2 \,] \end{array} \right]$$

❑ STEP5 - Elimination of redundant statement associations.

- Sequence association: in this case there no associations to be removed by applying CONCATFLAT, line a_1.
- Parallelism association:
  - ∗ One parallelism association is found in S'. It is shown below within a dot frame. The statement to be flattened is denoted by the auxiliary S_P:

$$S_P :: \left[\left[\left[\left[\begin{bmatrix} S_6; \\ S_7 \end{bmatrix} \;||\; \begin{bmatrix} S_3; \\ S_4 \end{bmatrix}\right]; \\ \left[u := e \;||\; \begin{bmatrix} S_8; \\ S_9 \end{bmatrix}\right]; \\ S_5 \end{array}\right] \;\Big|\Big|\; \left[S_1 \;||\; S_2\right]\right]$$

This step applies reduction COOPFLAT, line a_2.

❑ FINAL EQUIVALENCE

When the communication elimination reduction ends, the following interface equivalence holds:

$$S \;=_{\mathcal{O}}\; S'$$

which, substituting statements S and S', becomes:

$$S :: \left[\left[\left[\begin{array}{c} S_0; \\ \left[S_1 \;||\; S_2 \;||\; \left[\begin{array}{c} S_3; \\ S_4; \\ \alpha \Rightarrow u; \\ S_5 \end{array}\right]\right] \end{array}\right]\right] \;||\; \left[\begin{array}{c} S_6; \\ S_7; \\ \left[\left[\begin{array}{c} S_8; \\ S_9 \end{array}\right] \;||\; \alpha \Leftarrow e\right] \end{array}\right]\right]$$

$$=_{\mathcal{O}}$$

$$S' :: \left[\begin{array}{c} S_0; \\ \left[\left[\begin{array}{c} S_6; \\ S_7 \end{array}\right] \;||\; \left[\begin{array}{c} S_3; \\ S_4 \end{array}\right]\right]; \\ \left[u := e \;||\; \left[\begin{array}{c} S_8; \\ S_9 \end{array}\right]\right]; \\ S_5 \end{array}\right] \;||\; S_1 \;||\; S_2$$

Observe that the matching communication pair, $(\alpha \Rightarrow u, \alpha \Leftarrow e)$ within S, has been eliminated, and in S' only remains the assignment $u := e$. The communication elimination reduction also transforms many other statements, in the following some of them are enumerated.

- S_0 is parallel to all S^r statements of S, but it does not remain parallel to any in S'.
- The concatenation $[\ S_3; S_4\ ]$ remains in parallel with $[\ S_6; S_7\ ]$ after the transformation, but $[\ S_8; S_9\ ]$ is in sequence in S'.
- $[\ S_6; S_7\ ]$ is parallel to S_5 in S but in sequence in S'.
- $S_1 \mid\mid S_2$ is always in parallel to the other statements either in S or S'.

In general the substatements which were not parallel in S continue being not parallel in S', and some substatements which were parallel in S, now are connected in sequence in S'.

4.4 Elimination from a k-ary Cooperation

The previous section dealt with binary communication elimination, the removal of a matching communication pair from a selection-free BCS, of the form $[S^l||S^r]$.

This section presents the communication elimination from a k-ary cooperation statement, $S :: [\cdots \mid\mid S^l(\ell) \mid\mid \cdots \mid\mid S^r(m) \mid\mid \cdots]$, based on Theorem 5 of page 54. Any k-ary cooperation statement can always be transformed into $S([S^l(\ell)||S^r(m)])$, and within the obtained binary cooperation association the communication elimination can be applied. Next procedure details the implementation.

Procedure COMELI – elimination form a selection-free BCS

Input: $S :: [\cdots \mid\mid S^l(\ell) \mid\mid \cdots \mid\mid S^r(m) \mid\mid \cdots]$, a selection-free BCS, and $p = (\ell, m)$ a matching communication pair.

Output: S' a selection-free BCS equivalent to S, where p has been eliminated, or a failure indication.

$S' := S$
if S' *contains > 2 parallel top statements* **then**
  $i_\ell :=$ Index of the top statement of S' which contains ℓ.
  $i_m :=$ Index of the top statement of S' which contains m.
  $S' ::$ BINCOOPASSO(S', i_ℓ, i_m)
(*failure*, S') :: BIN-COMELI(S', p)

Basically, it calls the communication elimination, procedure BIN-COMELI, after applying the binary parallelism association transformation, BINCOOPASSO, intro-

duced in the next section 5.4 of page 127. Procedure BINCOOPASSO obtains a cooperation where statements $S^l(\ell)$ and $S^r(m)$ have been associated according to the indexes i_ℓ and i_m.

Computational complexity

The procedure calls BINCOOPASSO and BIN-COMELI. BINCOOPASSO depends on the number of cooperation statements within S, in this case k. The complexity of BIN-COMELI is of order n^2, as shown above in page 86.

$$C_{\text{COMELI}} = O(n^2 + n \times st + k)$$

where n is the structural order of $[S^l(\ell)||S^r(m)]$.

4.5 General Communication Elimination

The last step is the elimination of all pairs within an input statement S. Next procedure, GEN-COMELI, is the implementation of the one of page 55. It applies iteratively procedure COMELI and tries to eliminate the first matching pair, p, obtained from the competing pairs found within S'. The competing pairs are formed with communication substatements found by procedure COMFRONT , commented in next page. The elimination order of the pair is not important as lemma 9 of page 55 establishes. The procedure, illustrated in the next page, ends in one of three following states:

- **Success**: the iteration terminates and all inner matching pairs have been eliminated, it means that no communication left in the output program S', the communication front is empty.

- **Deadlock**: some communications remains within S', COMFRONT$(I, S') \neq \emptyset$, but no more matching pairs are found.

- **Failure**: when an applicability condition of a communication elimination law is not satisfied, procedure BIN-COMELI within COMELI reports a failure and the procedure ends after exiting the *while* loop.

Procedure GEN-COMELI − general elimination form a selection-free BCS

Input: $S :: [S_1 \,||\, S_2 \,||\, \cdots \,||\, S_k]$, a selection-free BCS.

Output: S' a selection-free BCS equivalent to S, where all matching pairs have been eliminated, or a failure indication.

```
S' := S
failure := false
I := the set of internal channels of S'
while ¬failure ∧ { S' has a competing pair p } do
  (failure, S') :: COMELI(S', p)

if ¬failure then
  if COMFRONT(I, S')=∅ then
    terminate with success
  else
    terminate with deadlock
else
  terminate with failure
```

Procedure COMFRONT calculates the *communication front* traversing the statement recursively, searching for internal communication substatements in its concatenation ordering. Therefore, its computational complexity is $O(st)$, where st is the number of substatements of the input statement.

Computational complexity of GEN-COMELI

Informally the complexity of procedure GEN-COMELI depends on the number of communication matching pairs to be eliminated, p, within S, and the complexity of COMELI, page 103.

Each iteration tries to remove a matching pair formed with substatements in COMFRONT(I, S'). The main loop consists of p iterations, where p is the number of pairs to be eliminated.

$$C_{\text{GEN-COMELI}} = O(p \times (st + n^2 + n \times st + k)))$$

where,

- n is the structural order of $[S^l(\ell)||S^r(m)]$
- k the number of parallel statements of the LCA parallelism of the communication pair
- st the number of statements of the input S
- p the number of pairs to be eliminated from S

The upper bound for p and n can be assumed to be st, then the complexity is $O(st^3)$.

4.6 Conclusions

The detailed steps of a procedure for the elimination of a single matching pair of communication operations from a selection-free BC statement have been presented in this chapter. This was compulsory since the communication elimination procedure of chapter 3 already used the elimination of a single pair as a fundamental step. After studying the chapter, the reader is prepared to understand the corresponding programs, which are expressed in PADD. Actually, this was one of the motivations, transparency, for going into that level of detail. For truth's sake, some parts of the PADD programs would be written according to the guidelines of this chapter, if they had to be written again.

The bulk of the chapter is on the elimination of a pair from a binary cooperation statement, procedure BIN-COMELI. Expressed in other words, this involves the operations required for the application of a proper communication elimination law to a specific pair belonging to the front of a given statement.

In summary:

(a) The numbers of nested parallelisms embedding each communication operation of the pair (called the two orders n^l and n^r) are determined, the law to be applied depends on them.

(b) The original statement is filled-in with nil substatements via application of basic laws, in preparation for structure matching.

(c) The applicability conditions of chapter 3 are checked; if satisfied the law is applied, otherwise return with failure occurs.

(d) Remove redundant structures embedding nil statements, a cleanup final step.

The chapter has detailed the evolution of the procedure on a simple, order 2, example.

For the sake of completeness, procedure GEN-COMELI for the elimination of pairs from an n-ary cooperation has been detailed. Computational complexity of GEN-COMELI has been evaluated along the chapter, concluding to be of polynomial complexity in the maximum order of $O(st^3)$.

Chapter 5

THE INTERACTIVE PROVER TOOL

Communication elimination and distributed program sequentialization proofs, cannot be hand constructed. A tool is needed. This chapter gives details about the interactive prover which has been developed. Given a system model, program, the objective is obtaining an interface equivalent program by applying only laws and transformation procedures. The tool guarantees that the program changes as a result of these applications only. It integrates the laws described in section 2.9 of chapter 2, and reduction procedures such as communication elimination, procedure substitution, forward variable elimination, cooperation and concatenation permutation, etc., which are detailed in section 5.4 below and in chapter 4. All applied steps are stored to allow browsing of the proof, at a later stage. Among the objectives of this chapter is the provision of information required both to use and to continue development of the prover. The following are some of the overviewed topics: Input statement inner representation and preprocessing; representation and application of basic laws; and the transformation procedures.

5.1 Interface Components and Overview

The prover has the following interface components:

Top-level window. The laws and transformation procedures are selected and applied to the input statement from this window. Each application results into a proof step.

Navigation tree window. To browse the already applied proof steps, moving from one to another, or deleting a step.

Repository of laws window. To browse over the set of laws. Only laws from the repository can be applied. New laws can not be introduced directly into the tool.

Repository of transformation procedures menu. These transformations can be applied, for instance, to simplify the input statement. No new transformation procedures can be introduced since they are embedded within the tool.

The proof steps applied by the user are stored in a file. This file is displayed as a vertical tree on the left hand side of the screen. Each node of the navigation tree, may be unfolded, shows the name and parameters of either the law or the applied transformation procedure. The navigation tree is implemented for browsing purposes. The user can move from the beginning to the end of the proof, or inspect the program form of an intermediate proof step.

5.2 The Input Statement and its Preprocessing

The input statement describes the system or model to work with. Any valid PADD program can be loaded into the tool, from a purely sequential one to a complex one with parallelism and internal communications. The program must contain at least one procedure. It needs to be syntactically correct and to have correct typing of variables and expressions. The prover tool has been developed assuming this syntactic correctness. The tool displays the program on the right hand side of the screen. Internally the prover codifies the input statement in an expanded PADD notation. This is done to simplify the structure matching involved in the application of the laws and reductions.

The expanded notation represents the expressions within the textual nodes of the input PADD program in tree-like expanded form. The expansion builds a new tree where each node contains either variable and function identifiers, terminal symbols or numbers. The expansion preserves the input tree schema. This means that the vertical predecessor and successor of each original node is not changed. The expansion creates only diagonal and/or horizontal subtrees.

The main terminal symbols are:

`:=`	assignment symbol
`()`	function or array symbol
`true` | `false`	boolean values
`+` | `-` | `*` | `/`	infix arithmetic operators
`<` | `>` | `<=` | `=>` | `=` | ¬`=`	infix relational operators
¬	negation operator
`<>` | `[]`	input and output connections operators
∞	indefinite loop symbol
`?`	selection symbol
`||`	parallelism symbol
`*`	iteration symbol
`#`	comment symbol
`:`	declaration separator
`..`	subrange operator

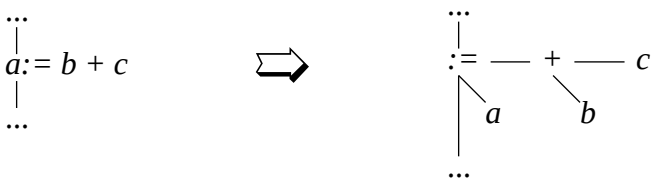
In general all PADD keywords are terminal symbols.

A PADD parser, implemented within the prover, analyzes the expressions of each node of the input statement. It returns a structure where the expression is stored in tree form.

The following are examples of nodes which are expanded:

Assignments:

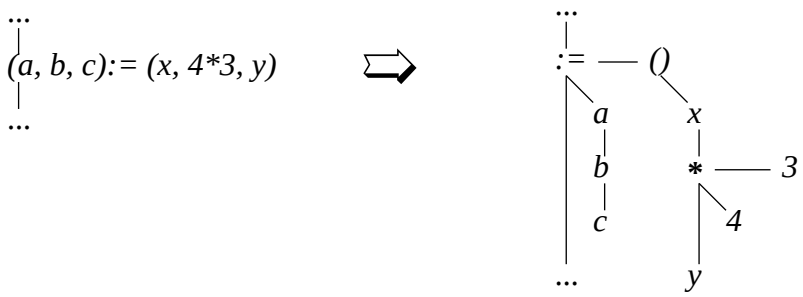
Consider the following example:



The left hand side of ':=' in the expression $a := b + c$, variable a, is moved after the expansion as a new diagonal node, while the right hand side is expanded as

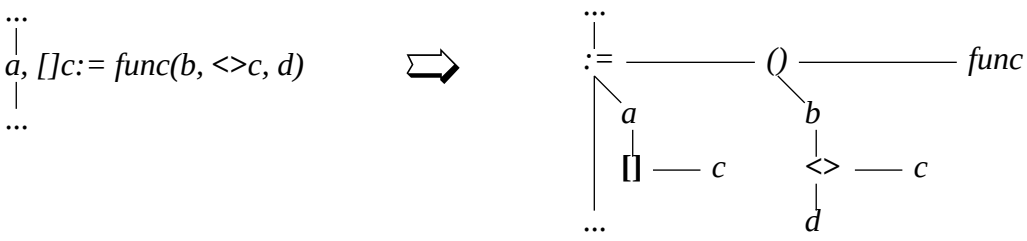
a new horizontal one. The same criteria is applied to $b + c$, the LHS of '+', b, is expanded to a diagonal node and c to a horizontal one.

In the case of multiple assignments, the LHS creates a new diagonal subtree with a vertical list of variables to be written, similarly for the RHS but inserting the auxiliary terminal symbol '()'. This is shown in the following example:



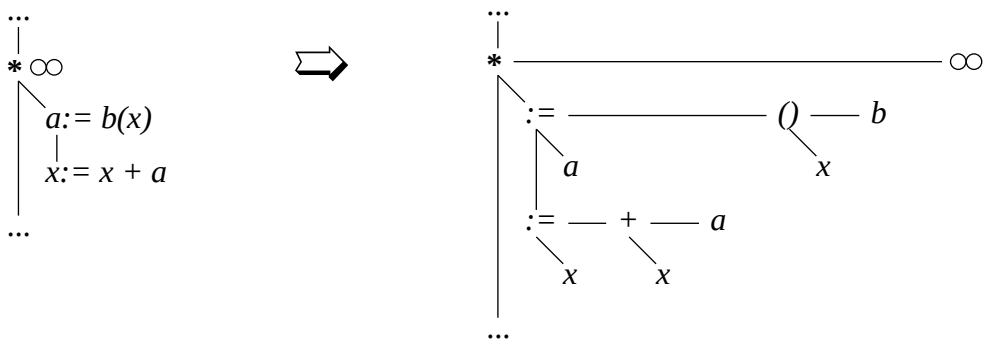
Procedure References:

This expansion is similar to the assignment. The procedure name, $func$, is moved to the horizontal node of the procedure reference symbol, (), while the results and parameters of the procedure are listed in sequential order constructing a new subtree in the diagonal of " := " and () respectively. Observe the connections in the expanded form, the connection identifier is moved to the horizontal node of the connection operator.



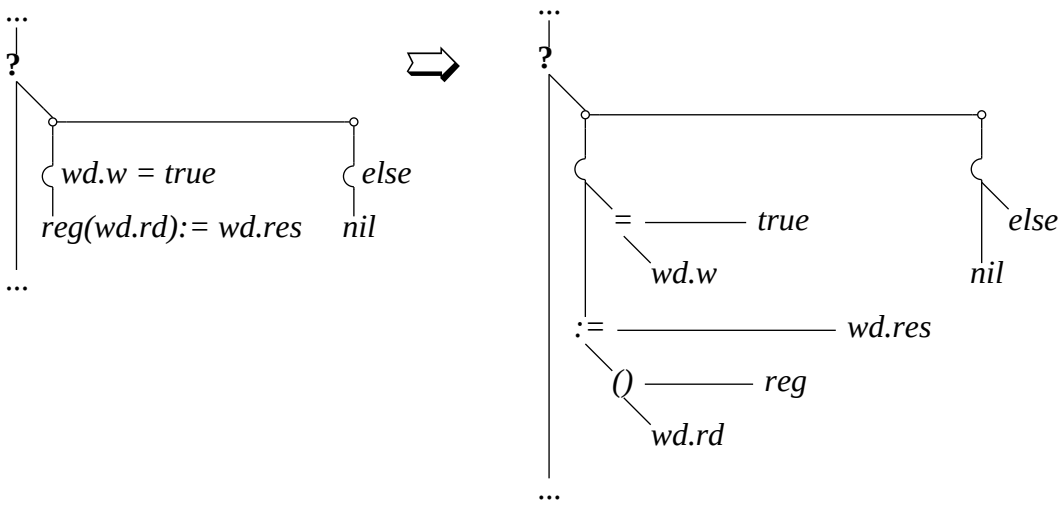
Iterations:

The iteration symbol remains in the same node, and the loop forever symbol is expanded to its horizontal node. The body of the iteration is transformed as the next example shows.



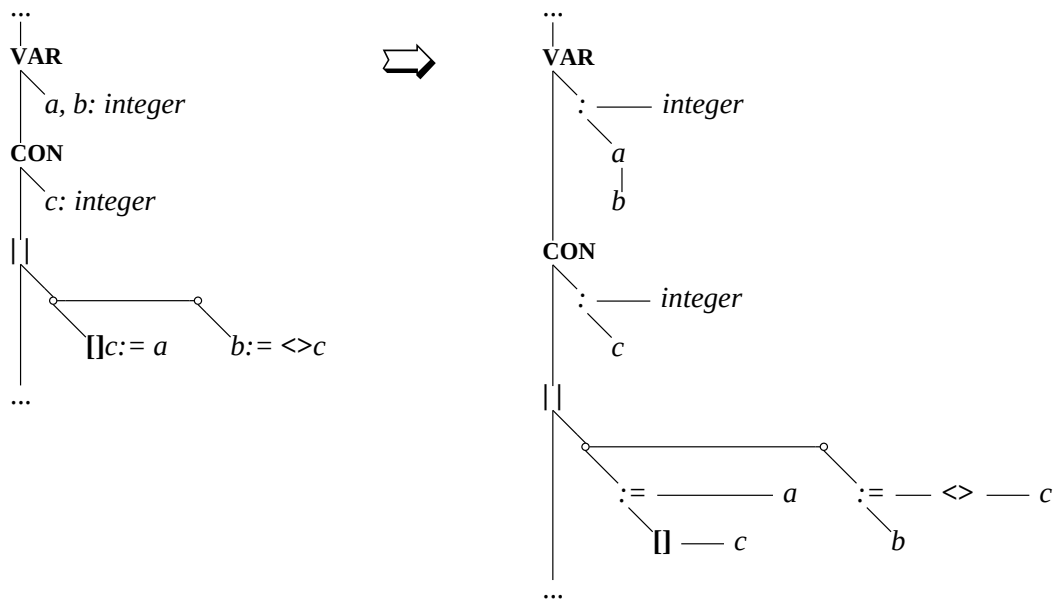
Selections:

The basic selection structure remains unchanged, only the condition is expanded as an expression with two operands and one infix operator. Note that the arrays are expanded as procedure references, such as $reg(wd.rd)$ in the following example.



Declarations:

Variable and connection declarations are also expanded. The colon symbol, :, is treated as an infix operator.



5.3 The Basic Laws: Representation and Application

5.3.1 Notational Conventions

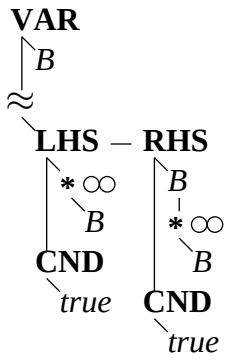
Both the law application and the transformations are applied to a certain point within the input statement, S. This applicability point is indicated as p **within** S, where p stands also for the substatement within S starting at the point. In many cases, p corresponds to a sequence of substatements. Usually the transformation procedures are applied only to the top leftmost substatement of p.

5.3.2 Notation for the Laws

The laws presented in chapter 2 were expressed in textual form, but within the prover tool they are represented in tree-like PADD notation. This facilitates their matching with program statements. For instance, the *Loop Forever Unfold* law from section 2.9 of chapter 2, will illustrate the tree-like law notation:

$$[\,\textbf{loop forever do}\ B\,]\ \approx\ [\,B; \textbf{loop forever do}\ B]$$

which written in tree notation looks as follows:



In the above tree notation, both sides of the ≈ symbol in the textual form are represented as horizontal sub-trees. The left hand side, [**loop forever do** B] is written under the scope of the keyword LHS of the tree, and the right hand side under RHS.

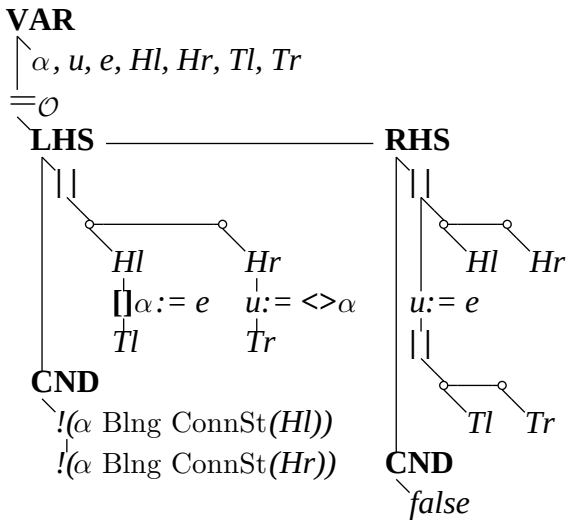
The law notation introduces the concept of *variables for matching*. They are located and declared under the scope of the keyword VAR. Variables are separated by commas and must be alphanumeric identifiers. These variables are always of type *statement*.

The laws have applicability conditions. In the tree notation they are located under the scope of the keyword CND. Next subsection 5.3.3.2 will detail them. In general the laws can be applied from left to right and vice-versa. Applying the law from left to right, the conditions to be fulfilled are the ones located under LHS. Similarly for RHS. If the law does not have applicability conditions, as the above one, then conditions are true. A false in CND indicates that the law can not be applied in that direction.

As a second example, the basic communication elimination law is presented, it was introduced in chapter 3. The textual form is:

$$[H^l;\ \alpha \Leftarrow e;\ T^l] \mid\mid [H^r;\ \alpha \Rightarrow u;\ T^r] \ =_{\mathcal{O}}\ [H^l \mid\mid H^r];\ u := e;\ [T^l \mid\mid T^r]$$

and in tree notation:



This law can only be applied from left to right, as its right hand side condition declares. In this case, the LHS conditions, relates channel α with Hl and Hr statements. Explicitly neither Hl nor Hr should contain any reference to channel α to apply successfully the law. The syntax and semantics are explained below in subsection 5.3.3.2.

5.3.3 Procedure Apply

The application of a law involves several steps. These are encapsulated into procedure APPLY, whose algorithmic details are shown below. It applies the input law l at a specific point of S, and obtains an equivalent program. The application is from left to right or vice-versa depending on the parameter d. The notation of the input statement S is the one introduced in subsection 5.2, also the output S' is in expanded PADD notation. The first step checks whether the statement at location p of the input statement, written as $S[p]$, matches the side of law l indicated by d. In case of applicability conditions, they must be fulfilled, and then the transformation is carried out.

Procedure APPLY − apply a law

Input: S is the input statement in expanded PADD notation, p is the applicability substatement within S, l is the law name, and d is the applicability direction of l.

Output: S' a program equivalent to S, where l has been applied, or a failure indication.

Structure Matching: Determine whether the side of law l matches p within S, and outputs a list r with the matching results.

if $S(p)$ *matches* l **then**

  Verification of Conditions: Check the applicability conditions of law l over S.

  if *conditions of* l *are satisfied* **then**

    Transformation: Proper application of law l, S is transformed into S'.

  else

    terminate with *failure*, applicability conditions of l are not fulfilled.

else

  terminate with *failure*, p within S and l do not match.

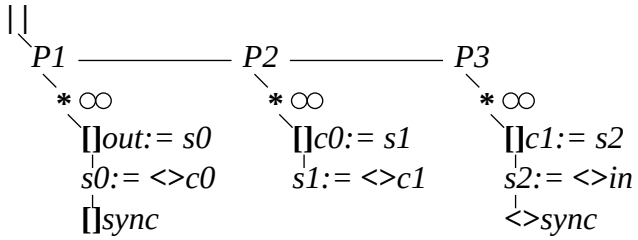
Next subsections will explain each step of the procedure: *Structure Matching*, *Verification of Conditions*, and *Transformation*.

5.3.3.1 Structure Matching

In general, given a tree T and a pattern P, which is also a tree, P matches T, if there exists a one-to-one mapping from the nodes of P into the homologous nodes of T. In case of a pattern matching with variables, P can contain "wild-card" variables (called *variables for matching*). Note that a variable can match an entire subtree within T. Tree pattern matching is used in a number of programming tasks such as mechanical theorem proving, term rewriting, or symbolic computation. Several techniques and algorithms are described in [AC75, HO82, RR92, SZ97]. The pattern matching is used in the simplification of tree expressions.

In this step of procedure APPLY, a list r is created if the structure matching succeeds. This list contains the relation between the variables of the law l and their matching subtrees of the input statement S.

Consider program S_1,

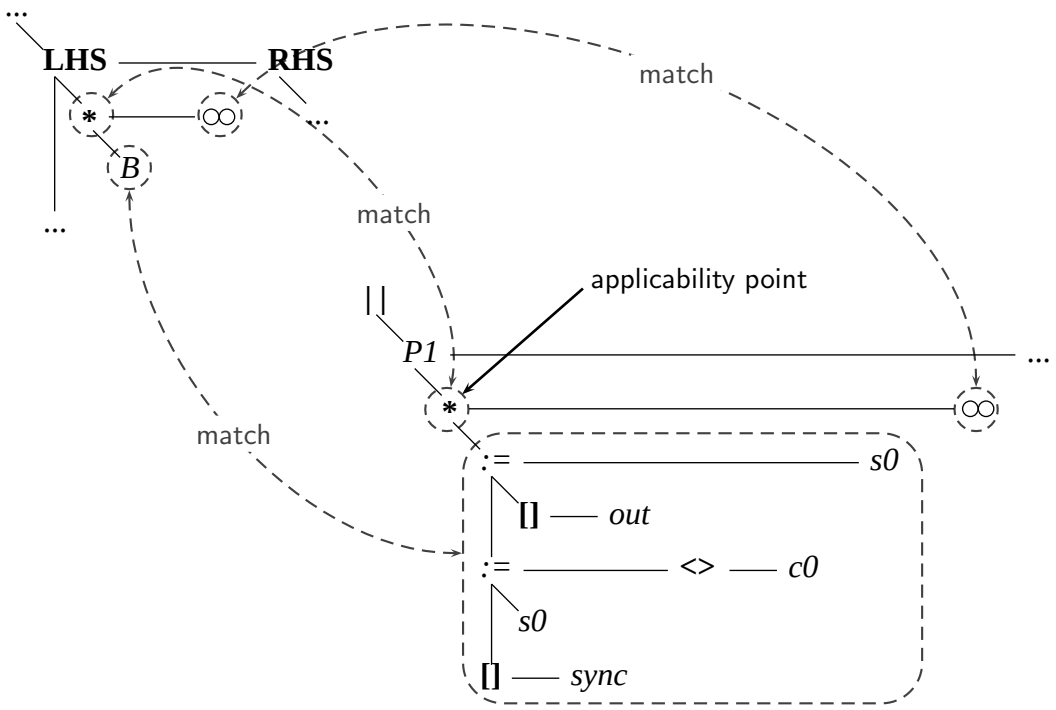


where *in* and *out* are external channels, $c0$, $c1$ and *sync* are internal channels, and $s0$, $s1$, and $s2$ are local variables.

As an example, the unfolding law is applied to S_1 from left to right:

$$[\,\textbf{loop forever do } B\,] \;\approx\; [\,B;\textbf{loop forever do } B]$$

For instance, the applicability point is the first loop forever symbol of the leftmost parallel process of S_1. The following illustration shows the result of the structure matching procedure, both the law and S_1 are in expanded notation:



The matching procedure traverses S_1 comparing its structure with the law, which is the pattern. The terminal symbols of the pattern, in this case $*$ and ∞, must match exactly, otherwise the procedure ends with failure. The third element involved in the structure matching is the variable B. It matches anything in the diagonal of $*$. In this case:

$$B \quad :: \quad \begin{bmatrix} []out := s0 \\ s0 := <>c0 \\ []sync \end{bmatrix}$$

The procedure ends without failure when both trees match.

5.3.3.2 Verification of Conditions

The applicability conditions are expressed in terms of set operations and identifiers. *Identifier sets* are introduced after the set operators. The elements of these sets are either variable or connection identifiers of the input statement S. The basic set operations implemented are:

- Unio: Union of two sets.
- Intsc: Intersection of two sets.
- Diff: Difference of two sets.
- Blng: An element belongs to a set.

The identifier sets are detailed next:

- Basic Sets:
  - OutSt(S): Set of all output connection identifiers within statement S.
  - InpSt(S): Set of all input connection identifiers within S.
  - VarRSt(S): Set of all result (written) variables within S.
  - VarPSt(S): Set of all parameter (read) variables of S.
- Compound Sets:
  - ConnSt(S): All connection identifiers of S.
    $\mathsf{ConnSt}(S) = \mathsf{OutSt}(S) \cup \mathsf{InpSt}(S)$
  - RPVSt(S): Result and parameter variable set.
    $\mathsf{RPVSt}(S) = \mathsf{VarRSt}(S) \cap \mathsf{VarPSt}(S)$

- ResSt(S): Result variables and output connections identifier set.
  ResSt(S) = VarRSt(S) ∪ OutPSt(S)
- ParSt(S): Parameter identifiers set. ParSt(S) = VarPSt(S) ∪ InpPSt(S)
- VCnSt(S): Interface identifier set. VCnSt(S) = ResSt(S) ∪ ParSt(S)

For instance the following condition:

$$\mathsf{VCnSt}(S_1)\ \mathsf{Intsc}\ \mathsf{VCnSt}(S_2) = \text{empty}$$

states that substatements S_1 and S_2 share neither connection, nor variables.

The applicability conditions are part of the laws and lemmas. If the conditions are fulfilled then the law can be applied and the program is transformed according to it. Complex laws and lemmas, the ones that can not be expressed as a simple tree notation, are coded as transformation procedures, which also have applicability conditions. Graphically (see page 114) the conditions are located under the scope of the keyword CND, at the bottom of the law schema. Each law can contain several conditions, one below the other, in sequence. This means that they are linked implicitly with a boolean AND operator, thus all of them must be fulfilled before applying the law.

The full grammar of the applicability conditions is the following:

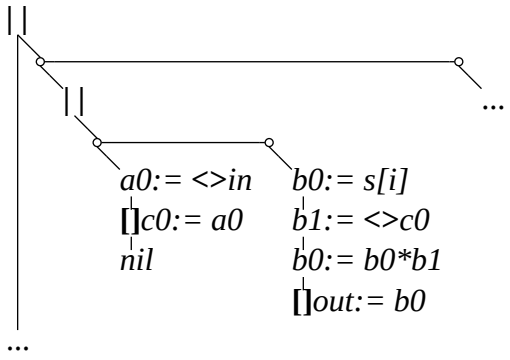
condition	::=	*bool-value*	a boolean value: *true* or *false*
	|	¬ (*condition*) | ! (*condition*)	negation operator
	|	*var* `Blng` *set*	belongs to, a set operator
	|	*set infix set* `= empty`	set operations
infix	::=	`Unio`	union of two sets
	|	`Intsc`	intersection of two sets
	|	`Diff`	difference of two sets
set	::=	`InpSt(`*var*`)` | `OutSt(`*var*`)`	in/out connection sets
	|	`ConnSt(`*var*`)`	connection set
	|	`VarRSt(`*var*`)` | `VarPSt(`*var*`)`	result/parameter sets
	|	`RPVSt(`*var*`)` | `ParSt(`*var*`)`	variable sets
	|	`ResSt(`*var*`)` | `VarSt(`*var*`)`	variable and connection sets
	|	`VCnSt(`*var*`)`	
bool-value	::=	`true` | `false`	
var	::=	*alpha* { *alpha* | *digit* | _ }	a variable identifier
alpha	::=	$[a-zA-Z]$	
digit	::=	$[0-9]$	

Note that `empty` is the keyword for the empty set ($\varnothing$).

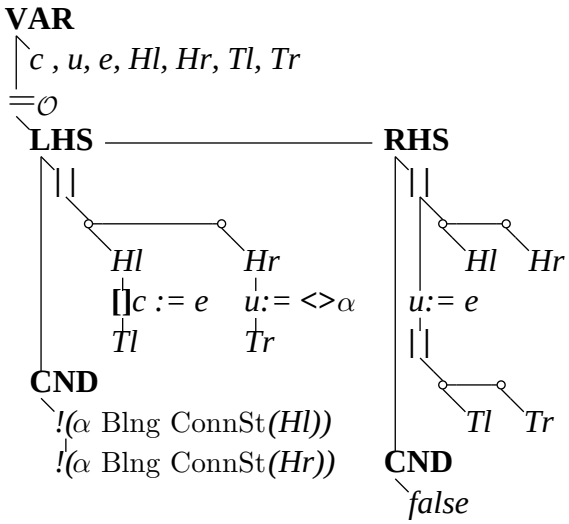
This grammar is enough to express the majority of the laws. However, some complex elimination laws, with more complex applicability conditions, are treated separately as transformation procedures.

5.3.3.3 Example

Given a program S_2:



and the basic communication elimination law:



The law is applied from left to right to the inner parallelism of S_2, obtaining the following matching results:

$$
\begin{array}{lcl}
\alpha & :: & c0 \\
u & :: & b1 \\
e & :: & a0 \\
Hl & :: & a0{:=} <>in \\
Hr & :: & b0{:=}\ s[i] \\
Tl & :: & nil \\
Tr & :: & \left[\begin{array}{l} b0 := b0 * b1 \\ []out := b0 \end{array}\right]
\end{array}
$$

now the applicability conditions can be evaluated. According to the law, these are:

!(α Blng ConnSt*(Hl))*

where ConnSt*(Hl)* = ConnSt*(a0:= <>in)* = {in},

then *!(c0* Blng {in} *)* = true

and *!(*α Blng ConnSt*(Hr))*

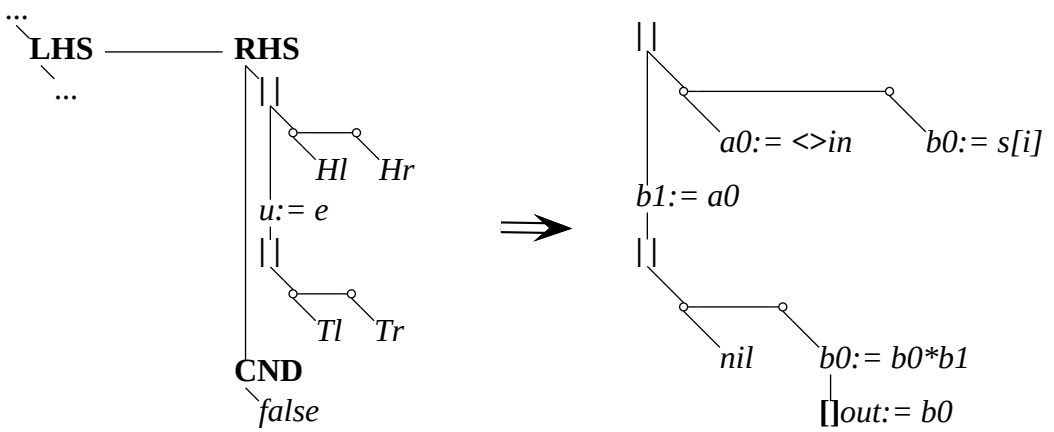
where ConnSt*(Hr)* = ConnSt*(b0:= s[i])* = $\varnothing$, the expression becomes true

Since both applicability conditions are true, the law can be applied to S_2.

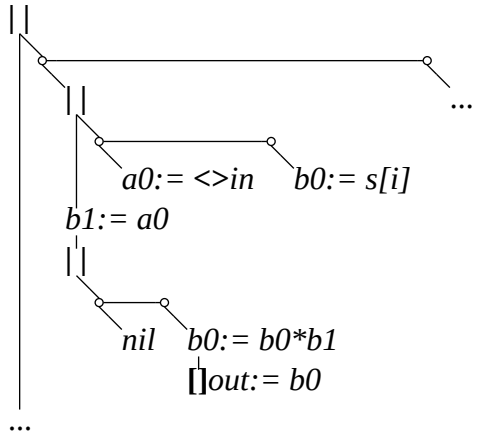
5.3.3.4 Transformation

This is the last step of procedure apply. It is carried out if the applicability conditions are satisfied. The input statement is transformed by substituting the matching tree by the schema of the other side of the law, where the variables for matching are replaced by their corresponding trees within list r.

In the above example, the equivalences of variables u, e, Hl, Hr, Tl and Tr have been obtained . The below transformation example uses them. The final form of S_2', which is the result of applying the law to S_2, is shown at the right.



Continuing with the example, the transformation procedure basically copies the RHS subtree of the law, and replaces the variables for matching. Then, the obtained tree is connected at the applicability point of S_2, after cutting and removing the original subtree. S_2' has the form:



5.4 The Transformation Procedures

5.4.1 Introduction

This section presents the transformation procedures, also called reduction procedures in particular cases. Their goal is the transformation of the input statement applying only the laws and lemmas described in 2.9. These guarantee that the transformation always corresponds to the application of a sequence of laws.

The motivation for implementing a set of transformation procedures is to go beyond the simplicity and the limitations of procedure APPLY, by making the prover a more powerful tool. Some of their advantages are the following:

- Avoiding some tedious parts of the verification, since they can be encapsulated in transformation procedures. For example, the communication elimination procedure, to automate some of its preliminary steps, which apply basic laws, before proper elimination.

- Allowing the iterative application of some laws. For instance, those that carry out simplifications or reductions, such as the simple arithmetical expression simplification. The basic implementation would be: given a law, the transformation searches the first applicability point within the input statement and applies it, then the procedure repeats again these two steps until no more applicability points are found.

- Complex applicability conditions can be integrated easily in the transformation procedures, bypassing the limitations of the grammar presented in subsection 5.3.3.2. For example, communication elimination needs to verify a huge set of conditions before applying the elimination law as seen in chapter 4.

- Bypassing the need of introducing of an infinity of laws in some situations. For instance, in the case of concatenation associativity, law 3 of page 38. This law has infinite possible forms depending on the values of k, l, and n. Since procedure APPLY is based on pattern matching, a repository with infinite representations of the concatenation associativity law would be needed, this would be neither practical nor possible. Therefore implementing the law as a transformation procedure is the approach taken in this work.

In the last situation, laws are not applied directly. The correctness of the transformation procedure guarantees that the same transformation could be obtained by direct application to the input statement of a sequence of laws. The following subsections detail the transformation procedures implemented and embedded into the prover tool.

5.4.2 Repository of Transformation Procedures

5.4.2.1 Communication Elimination

This is an iterative procedure that removes matching pairs of communication substatements from the input statement. Since it is one of the main topics of this work, it was explained in detail in chapter 4, after introducing the concepts in chapter 3. The elimination implies complex applicability conditions, these are coded within this transformation procedure. However the transformation is done by direct application of laws.

5.4.2.2 Cooperation Permutation

The cooperation permutation transformation procedure is based on law 7 of page 39:

$$S_1||\cdots||S_n \;\approx\; S_{p_m(1)}||\cdots||S_{p_m(n)}$$

where $p_m(k)$, for $k = 1..n$, denotes the *k-th* integer of a permutation of the list $\langle 1, 2, \ldots, n\rangle$.

The law has infinite possible forms, thus the user must indicate which one he wants to apply and the applicability point within the input statement. The interface prompts the user to enter the desired permutation, a list of numbers separated by commas. The numbers correspond to the final position of each statement S_x within the output cooperation composition.

The procedure validates that a parallelism symbol is at the applicability point, and the permutation list, p_m, by verifying the following conditions:

1. Numbers within the list are not repeated.
2. The length of the list must equal the number of parallel statements of the input cooperation statement.
3. The list must contain numbers in the interval $[1..n]$.

If the conditions are not fulfilled, the procedure warns the user to enter again the list. This verification process is done via the prover tool interface.

The procedure, shown below, has as inputs the statement S to be transformed, a valid permutation list from the interface, and the applicability point within S. S' is the output, and no indication of failure is needed since a congruence is applied and any valid permutation is allowed.

Procedure COOPPERMUT − cooperation permutation

Input: S is the input statement in expanded PADD notation, p is the applicability point within S, and p_m is the permutation list.
Output: S' a program equivalent to S, transformed as in the above law.

$S' := S$
$S_{temp} :=$ p within S'
$n :=$ NUMBERCOOPSTATEMENT(S_{temp})
for $i := 1$ ***to*** n **do**
  $a(i) :=$ GETCOOPSTATEMENT(S_{temp}, i)
$S_{temp} :=$ REMOVECOOPSTATEMENTS(S_{temp})
for $i := 1$ ***to*** n **do**
  $S_{temp} :=$ APPENDCOOPSTATEMENT(S_{temp}, $a(p_m(i))$)
p within $S' := S_{temp}$

After the second assignment, S_{temp} equals the subtree at p within S'. The top statement of S_{temp} must be a cooperation composition, and its root node be a parallel symbol. These are preconditions guaranteed by the prover interface. Several basic cooperation procedures are applied to obtain the output equivalent program. These are the following:

- NUMBERCOOPSTATEMENT(S): returns the number of parallel substatements of the top cooperation composition substatement within statement S.
- GETCOOPSTATEMENT(S, i): returns the i-th parallel substatement of the top cooperation substatement within S. From now on, the parallel processes are counted from the left.
- REMOVECOOPSTATEMENTS(S): deletes all the cooperation substatements from the top cooperation substatement within S, and returns the resulting statement.
- APPENDCOOPSTATEMENT(S, t): appends the parallel substatement t at the right end of the top cooperation substatement within S, and returns the resulting one.

The transformation is applied in two steps. First, all parallel substatements of the cooperation composition S_{temp} are stored into array a, and then they are removed from the cooperation, reducing it to an empty cooperation. The last step constructs the new cooperation by appending to the right, one by one and starting with an

empty parallel statement, the substatements according to the list, p_m, entered by the user, thus obtaining S' with the desired permutation.

The **computational complexity** is $O(n)$, since the procedure traverses at least twice the n parallel processes of the input statement S.

5.4.2.3 Cooperation Associativity

It applies law 8 of page 39, cooperation associativity:

$$[S_1||\cdots||S_k||\cdots||S_l||\cdots||S_n\ ] \approx [\ S_1||\cdots||\ [\ S_k||\cdots||S_l\ ]\ ||\cdots||S_n\ ]$$

The implementation is quite similar to the previous transformation procedure. The user is prompted to enter the interval of parallel statements to be associated. For example, the above sample law, associates S_k through S_l. Then the user should type k-l, where k and l are integers such that $1 \leq k < l \leq n$. The prover interface guarantees that a valid k-l interval is passed as parameter to the procedure, and that the applicability point indicated by the user corresponds to a parallelism symbol. Next procedure COOPASSO implements law 8.

Procedure COOPASSO − cooperation association

Input: S is the input statement in expanded PADD notation, p is the applicability point within S, and the naturals k and l, where $k < l \leq n$.
Output: S' a program equivalent to S, transformed as in the above law.

$S' := S$
$S_{temp} :=$ p within S'
for $i := k$ ***to*** l **do**
  $a(i-k) :=$ GETCOOPSTATEMENT(S_{temp}, i)
for $i := k$ ***to*** l **do**
  $S_{temp} :=$ DELCOOPSTATEMENT(S_{temp}, k)
$S_0 :=$ NEWCOOPSTATEMENT()
for $i := k$ ***to*** l **do**
  $S_0 :=$ APPENDCOOPSTATEMENT(S_0, $a(i-k)$)
$S_{temp} :=$ INSERTCOOPSTATEMENT(S_{temp}, S_0, k)
p within $S' := S_{temp}$

It uses three new basic cooperation procedures:

- DELCOOPSTATEMENT(S, i): deletes the i-th parallel substatement from the cooperation composition S, and returns the resulting statement. As a consequence, the number of parallel statements of S is decreased by one.

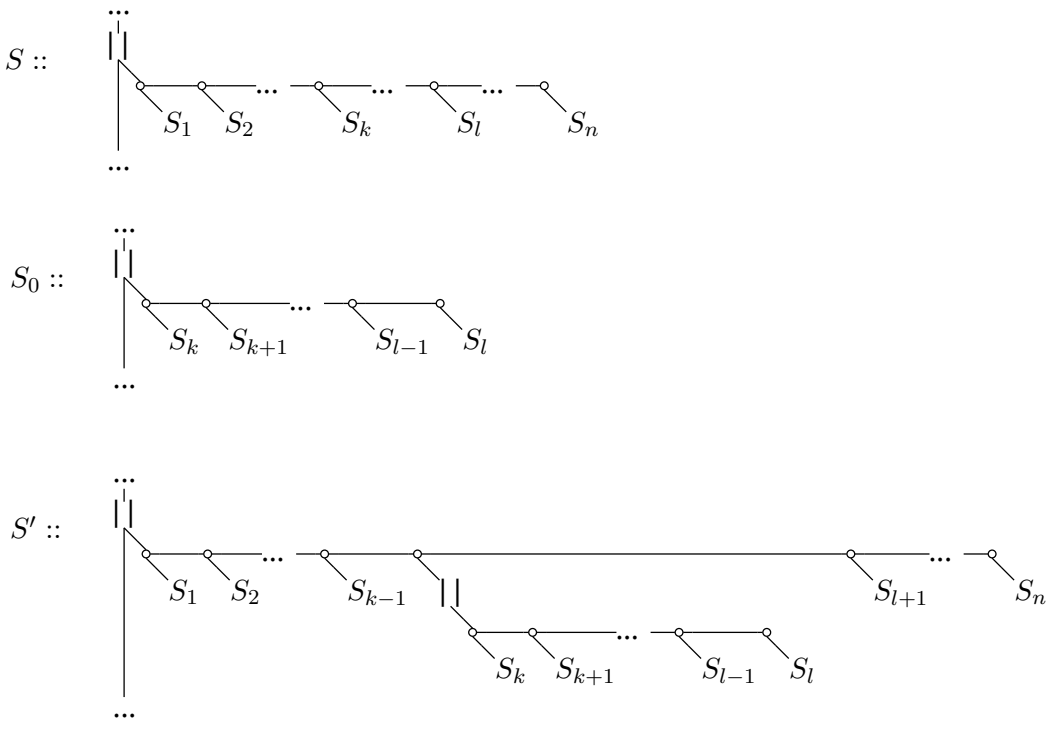
- NEWCOOPSTATEMENT(): returns an empty cooperation structure, which contains only a parallel symbol.

- INSERTCOOPSTATEMENT(S, t, i): inserts the parallel statement t at the i-th position of the top cooperation composition within S, and returns the resulting statement.

The procedure stores the parallel substatements involved in the association k-l, in array a, before deleting them. A new cooperation, S_0, is constructed with all the elements in a. The last step of the transformation inserts S_0 into the top cooperation statement, S_{temp}, at the k-th position.

The **computational complexity** is $O(n)$, since the procedure iterates at most three times through n, where n is the number of parallel processes of the input statement S.

As an example, the following schemas illustrate the input S, and output S', of the procedure, and the intermediate form S_0:



After the transformation, the output statement S' satisfies:

$$\text{NUMBERCOOPSTATEMENT}(S') := \\ \text{NUMBERCOOPSTATEMENT}(S) - \text{NUMBERCOOPSTATEMENT}(S_0) + 1$$

5.4.2.4 Binary Cooperation Associativity

This is a special case of the previous transformation. Usually this transformation is applied within the communication elimination transformation procedure, COMELI of page 102, as one of its pre-processing steps, as explained in chapter 4. Here the user is prompted to enter two integers: k, l. The interface validates that the applicability point is at a parallelism symbol, and the integers satisfy the inequality $1 \leq k < l \leq n$, otherwise an error message is prompted.

Basically it applies two laws: first cooperation commutativity

$$[S_1||\cdots||S_k||\cdots||S_l||\cdots||S_n\ ] \approx \\ [S_1||\cdots||S_{k-1}||S_k||S_l||S_{k+1}||\cdots||S_{l-1}||S_{l+1}||\cdots||S_n\ ]$$

and then cooperation associativity

$$[S_1||\cdots||S_{k-1}||S_k||S_l||S_{k+1}||\cdots||S_{l-1}||S_{l+1}||\cdots||S_n\ ] \\ \approx \\ [S_1||\cdots||S_{k-1}||\ [\ S_k||S_l\ ]\ ||S_{k+1}||\cdots||\cdots||S_n\ ]$$

The procedure is the following:

Procedure BINCOOPASSO − binary cooperation association

Input: S is the input statement in expanded PADD notation, p is the applicability point within S, and the integers k and l, where $1 \leq k < l \leq n$.

Output: S' a program equivalent to S, transformed as in the above laws.

$n :=$ NUMBERCOOPSTATEMENT(p within S)

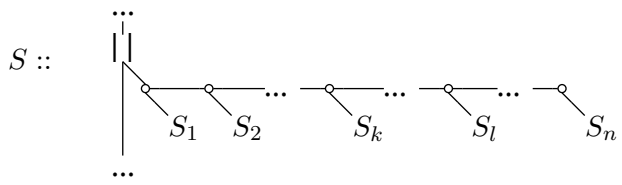
$p_m := \langle 1,\ \ldots,\ k-1,\ k,\ l,\ k+1,\ \ldots,\ l-1,\ l+1,\ \ldots,\ n\rangle$

$S' :=$COOPPERMUT(S, p, p_m)

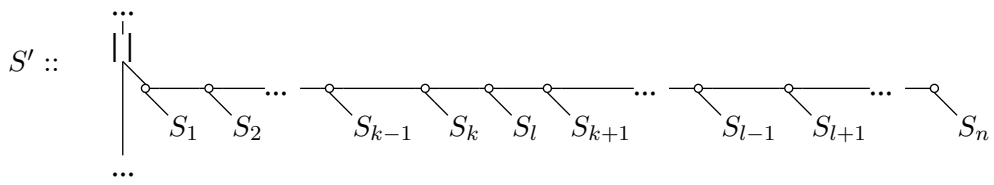
$S' :=$COOPASSO(S', p, k, $k+1$)

The procedure calculates the desired cooperation permutation list, p_m, according to parameters k and l, before calling transformation COOPPERMUT. Note that the parallel process l is moved to position $k+1$. Finally COOPASSO associates the two parallel processes labeled as k and l.

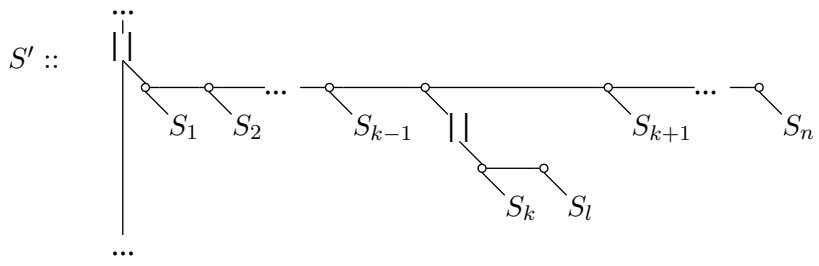
The following are the input, intermediate and output forms of procedure BINCOOPASSO:



After ending COOPPERMUT one obtains:



The output form is the following, after COOPASSO:



The procedure calls COOPPERMUT and COOPASSO. Since their complexities are $O(n)$, the **computational complexity** of BINCOOPASSO is $O(n)$.

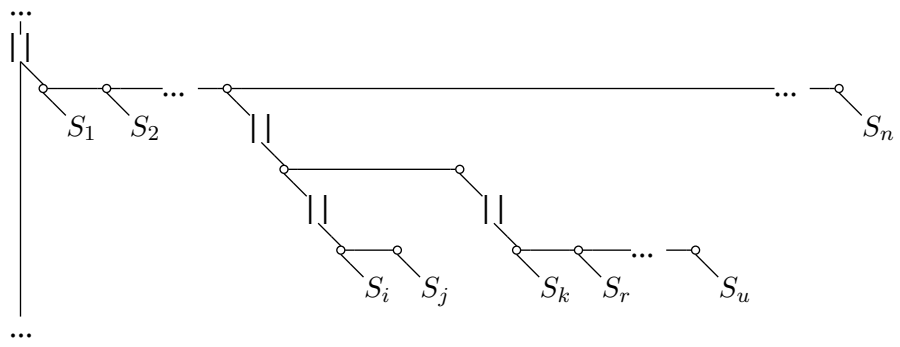
5.4.2.5 Cooperation Flattening

This transformation removes the cooperation associations found within a cooperation statement by applying from right to left law 8 and iteratively. In general, the statement can contain nested cooperation associations. The user must indicate the applicability point within the input statement. The interface validates that the point is at a parallel symbol, otherwise it prompts an error message.

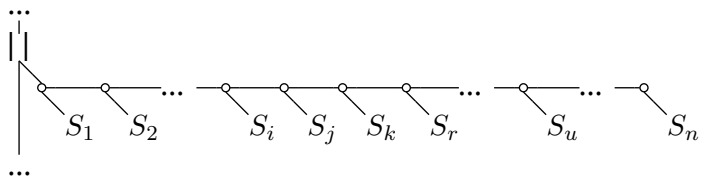
The procedure searches, within the input cooperation statement, for parallel subprocesses to be flattened. The goal is to obtain a statement without cooperation

associations.

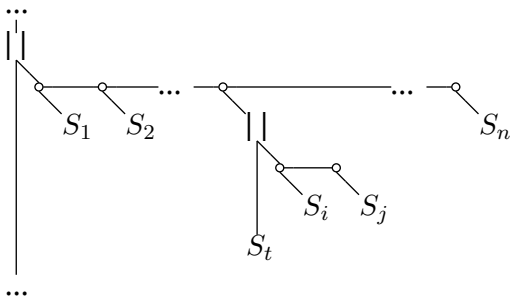
As an example, the following program contains several cooperation associations:



The procedure transforms the cooperation statement as follows:



Next example illustrates a situation where the flattening can not take place.

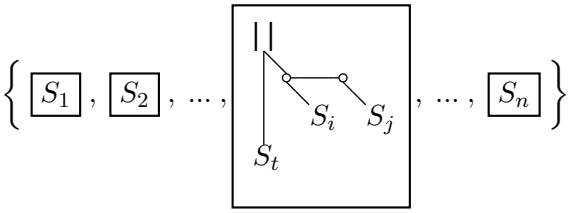


It has an inner nested parallelism concatenated with another statement.

The procedure, shown below, starts by identifying the parallel substatements of the top cooperation statement of S_{temp} that can be flattened. This is carried

out with a traversing of the parallelism tree structure by procedure GETCOOPSTATEMENTTOFLAT, which preserves any inner parallelism in concatenation with some other statements. The obtained substatements are stored into array a. The last step constructs the new cooperation by appending, one by one, the substatements of a.

For instance, if the last example is used as the input of the procedure, the substatements returned from GETCOOPSTATEMENTTOFLAT are the following:



Procedure COOPFLAT − cooperation flattening

Input: S is the input statement in expanded PADD notation, and p is the applicability point within S.
Output: S' a program equivalent to S, but flattened.

$S' := S$
$S_{temp} := p$ within S'
$a :=$ GETCOOPSTATEMENTTOFLAT(S_{temp})
$S_{temp} :=$ REMOVECOOPSTATEMENTS(S_{temp})
for $i := 1$ ***to*** $length(a)$ **do**
  $S_{temp} :=$ APPENDCOOPSTATEMENT(S_{temp}, $a(i)$)
p within $S' := S_{temp}$

The procedure calls GETCOOPSTATEMENTTOFLAT which traverses S_{temp}, then its computational complexity depends on the number of statements, st, of the input statement, and the **computational complexity** of COOPFLAT is $O(st)$.

5.4.2.6 Concatenation Permutation

The transformation procedure applies law 4:

$$S_1;\ \cdots;\ S_n\ \approx\ S_{p_m(1)};\ \cdots;\ S_{p_m(n)}$$

where $p_m(k)$, for $k = 1..n$, denotes the *k-th* integer of a permutation of the list $\langle 1, 2, \ldots, n \rangle$.

The transformation has applicability conditions: the statements S_1 , ... , S_n must be disjoint and have no communication statements.

The user indicates the desired permutation by entering a list of numbers separated by commas. The permutation list must satisfy the following:

1. Numbers within the list are not repeated.
2. The length of the list must equal the number of sequential statements of the input concatenation statement.
3. The list must contain numbers in the interval $[1..n]$.

The interface checks that the applicability point indicated by the user is at a concatenation with at least n substatements after it in the concatenation, otherwise it prompts an error message.

The procedure is the following:

Procedure CONCATPERMUT − concatenation permutation

Input: S is the input statement in expanded PADD notation, p is the applicability point within S, p_m is the permutation list, and n the number of concatenation statements involved in the permutation.

Output: S' a program equivalent to S, but transformed as in the above law, or a failure indication.

$S' := S$
$S_{temp} := p$ within S'
for $i := 1$ ***to*** n **do**
  $a(i)$:= GETCONCATSTATEMENT(S_{temp}, i)
if DISJOINTNOCOMM*(a, n)* **then**
  for $i := 1$ ***to*** n **do**
    S_{temp} := REPLACECONCATSTATEMENT(S_{temp}, $a(p_m(i))$, i)
else
  exit with failure, statements within a are not disjoint and/or communicate
p within $S' := S_{temp}$

Some new concatenation procedures are used. These are the following:

- GETCONCATSTATEMENT(S, i): returns the *i-th* concatenated substatement of S.

- DISJOINTNOCOMM(a, n): returns *true* if the substatements $a(1), a(2), \ldots, a(n)$ are disjoint and they do not contain any communication statement.

  - They are disjoint if $VarPSt(a(i)) \cap VarRSt(a(j)) = \emptyset$, and $VarRSt(a(i)) \cap VarRSt(a(j)) = \emptyset$, for $i, j := 1..n$ and $i \neq j$,

    $VarPSt$ is the set of all read (parameter) variables within $a(i)$, and $VarRSt$ is the set of all written (result) variables within $a(j)$.

  - They do not have communication statements if there exists at most one $a(i)$ such that $ConnSt(a(i)) \neq \emptyset$, for $1 \leq i \leq n$.

    $ConnSt$ is the set of all communication statements within $a(i)$.

- REPLACECONCATSTATEMENT(S, t, i): the *i-th* concatenation substatement of S is replaced by substatement t, and returns the resulting statement.

The procedure stores the concatenation substatements into array a, and procedure DISJOINTNOCOMM checks whether they are disjoint and have no communication statements. If the applicability conditions are satisfied then the concatenation substatements are replaced according to the permutation list p_m.

Procedure DISJOINTNOCOMM traverses n times the n sequential processes of the input statement S, then **computational complexity** of CONCATPERMUT is $O(n^2)$.

5.4.2.7 Concatenation Association

The transformation procedure applies law 3 from left to right:

$$S_1; \cdots; S_k; \cdots; S_l; \cdots; S_n \;\approx\; S_1; \cdots; [S_k; \cdots; S_l]; \cdots; S_n$$

where k and l are integers such that $1 \leq k < l \leq n$.

The interface prompts the user to enter the two integers k-l which must satisfy the above inequality. The applicability point, indicated by the user, must be a concatenation with at least l substatements after it in the concatenation, otherwise it prompts an error message.

The procedure associates the sequential substatements by leaving them under the scope of a PADD comment, see subsection 2.1.4.

The implementation is as follows:

Procedure CONCATASSO − concatenation association

Input: S is the input statement in expanded PADD notation, p is the applicability point within S, and the naturals k and l, where $k < l \leq n$.
Output: S' a program equivalent to S, transformed as in the above law.

$S' := S$
$S_{temp} :=$ p within S'
for $i := k$ ***to*** l **do**
  $a(i) :=$ GETCONCATSTATEMENT(S_{temp}, i)
for $i := k$ ***to*** l **do**
  $S_{temp} :=$ DELCONCATSTATEMENT(S_{temp}, k)
$S_0 :=$ NEWCONCATSTATEMENT()
for $i := k$ ***to*** l **do**
  $S_0 :=$ APPENDCONCATSTATEMENT(S_0, $a(i-k)$)
$S_{temp} :=$ INSERTCOMMENTSTATEMENT(S_{temp}, k)
$S_{temp} :=$ REPLACEDIAGONAL(S_{temp}, S_0, k)
p within $S' := S_{temp}$

Some new concatenation procedures are needed:

- DELCONCATSTATEMENT(S, i): deletes the *i-th* concatenated substatement of S, and returns the resulting statement. As a consequence, the number of sequential statements is decreased by one.

- NEWCONCATSTATEMENT(): returns an empty concatenation structure.

- APPENDCONCATSTATEMENT(S, t): appends the substatement t at the end of the concatenation statement S, and returns the resulting one.

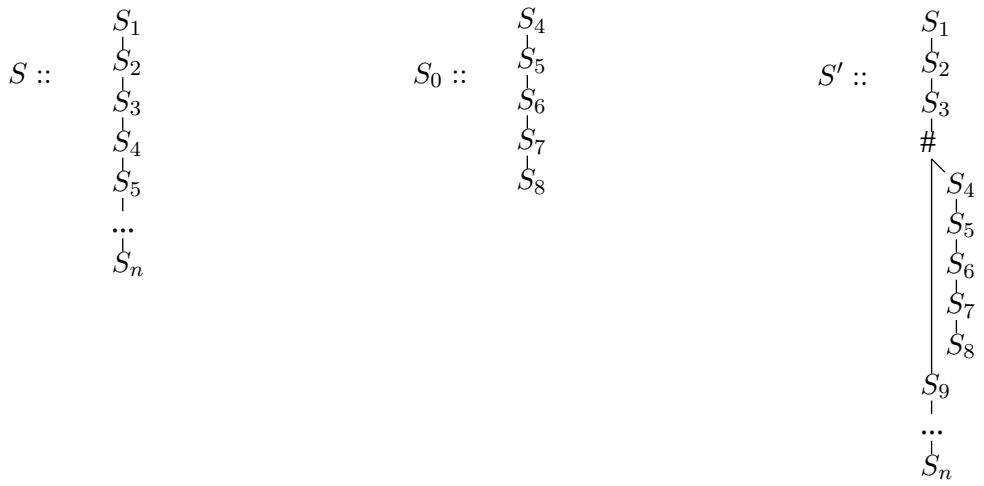
- INSERTCOMMENTSTATEMENT(S, i): inserts a comment, a node with the symbol #, at the *i-th* concatenated substatement of S, and returns the resulting one.

- REPLACEDIAGONAL(S, t, i): changes the diagonal subtree of the *i-th* substatement of S to substatement t, and returns the resulting one.

The substatements to be associated, k and l, are stored in array a before deleting them. A new concatenation, S_0, is constructed with all the elements in a. S_0 is

connected as the diagonal subtree of the comment node, which is inserted at *k-th* position by INSERTCOMMENTSTATEMENT.

As an example, the following schemas illustrate the input, S, and output, S', of the procedure, and the intermediate form S_0; where $k = 4$ and $l = 8$:



In S', observe that concatenation S_0 is placed at the diagonal of the inserted comment symbol.

The **computational complexity** is $O(n)$, since the procedure iterates at most three times through n, where n is the number of sequential processes of the input statement S.

5.4.2.8 Concatenation Flattening

This transformation removes the concatenation associations found within the input sequential statement with associations by applying iteratively the following special case of law 3:

$$[S_1; S_2; S_3; \cdots]; \cdots; S_n \;\approx\; S_1; S_2; S_3; \cdots; S_n$$

with $k = 1$ and l is such that $1 < l \leq n$.

The iterative procedure that removes comment associations is the following:

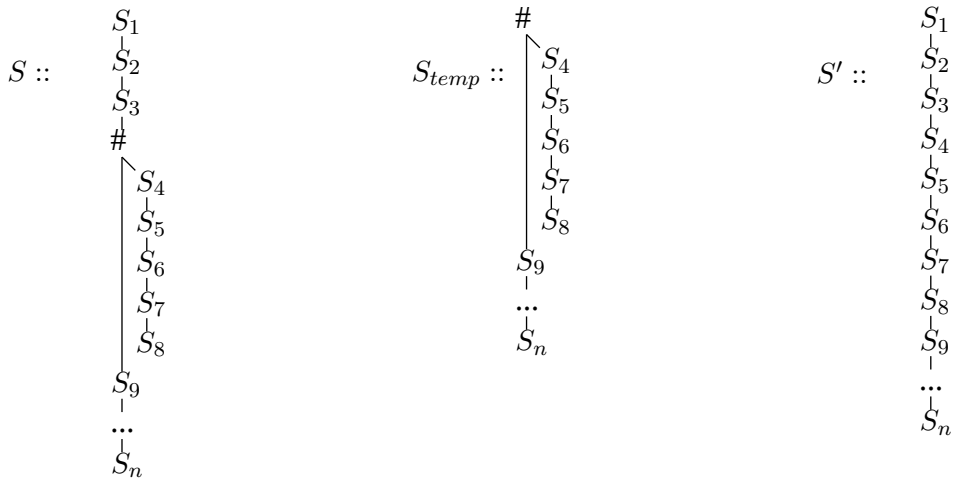
Procedure CONCATFLAT – iterative concatenation flattening

Input: S is the input statement in expanded PADD notation.
Output: S' a program equivalent to S, transformed as in the above law.

$S' := S$
foreach *concatenation association within* S' **do**
  $p :=$ points at the comment node found
  $S_{temp} := p$ within S'
  apply the law $\left\{ [S_1; S_2; S_3; \cdots]; \cdots; S_n \;\approx\; S_1; S_2; S_3; \cdots; S_n \right\}$ to S_{temp}
  p within $S' := S_{temp}$

The procedure, shown above, traverses the input statement searching for comment nodes. For each one found, an applicability point p is obtained, and the above law is applied.

As an example, the following shows the input S with just one concatenation association, the intermediate form obtained, S_{temp}, and the output S':



The procedure traverses all the substatements of the input statement, then the **computational complexity** depends on the number of statements, $O(st)$.

5.4.2.9 Cooperation and Concatenation

The transformation procedure applies law 11 of page 39:

$$S_1||\cdots||S_n \quad =_{\mathcal{O}} \quad S_{p_m(1)};\ \cdots\ ;S_{p_m(n)}$$

where $p_m(k)$, for $k = 1..n$, denotes the *k-th* integer of a permutation of the list $\langle 1, 2, \ldots, n\rangle$. As applicability conditions, statements S_k must be disjoint and have no communication statements.

The procedure is the following:

Procedure COOPCONCAT − cooperation and concatenation

Input: S is the input statement in expanded PADD notation, p is the applicability point within S, p_m is the permutation list, and n the number of statements involved in the permutation.

Output: S' a program equivalent to S, transformed as in the above lemma, or a failure indication.

$S' := S$
$S_{temp} :=$ p within S'
for $i := 1$ ***to*** n **do**
  $a(i) :=$ GETCONCATSTATEMENT(S_{temp}, i)
if CHECKCOOPCONCATAPPCOND*(a, n)* **then**
  $S_{temp} :=$ REMOVECOOPERATION(S_{temp})
  $S_0 :=$ NEWCONCATSTATEMENT()
  for $i := 1$ ***to*** n **do**
    $S_0 :=$ APPENDCONCATSTATEMENT(S_0, $a(p_m(i))$)
  $S_{temp} :=$ INSERTCONCATSTATEMENT(S_{temp}, S_0), 1)
else
  exit with failure, statements within array a are not disjoint and/or communicate
p within $S' := S_{temp}$

The interface prompts the user to enter the permutation, a list of numbers separated by commas. The numbers correspond to the final position of each statement S_x in the concatenation order of the output statement. The procedure validates that a parallelism symbol is at the applicability point, and the permutation list, *PermutList*, by verifying the following conditions:

1. Numbers within the list are not repeated.
2. The length of the list must equal the number of parallel statements of the input cooperation statement.
3. The list must contain numbers in the interval $[1..n]$.

If they are not fulfilled, the interface warns the user to enter again the list.

The transformation is similar to the other permutations presented in this section. The procedure extracts the parallel processes from the top cooperation of S_{temp}. CHECKCOOPCONCATAPPCOND procedure, detailed below, verifies whether they satisfy the applicability conditions. Finally, after removing the cooperation statement, procedure REMOVECOOPERATION, the procedure constructs the new concatenation S_0, which is inserted at the top of S_{temp}.

Some new functions are used in the above procedure COOPCONCAT:

- CHECKCOOPCONCATAPPCOND(a, n): checks whether the n substatements of a are disjoint and have no communication statements.
  - They are disjoint if $VarPSt(a(i)) \cap VarRSt(a(j)) = \emptyset$, and $VarRSt(a(i)) \cap VarRSt(a(j)) = \emptyset$, for $i, j := 1..n$ and $i \neq j$,

    where $VarPSt$ is the set of all read variables within $a(i)$, and $VarRSt$ is the set of all written variables within $a(j)$.
  - They do not have communication statements if there exists at most one $a(i)$ such that $ConnSt(a(i)) \neq \emptyset$, for $1 \leq i \leq n$.

    $ConnSt$ is the set of all communication statements within its argument.
- REMOVECOOPERATION(S): deletes the top cooperation statement within S, and returns the resulting statement.
- INSERTCONCATSTATEMENT(S, t, i): inserts the sequential substatement t at the *i-th* position, in the concatenation order, of S, and returns the resulting statement.

The **computational complexity** is $O(n^2)$. This upper bound is determined by procedure CHECKCOOPCONCATAPPCOND, which checks the applicability conditions of the TP.

5.4.2.10 Elimination of Redundant Variables

The variable elimination procedure applies law 13 of page 40,

$$[\ v := e; S_1(v); S_2\ ] \ =_{\mathcal{O}} \ [\ S_1(e); S_2\ ]$$

law 17 of page 41,

$$[\ (\bar{v}) := (\bar{e}); S_1(\bar{v}); S_2\ ] \ =_{\mathcal{O}} \ [\ S_1(\bar{e}); S_2\ ]$$

and law 18 of page 41,

$$\begin{array}{c} [\ (v.v_1, ..., v.v_i, ..., v.v_j, ..., v.v_n) := (e_1, ..., e_i, ..., e_j, ..., e_n);\ S_1(v.v_i, ..., v.v_j);\ S_2\ ] \\ =_{\mathcal{O}} \\ [\ (v.v_1, ..., v.v_n) := (e_1, ..., e_n);\ S_1(e_i, ..., e_j);\ S_2\ ] \end{array}$$

where $\bar{e}$ is a list of n expressions, and $\bar{v}$ is a list of n variables.

Procedure VARELIM tries to remove the redundant variable assignment, indicated by the user, from the input statement S by applying one of the above lemmas. The applicability conditions detailed in the lemmas are checked and must be fulfilled before removing the variable assignment. S_2 is tried to be identified, for certain simple special forms. When this identification does not succeed, the procedure exits with failure. More general forms for S_2 will be included in future versions of the procedure. Once S_2 has been identified, S_1 is established. In the last step of procedure VARELIM, the read variables within S_1 are replaced by their corresponding expressions, as the above lemmas establish.

5.5 Conclusion

The design of an interactive prover to help in the construction of sequentialization proofs has been treated in this chapter. Being developed for mechanizing and partially automating these equivalence proofs, it works from an input program or statement which is transformed guaranteeing that only laws and transformation procedures are applied.

Having in mind a final goal form, the user has to guide the prover, via commands, in order to obtain the goal. An interface *command* selects a law or a transformation procedure, from two corresponding repositories, for application. In general, a command specifies the application point within the current program, or statement, which is the result of a sequence of commands applied to the input and to its transformation successors up to the present state.

Any basic law transformation is carried out by the *apply* procedure. This checks the applicability conditions of the law and, when they are satisfied, performs structure matching at a specified application point followed by the statement transformation. The substitution of procedure references by their bodies is another elementary transformation at the same basic level as *apply*.

At a second complexity level, the prover has *transformation procedures*, TPs, carrying out simple transformations such as parallelism permutation and flattening; concatenation permutation, association, and flattening; elimination of redundant assignments and variables, etc. A transformation procedure guarantees that the output or resulting program can be obtained from its input statement by a sequence of law applications, as reductions. In most of the cases the transformation is done via a sequence of *apply* invocations.

More complex transformation procedures form a *third layer*. For instance for the iterative application of TPs of the second layer. Also, more complex TPs such as COMELI for the automatic elimination of communication pairs, which was covered in chapter 4, belong to the third layer.

The chapter has gone into much detail about the internal data structures used to represent trees, for structure matching, representation of laws, etc. These have been inspired by the dimensional flowcharting trees used in PADD; mostly for historical reasons. The input statement is transformed into internal structure before any transformation takes place. All this detail will be useful for those that may further develop the tool.

The prover has an expandable design, admitting more TPs, being defined in terms of existing ones, *apply*, and by other means. Further work on the automatic construction of more parts of DPS proofs, such as TPs combining parallelism to concatenation transformations with redundant variable elimination should be carried out. This is outlined as future work in chapter 7.

Chapter 6

CORRECTNESS PROOF OF A PIPELINED PROCESSOR ARCHITECTURE

An equivalence proof of a pipelined processor model is presented. The first sections detail the proposed architecture that performs only register to register instructions. However, the complexity of a level of forwarding circuits is included. The model, $Pipeline_2$, is hierarchical with two levels of parallelism.

Last section describes with great detail the proof, which establishes interface equivalence between the pipeline processor model, with parallelism and inner communications, $Pipeline_2$, and a simple purely sequential model, a *Von Neumann* processor loop. The set of registers is the interface set $\mathcal{O}$ of the equivalence. Thus for a given program and initial state of the registers both models give the same register values as a result. This is very intuitive: a quite complex parallel architecture is formally equivalent to a simple typical processor loop. This is precisely the function it should perform. Furthermore, due to the total elimination of inner communications, deadlock freeness is established.

6.1 Introduction to the DLX Processor

A fundamental pipeline architecture is the DLX processor [HP90]. The DLX is a relatively simple RISC-type architecture. It features a minimal instruction set, relatively few addressing modes, and a processor organization designed to simplify implementation. The pipelined processor hardware is organized in parallel stages. These stages are the Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). Any instruction read from the memory is executed in steps as it traverses the stages of the pipeline, also data flows between the parallel stages.

The DLX is a 32-bit word-oriented system. The CPU contains a 32-bit ALU and 32 general registers organized in a register file. The registers are part of the Decode stage. There are four pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB), between every two consecutive stages to temporarily store the intermediate results and/or information to be used in later stages. The DLX processor executes each instruction in a number of physical steps, the pipeline stages. Because the pipe stages are hooked together, all the stages must be ready to proceed at the same time. The duration of a pipeline stage corresponds to one machine clock period.

The following diagram, figure 6.1, illustrates the basis of the pipeline architecture. It is a very simple and regular design. Latches are the pipeline registers. It is easy to observe how the instruction flows from the Fetch stage to the Write Back stage.

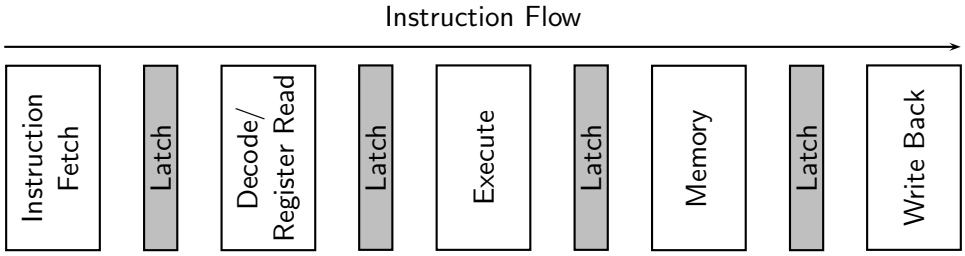


Figure 6.1: DLX stages and pipeline registers

DLX 32-bit instructions come in three formats: R-type, I-type, and J-type. All instruction formats must specify an *opcode*; however, the other information in the instruction varies by format. R-type (register) instructions specify three registers in the instruction - two source registers and one destination register. I-type (immediate) instructions specify one source register, one destination register, and a 16-bit immediate value that is sign-extended to 32 bits before it is used. J-type (jump) instructions consist of only the *opcode* and a 26 bit *operand*, which is used to calculate the destination address.

The three instruction formats are summarized in this table:

Format	Fields					Comments
	6 bits	5 bits	5 bits	5 bits	11 bits	
R-type	*opcode*	*rs1*	*rs2*	*rd*	*function*	Arithmetic instruction format
I-type	*opcode*	*rs1*	*rd*	*immediate*		Transfer, branch, imm, format
J-type	*opcode*	*offset*				Jump instruction format

There are several types of hazards that can stall the pipeline architecture. For example, in the following two successive instructions:

$$\begin{array}{lll} \text{LW} & R_1, \text{A} & (R_1 \leftarrow A) \\ \text{ADD} & R_3, R_1, R_2 & (R_3 \leftarrow R_1 + R_2) \end{array}$$

The ADD instruction has to wait, in the ID stage, for the data of the LW instruction to be written into R_1. This is known as a read-after-write (RAW) dependency. In order to minimize this wait, special *data forwarding* buses are introduced in the architecture. Data is forwarded whenever it is available back to EX to minimize the stalls. In the above example, data would be available when LW terminates the MEM stage.

6.2 Simplified DLX-like Model

6.2.1 Global View

The proof is applied to a simplified DLX-like pipeline processor architecture, which is a software model, not a hardware design. It features only an ALU register-register instruction set. Tables 6.1 and 6.2 give a summary of the R-Type instructions. As commented above, they are register-register instructions, either arithmetical or logical. They are executed by the ALU.

Instruction	Format	Description
add	`ADD RD, RS1, RS2`	Add the contents of registers `RS1` and `RS2`, and the result is placed in register `RD`
subtract	`SUB RD, RS1, RS2`	Subtract the contents of register `RS2` from `RS1`, and the result is placed in register `RD`
add unsigned	`ADDU RD, RS1, RS2`	Like `ADD`, but assumes unsigned values
subtract unsigned	`SUBU RD, RS1, RS2`	Like `SUB`, but assumes unsigned values

Table 6.1: $Pipeline_2$ R-Type instructions. Arithmetical operations

Instruction	Format	Description
and	`AND RD, RS1, RS2`	Performs a logical AND operation
or	`OR RD, RS1, RS2`	Performs a logical OR operation
shift left logical	`SLL RD, RS1, RS2`	The contents of register `RS1` is shifted left (and zero-filled) by the number found in register `RS2`, and the result is placed in register `RD`
shift right logical	`SRL RD, RS1, RS2`	Like `SLL`, but performs a right logical shift

set less than	`SLT RD, RS1, RS2`	`if (RS1 < RS2)`
		`RD ← 1 else`
		`RD ← 0`

set less than unsigned	`SLTU RD, RS1, RS2`	Like `SLL`, but assumes unsigned values

Table 6.2: *Pipeline*$_2$ R-Type instructions. Logical operations

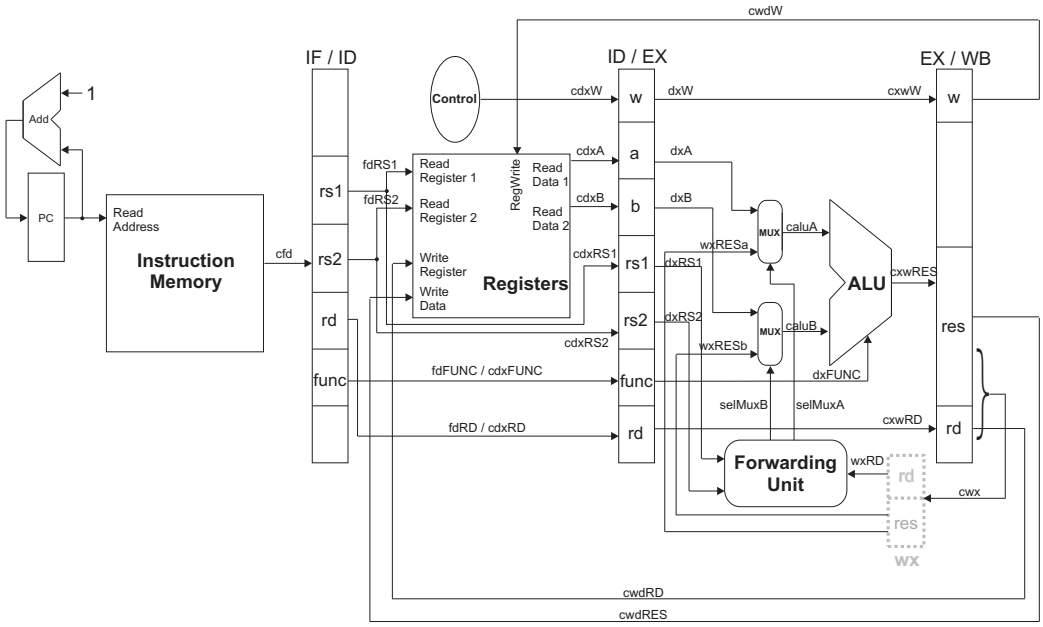


Figure 6.2: *Pipeline*$_2$ block diagram

The model does not have the Memory Access (MEM) stage, it only performs register-register operations. The pipeline model is a parallel composition of the following four stages:

$$Pipeline_2 :: \left[IF \;||\; ID_{par} \;||\; EX_{par} \;||\; WB_{unh}\right]$$

Figure 6.2 shows a block diagram of the DLX-like architecture.

This design is a hierarchical model which has two levels of parallelism. The first level is the four pipeline stages, and the second one corresponds to the intermediate stages, ID_{par} and EX_{par}, these are also implemented with inner parallel composition. The suffix $_{par}$ indicates a design with inner parallelism.

The architecture assumes synchronous communication channels between the stages. Each stage is implemented as a procedure. Next subsection will explain in detail each procedure of the $Pipeline_2$ model.

Observe that in the $Pipeline_2$ model the *opcode* field, used to distinguish between instruction types, has been eliminated because in all R-Type instructions it is equal to 0. The Instruction Decode stage always manipulates the same instruction format, and also the same fields: *rs1* (source register 1), *rs2* (source register 2), *rd* (destination register), *func* (function). The function field encodes the ALU operation.

6.2.2 Forwarding Unit

The DLX-like model provides a bypassing logic hardware. The stages ID, EX, and WB are involved in this mechanism.

The *forwarding unit* will solve the *Read-after-Write* (RAW) dependencies. These take place when an instruction reads a register value before it has been updated by the previous instruction.

As an example, consider the code:

$$\begin{array}{lll} \text{ADD} & R_1, R_2, R_3 & (R_1 \leftarrow R_2 + R_3) \\ \text{SUB} & R_5, R_1, R_6 & (R_5 \leftarrow R_1 - R_6) \end{array}$$

where R_1 causes a RAW dependency. The SUB instruction tries to read R_1 in the ID stage before it has been written by the ADD instruction. Without forwarding, the DLX pipeline would have to wait until the WB stage has written the value back.

Figure 6.3 shows the forwarding bypass paths. The forwarding unit checks the dependencies between the source register indexes of the current instruction and the destination register index of the previous instruction, and it calculates the multiplexor selector controls to input to the ALU the appropriates values. This is done

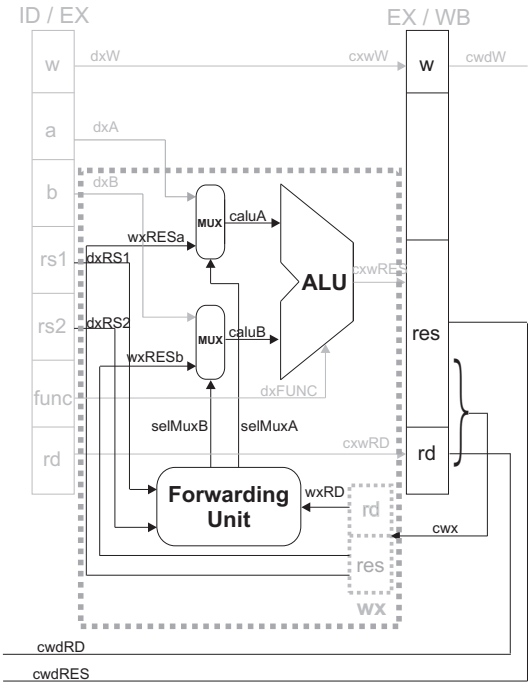


Figure 6.3: Forwarding mechanism

in this stage because the needed value is in the WB stage and instructions do not need their operand values until the beginning of their EX stage.

6.2.3 The $Pipeline_2$ Model

Each stage is modeled as a procedure. Procedures use some common data types. These data types and the procedures are given next.

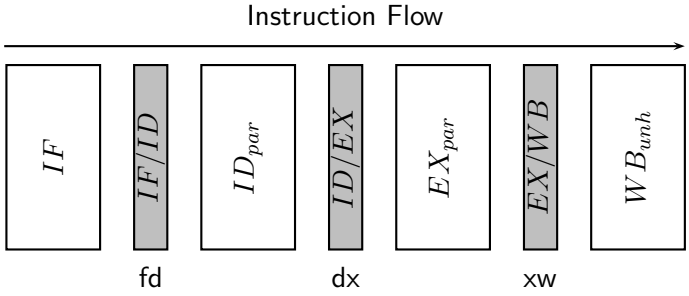


Figure 6.4: $Pipeline_2$ stages

Pipelining the datapath requires that values passed from one pipe stage to the next must be placed in the pipeline registers. These are fd, dx, and xw.

6.2.3.1 Data Types

This subsection gives the data types to be used in our model. These are the following:

Typ_IR: (rs1: bit5, rs2: bit5, rd: bit5, func: bit11)
Typ_DX: (w:bool, a:int32, b:int32, rs1:bit5, rs2:bit5, func:bit11, rd:bit5)
Typ_XW: (w:bool, res:int32, rd:bit5)
Typ_WX: (res:int32, rd:bit5)
Typ_XX: (w:bool, rd:bit5)

where $bit5$ denotes five bits data type, $bit11$ eleven bits, and $int32$ a 32 bit integer.

6.2.3.2 Procedure IF

Fetches the instruction from memory, into the *instr* variable of type $Type\_IR$. It is sent to the IF/ID (fd) register, of ID stage, via the *cfd* channel of type Typ_IR. The SPL form of this procedure is the following:

$$cfd ::= IF(mem) :: \left[\begin{array}{l}\textbf{loop forever do} \\ \quad \left[\begin{array}{l} instr := mem(pc); \\ pc := pc + 1; \\ cfd \Leftarrow instr \end{array}\right]\end{array}\right]$$

Next table shows the variables of procedure IF, and their sizes.

Var	Bits
pc	32
	32

Var	Fields	Bits
instr	rs1	5
	rs2	5
	rd	5
	func	11
		26

Table 6.3: Local variables of procedure IF

The IF stage has $32 + 26 = 58$ bits.

6.2.3.3 Procedure ID_{par}

This is the parallel form of the ID stage, as illustrated in figure 6.5. It accesses the register file, $reg(\cdot)$, to read the registers. The values are passed to the EX stage via $cdxW$, $cdxA$, $cdxB$, $cdxRS1$, $cdxRS2$, $cdxFUNC$, and $cdxRD$ channels. Also the ALU result at the WB stage is written into the destination register at the ID stage. The next instruction is received via the *cfd* channel.

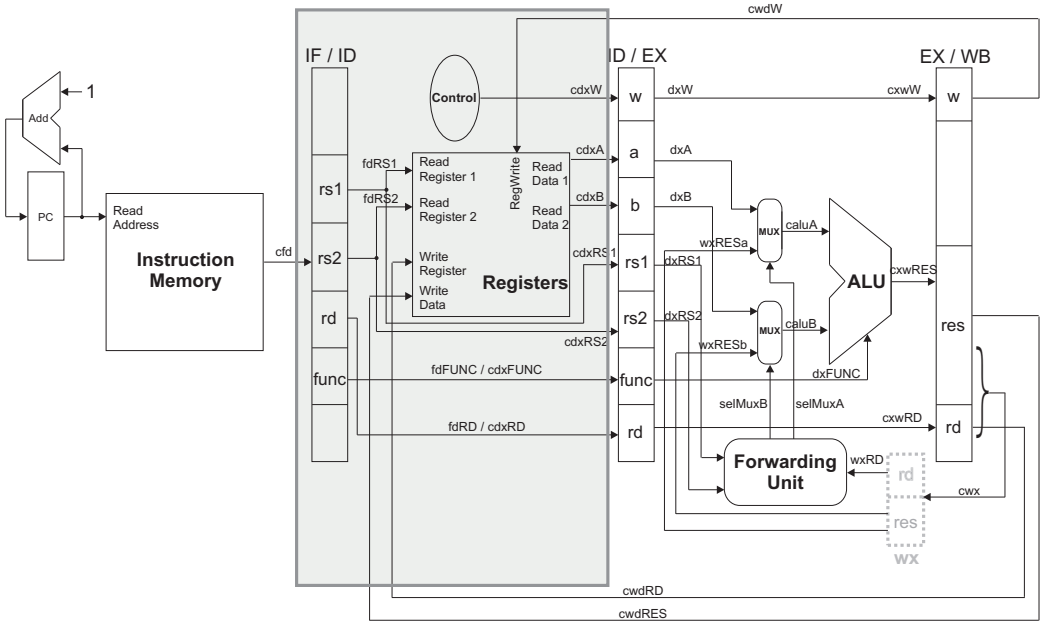


Figure 6.5: ID_{par} stage block diagram

In our model this stage is implemented in procedure ID_{par} shown below. The register file, reg, contains 32 registers of 32 bits. It is also a global variable of $Pipeline_2$, it is not a local variable of procedure ID_{par}.

$$(reg, cdxW, cdxA, cdxB, cdxRS1, cdxRS2, cdxFUNC, cdxRD) ::= \boxed{ID_{par}}(reg, cfd, cwdW, cwdRES, cwdRD) ::$$

$$\left[\begin{array}{l}
\left[\begin{array}{l}\textbf{loop forever do } (\ IF/ID\ (fd)\ register\ ) \\ \quad \left[\begin{array}{l} cfd \Rightarrow fd; \\ fdRS1 \Leftarrow fd.rs1; \\ fdRS2 \Leftarrow fd.rs2; \\ fdRD \Leftarrow fd.rd; \\ fdFUNC \Leftarrow fd.func \end{array}\right]\end{array}\right] \\
\| \\
\left[\begin{array}{l}\textbf{loop forever do } (\ Registers\ ) \\ \quad \left[\begin{array}{l} cwdW \Rightarrow wd.w; \\ cwdRES \Rightarrow wd.res; \\ cwdRD \Rightarrow wd.rd; \\ control \Rightarrow w; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (w, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ fdRS1 \Rightarrow ir.rs1; \\ fdRS2 \Rightarrow ir.rs2; \\ fdRD \Rightarrow ir.rd; \\ fdFUNC \Rightarrow ir.func; \\ cdxW \Leftarrow xdx.w; \\ cdxA \Leftarrow xdx.a; \\ cdxB \Leftarrow xdx.b; \\ cdxRS1 \Leftarrow xdx.rs1; \\ cdxRS2 \Leftarrow xdx.rs2; \\ cdxFUNC \Leftarrow xdx.func; \\ cdxRD \Leftarrow xdx.rd \end{array}\right]\end{array}\right] \\
\| \\
\left[\begin{array}{l}\textbf{loop forever do } (\ Control\ ) \\ \quad \left[\begin{array}{l} c := true; \\ control \Leftarrow c \end{array}\right]\end{array}\right]
\end{array}\right]$$

The local variables and their sizes are shown in the following table:

Registers

Var	Fields	Bits
wd	w	1
	res	32
	rd	5
		38

Var	Fields	Bits
ir	rs1	5
	rs2	5
	rd	5
	func	11
		26

Var	Fields	Bits
xdx	w	1
	a	32
	b	32
	rs1	5
	rs2	5
	func	11
	rd	5
		91

fd register

Var	Fields	Bits
fd	rs1	5
	rs2	5
	rd	5
	func	11
		26

Control

Var	Bits
c	1
	1

Table 6.4: Local variables of procedure ID_{par}

The ID stage has $26 + 38 + 26 + 91 + 1 = 182$ bits.

6.2.3.4 Procedure EX_{par}

This is the parallel form of the EX stage, figure 6.6. The new instruction data is received via $cdxA$ and $cdxB$ channels. The forwarding data comes from the WB stage through the auxiliary wx register. Register dependencies are checked in the *Forwarding Unit*, and sent to the ALU from the Mux selectors. The ALU performs

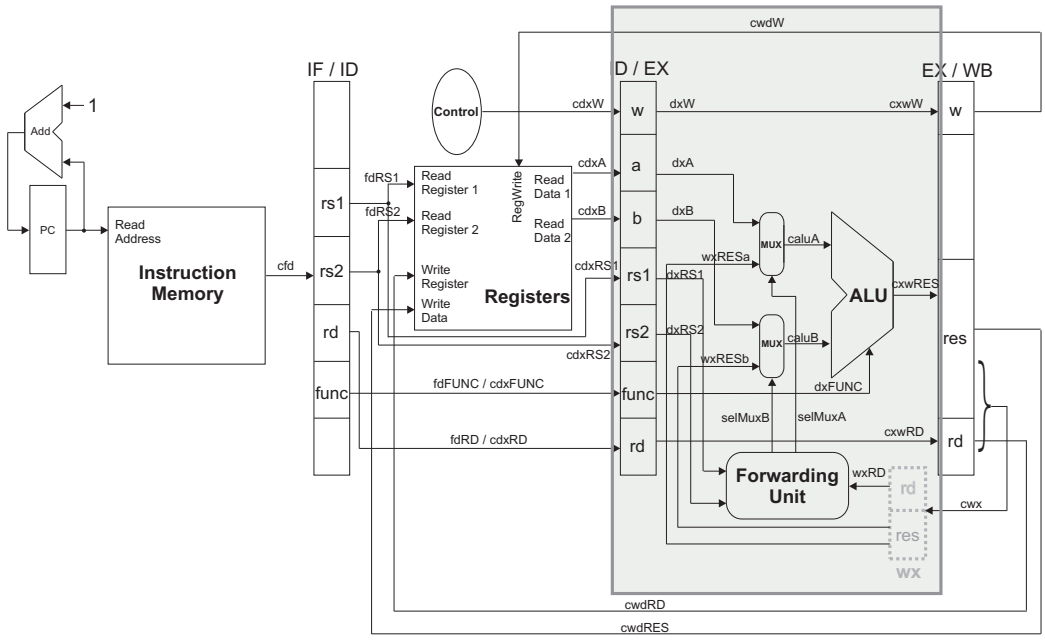


Figure 6.6: EX stage block diagram

the operation specified by the $cdxFUNC$ channel on the a and b values. The result

and the destination register are sent to next stage via $cxwRES$ $cxwRD$ channels respectively.

This stage is implemented as six parallel processes in the following procedure:

$$(cxwW, cxwRES, cxwRD) ::= \boxed{EX_{par}}(cdxW, cdxA, cdxB, cdxRS1, cdxRS2, cdxFUNC, cdxRD, cwx) ::$$

$$\left[\begin{array}{l}
\left[\begin{array}{l}\textbf{loop forever do } (\ ID/EX\ (dx)\ register\ ) \\ \quad \left[\begin{array}{l} dxA \Leftarrow dx.a;\ dxB \Leftarrow dx.b;\ dxRS1 \Leftarrow dx.rs1;\ dxRS2 \Leftarrow dx.rs2;\ dxFUNC \Leftarrow dx.func; \\ xxw.w := dx.w;\ xxw.rd := dx.rd; \\ cdxW \Rightarrow dx.w;\ cdxA \Rightarrow dx.a;\ cdxB \Rightarrow dx.b;\ cdxRS1 \Rightarrow dx.rs1;\ cdxRS2 \Rightarrow dx.rs2; \\ cdxFUNC \Rightarrow dx.func;\ cdxRD \Rightarrow dx.rd;\ cxwW \Leftarrow xxw.w;\ cxwRD \Leftarrow xxw.rd \end{array}\right]\end{array}\right] \\
\| \\
\left[\begin{array}{l}\textbf{loop forever do } (\ wx\ register\ ) \\ \quad \left[\begin{array}{l} cwx \Rightarrow wx; \\ wxRESa \Leftarrow wx.res;\ wxRESb \Leftarrow wx.res;\ wxRD \Leftarrow wx.rd \end{array}\right]\end{array}\right] \\
\| \\
\left[\begin{array}{l}\textbf{loop forever do } (\ Forwarding\ control\ ) \\ \quad \left[\begin{array}{l} dxRS1 \Rightarrow rs1;\ dxRS2 \Rightarrow rs2; \\ wxRD \Rightarrow rd; \\ selectA := (rs1 = rd);\ selectB := (rs2 = rd); \\ selMuxA \Leftarrow selectA;\ selMuxB \Leftarrow selectB \end{array}\right]\end{array}\right] \\
\| \\
\left[\begin{array}{l}\textbf{loop forever do } (\ Multiplexor\ of\ ALU\ input\ A\ ) \\ \quad \left[\begin{array}{l} dxA \Rightarrow a;\ wxRESa \Rightarrow resA;\ selMuxA \Rightarrow selA; \\ \textbf{if } selA \textbf{ then } [outA := resA] \textbf{ else } [outA := a]; \\ caluA \Leftarrow outA \end{array}\right]\end{array}\right] \\
\| \\
\left[\begin{array}{l}\textbf{loop forever do } (\ Multiplexor\ of\ ALU\ input\ B\ ) \\ \quad \left[\begin{array}{l} dxB \Rightarrow b;\ wxRESb \Rightarrow resB;\ selMuxA \Rightarrow selB; \\ \textbf{if } selB \textbf{ then } [outB := resB] \textbf{ else } [outB := b]; \\ caluB \Leftarrow outB \end{array}\right]\end{array}\right] \\
\| \\
\left[\begin{array}{l}\textbf{loop forever do } (\ ALU\ ) \\ \quad \left[\begin{array}{l} caluA \Rightarrow aluA; caluB \Rightarrow aluB; \\ dxFUNC \Rightarrow func; \\ xxw.res := alures(func, aluA, aluB); \\ cxwRES \Leftarrow xxw.res \end{array}\right]\end{array}\right]
\end{array}\right]$$

Table 6.5 contains the local variables, and their sizes.

The EX stage has $91 + 6 + 37 + 17 + 97 + 97 + 107 = 452$ bits.

dx register

Var	Fields	Bits
dx	a	32
	b	32
	rs1	5
	rs2	5
	func	11
	w	1
	rd	5
		91

Var	Fields	Bits
xxw	w	1
	rd	5
		6

wx register

Var	Fields	Bits
wx	res	32
	rd	5
		37

Forwarding control

Var	Bits
rs1	5
rs2	5
rd	5
selectA	1
selectB	1
	17

Multiplexor A

Var	Bits
a	32
resA	32
selA	1
outA	32
	97

Multiplexor B

Var	Bits
b	32
resB	32
selB	1
outB	32
	97

ALU

Var	Bits
aluA	32
aluB	32
func	11
result	32
	107

Table 6.5: Local variables of procedure EX_{par}

6.2.3.5 Procedure WB_{unh}

This procedure provides the mechanism to store the ALU result into the register file via $cwdW$, $cwdRES$ and $cwdRD$ channels. It also forwards results to EX via the cwx channel. The data from the EX stage is received via $cxwW$, $cxwRES$ and $cxwRD$ channels at the end of the cycle. This is the unhidden form of the WB stage of section 6.2.4.4

$$(cwdW, cwdRES, cwdRD, cwx) ::= \boxed{WB_{unh}}(cxwW, cxwRES, cxwRD) ::$$

$$\left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l} cwdW \Leftarrow xw.w;\\ cwdRES \Leftarrow xw.res;\\ cwdRD \Leftarrow xw.rd;\\ cwx \Leftarrow xw;\\ cxwW \Rightarrow xw.w\\ cxwRES \Rightarrow xw.res\\ cxwRD \Rightarrow xw.rd\end{array}\right]\end{array}\right]$$

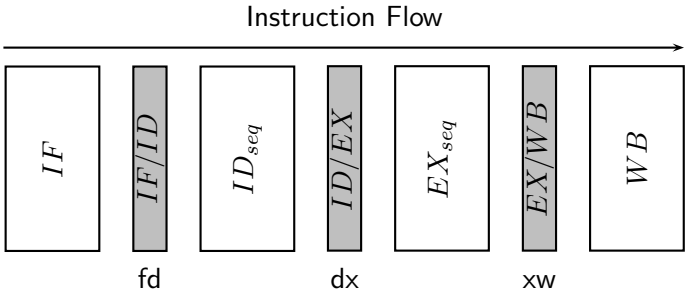
Table 6.6 shows the variable of procedure WB_{unh}, and it size.

Var	Fields	Bits
xw	w	1
	res	32
	rd	5
		38

Table 6.6: Local variable of procedure WB_{unh}

6.2.4 The $Pipeline_1$ Model

This section presents $Pipeline_1$, the sequential version of $Pipeline_2$. It plays the role of intermediate form in the global proof. Its stage procedures have no inner parallelism. In the case of ID_{seq} and EX_{seq}, they will be proven interface equivalent to their parallel versions. The block diagram has the same structure as before:



The following expression represents the architecture as a cooperation of four sequential processes communicating via synchronous channels:

$$Pipeline_1 :: \left[IF \;||\; ID_{seq} \;||\; EX_{seq} \;||\; WB\right]$$

The first level of parallelism is conserved in $Pipeline_1$ but not the second one. Procedures whose implementation was a parallel composition are replaced by their equivalent sequential versions. A simplified schema is shown in figure 6.7.

Pipelining the datapath requires that values passed from one pipe stage to the next must be placed in the pipeline registers. These are fd, dx, and xw. The synchronous communication channels forwarding values to these registers are *cfd*, *cdx*, and *cxw*. Channel *cwd* transfers result values back to the registers. Channel *cwx* transfers forwarding data to the EX stage. The sequential procedures which model the stages are introduced next.

6.2.4.1 Procedure IF

The same as subsection 6.2.3.2.

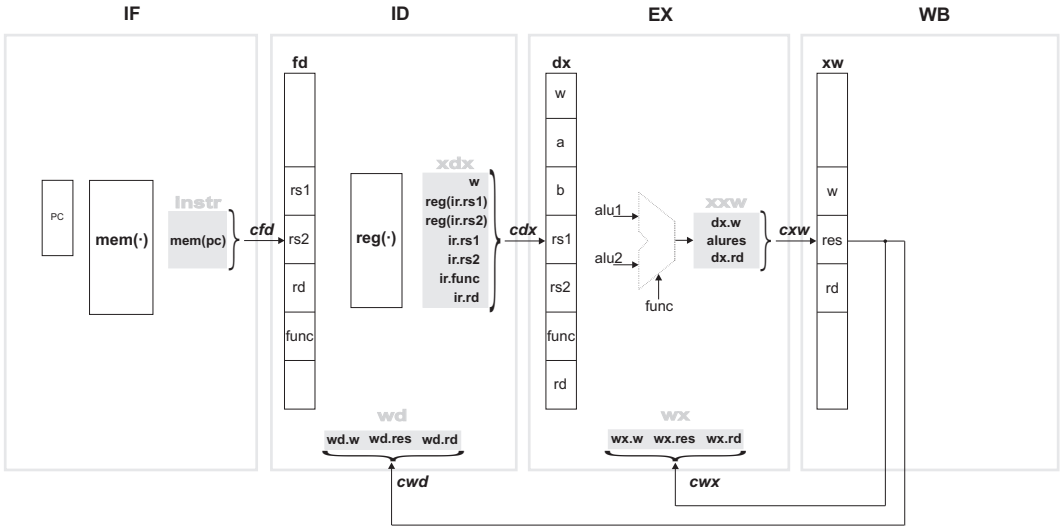


Figure 6.7: $Pipeline_1$ block diagram

6.2.4.2 Procedure ID_{seq}

It is given below. Channel *cwd* brings from stage WB, at the beginning of the cycle, the data to be written into *reg*. The two source values are then read from $reg(\cdot)$ and are kept in the xdx variable. The next instruction is received via the *cfd* channel. The values read from the registers and the register indexes (xdx) are passed to the next stage via channel cdx.

$$(reg, cdx) ::= ID_{seq}(reg, cfd, cwd) ::$$

$$\left[\begin{array}{l}\textbf{loop forever do} \\ \quad \left[\begin{array}{l} cwd \Rightarrow wd; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (w, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ cfd \Rightarrow ir; \\ cdx \Leftarrow xdx \end{array}\right] \end{array}\right]$$

6.2.4.3 Procedure EX_{seq}

It is given below. Channel *cwx* brings in forwarding data. Register dependencies are checked in the if statements, to enable forwarding. The new instruction data

is received via the cdx channel. The ALU result and the corresponding destination register index rd, are sent via the cxw channel to WB.

$$cxw ::= EX_{seq}(cdx, cwx) ::$$

$$\left[\begin{array}{l}\textbf{loop forever do} \\ \quad \left[\begin{array}{l} cwx \Rightarrow wx; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ cdx \Rightarrow dx; \\ cxw \Leftarrow xxw \end{array}\right]\end{array}\right]$$

6.2.4.4 Procedure WB

Transfers the ALU result, via the cwd channel, back to the ID stage. It also forwards results to EX via the cwx channel, and receives new data from EX via channel cxw.

$$(cwd, cwx) ::= WB(cxw) :: \left[\begin{array}{l}\textbf{loop forever do} \\ \quad \left[\begin{array}{l} cwd \Leftarrow xw; \\ cwx \Leftarrow xw; \\ cxw \Rightarrow xw \end{array}\right]\end{array}\right]$$

6.2.4.5 Procedure $Pipeline_1$

Its global program with the variable initializations is the following:

$$reg ::= Pipeline_1(reg, mem) ::$$

$$\left[\begin{array}{llll} \textbf{local} & reg & : \text{array[1..32] of int32} \\ \textbf{local} & pc & : \text{int32} \\ \textbf{local} & w & : \textbf{boolean} \\ \textbf{local} & instr, ir & : \text{Typ\_IR} \\ \textbf{local} & xdx, dx & : \text{Typ\_DX} \\ \textbf{local} & wd, wx, xxw, xw & : \text{Typ\_XW} \\ \textbf{local} & cfd & : \textbf{channel of } \text{Typ\_IR} \\ \textbf{local} & cdx & : \textbf{channel of } \text{Typ\_DX} \\ \textbf{local} & cwd, cxw, cwx & : \textbf{channel of } \text{Typ\_XW} \end{array}\right]$$

$$
\left[\begin{array}{l}
w := true;\\
pc := 1;\\
(ir.rs1, ir.rs2, ir.rd, ir.func) := (0,0,0,0);\\
(dx.w, dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) := (false,0,0,0,0,0,0);\\
(xw.w, xw.res, xw.rd) := (false,0,0);\\
(wx.w, wx.res, wx.rd) := (false,0,0);\\
\\
IF :: \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l} instr := mem(pc);\\ pc := pc+1;\\ cfd \Leftarrow instr\end{array}\right]\end{array}\right]\\
\\
\quad ||\\
\\
ID_{seq} :: \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l} cwd \Rightarrow wd;\\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil;\\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd)\\ \quad :=\\ (w, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd);\\ cfd \Rightarrow ir;\\ cdx \Leftarrow xdx\end{array}\right]\end{array}\right]\\
\\
\quad ||\\
\\
EX_{seq} :: \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l} cwx \Rightarrow wx;\\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil;\\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil;\\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd);\\ cdx \Rightarrow dx;\\ cxw \Leftarrow xxw\end{array}\right]\end{array}\right]\\
\\
\quad ||\\
\\
WB :: \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l} cwd \Leftarrow xw;\\ cwx \Leftarrow xw;\\ cxw \Rightarrow xw\end{array}\right]\end{array}\right]
\end{array}\right]
$$

6.3 Proof Schema

6.3.1 Overview

The proof of the simplified DLX-like pipeline processor model establishes interface equivalence between the parallelism and inner communications model, $Pipeline_2$, and a sequential model. This sequential model corresponds to a simple *Von Neumann* loop processor model, which has neither inner parallelism nor communications. The following program implements this sequential model:

$$reg ::= VNCycle(reg, mem) ::$$

$$\left[\begin{array}{lll}
\textbf{in-out} & reg & : \text{register file} \\
\textbf{external in} & mem & : \text{memory} \\
\textbf{local} & ir & : \text{Typ_IR} \\
\textbf{local} & pc & : \text{int32} \\
 & & \\
\textbf{for } k := 1..n \textbf{ do} & & \\
\quad \left[\begin{array}{l} ir := mem(pc); \\ pc := pc + 1; \\ reg(ir.rd) := alures(ir.func, reg(ir.rs1), reg(ir.rs2)) \end{array}\right] & &
\end{array}\right]$$

The *Von Neumann* processor model executes a program of only register-register instructions stored in the memory, mem. The program length is denoted by integer n, assuming that $n \geq 1$. First it fetches the instruction pointed to by the program counter, pc, and stores it in variable ir. Then the program counter is updated to point to the next instruction to be fetched. The source register indexes are $ir.rs1$ and $ir.rs2$, and the destination register index is $ir.rd$. Procedure $alures$ performs the requested operation, indicated by $ir.func$, on the two source register values, and stores the result in the destination register, $reg(ir.rd)$.

The main goal is to prove the following interface equivalence between the two models with the same essential behavior:

$$[reg := Pipeline_2(reg, mem)] \ =_{\mathcal{O}} \ [reg := VNCycle(reg, mem)] \qquad (6.1)$$

The observed set $\mathcal{O}$ is $\{reg, mem\}$, but since mem contains the program which is only read, the observed set can be reduced to $\mathcal{O} = \{reg\}$. The equivalence is only proved for any finite length program, composed of register to register ALU instructions. Hence, for the same initial values of the register file $reg(\cdot)$, the values of the registers at the end of the program are the same as those resulting from the same program running on *VNCycle* model.

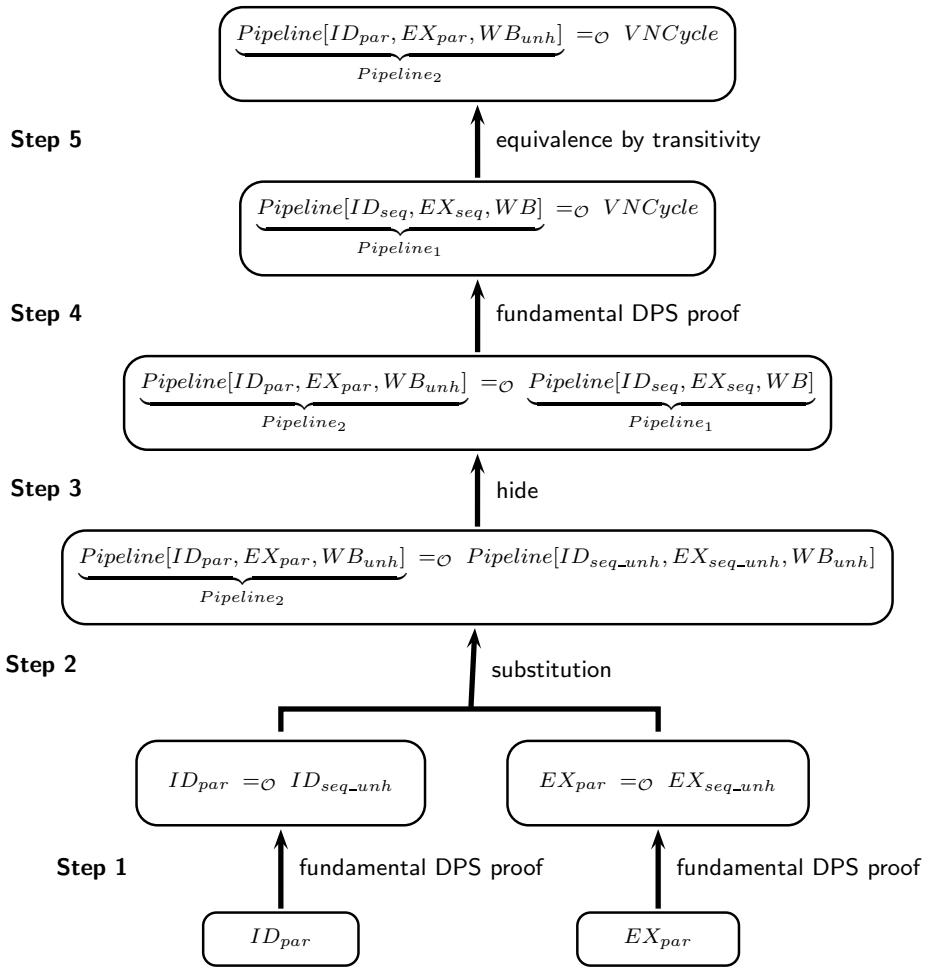


Figure 6.8: Hierarchical proof schema

The hierarchical model helps to organize the proof. It is decomposed into several steps corresponding to proofs of intermediate interface equivalences. Its global schema is shown in figure 6.8. An arrow corresponds to a proof step, whose result is shown within its ending node, above the arrow. In the nodes, the lists of output and input variables and channels of procedure references have been omitted for clarity. Only the procedure name is displayed.

A generic procedure, *Pipeline*, is defined for convenience. This procedure has three holes for the references to the procedures of the ID, EX and WB stages.

$Pipeline_2$ is defined as $Pipeline$ with the references to the equivalent procedures with inner parallelism and unhidden synchronous channels,

$$Pipeline_2 \;=\; Pipeline[ID_{par}, EX_{par}, WB_{unh}].$$

As mentioned before, $Pipeline1$ is defined with the references to sequential procedures,

$$Pipeline_1 \;=\; Pipeline[ID_{seq}, EX_{seq}, WB].$$

The procedure of the IF stage is the same in both implementations.

Interface equivalence 6.1 is proved in five main steps which have to be proved before, figure 6.8 illustrates them. Next subsections will detail each one. An overview of the proof is as follows:

The proof starts by focusing on components ID_{par} and EX_{par} of *Pipeline2*, Step 1. The fundamental proof is applied to each of them obtaining the equivalences

$$ID_{par} \;=_{\mathcal{O}}\; ID_{seq\_unh} \qquad \text{and} \qquad EX_{par} \;=_{\mathcal{O}}\; EX_{seq\_unh}$$

This step corresponds to *step 1* of the hierarchical proof with channel hiding of subsection 3.3.4.

Interface equivalence 6.2 establishes the equivalence between the instruction decode (ID) stage, with inner parallelism and communications, and the sequential version.

$$[(reg, coutID) := ID_{par}(reg, cfd, cinID)]$$
$$=_{\mathcal{O}}$$
$$[(reg, coutID) := ID_{seq\_unh}(reg, cfd, cinID)] \tag{6.2}$$

where $coutID$ is the list of output channels:

$$coutID = (cdxW, cdxA, cdxB, cdxRS1, cdxRS2, cdxFUNC, cdxRD)$$

and $cinID$ is the list of input channels:

$$cinID = (cwdW, cwdRES, cwdRD)$$

and $cinID$ is the list of input channels:

$$cinID = (cwdW, cwdRES, cwdRD)$$

Procedures ID_{par} and ID_{seq_unh} are detailed respectively in subsections 6.2.3.3 and 6.3.5.

Similarly for the execution stage, interface equivalence 6.3 relates parallel and sequential EX versions.

$$[(cxwW, cxwRES, cxwRD) := EX_{par}(cinEX)]$$
$$=_{\mathcal{O}}$$
$$[(cxwW, cxwRES, cxwRD) := EX_{seq\_unh}(cinEX)] \tag{6.3}$$

where $cinEX$ is the list of input channels:

$$cinEX = (cdxW, cdxA, cdxB, cdxRS1, cdxRS2, cdxFUNC, cdxRD, cwx)$$

Procedures EX_{par} and EX_{seq_unh} are detailed respectively in subsections 6.2.3.4 and 6.3.7.

The two above equivalences are substituted into *Pipeline*, in Step 2, obtaining the following equivalent procedure:

$$Pipeline[ID_{seq\_unh}, EX_{seq\_unh}, WB_{unh}]$$

This step corresponds to *step 2* of the hierarchical proof with channel hiding of subsection 3.3.4.

Some groups of connections are identified and hidden in Step 3. The obtained equivalences from procedure $Pipeline_2$ are the following:

$$(reg, coutID) := ID_{seq\_unh}(reg, cfd, cinID)$$
$$=_{\mathcal{O}}$$
$$[(reg, cdx) := ID_{seq}(reg, cfd, cwd)] \tag{6.4}$$

$$(cxwW, cxwRES, cxwRD) := EX_{seq\_unh}(cinEX)$$
$$=_{\mathcal{O}}$$

$$[cxw := EX_{seq}(cdx, cwx)] \tag{6.5}$$

$$(cwdW, cwdRES, cwdRD, cwx) := WB_{unh}(cxwW, cxwRES, cxwRD)$$
$$=_{\mathcal{O}}$$

$$(cwd, cwx) := WB(cxw) \tag{6.6}$$

where ID_{seq} is detailed in subsection 6.2.4.2, EX_{seq} in subsection 6.2.4.3, and WB in subsection 6.2.4.4.

The channel hide-unhide correspondences are:

h1: *cwd* $\longleftrightarrow$ *(cwdW, cwdRES, cwdRD)*
h2: *cxw* $\longleftrightarrow$ *(cxwW, cxwRES, cxwRD)*
h3: *cdx* $\longleftrightarrow$ *(cdxW, cdxA, cdxB, cdxRS1, cdxRS2, cdxFUNC, cdxRD)*

Finally the substitution at once into $Pipeline[ID_{seq\_unh}, EX_{seq\_unh}, WB_{unh}]$ leads to *Pipeline1* completing the step. It corresponds to *step 3* of the hierarchical proof with channel hiding of subsection 3.3.4.

Interface equivalence 6.7 of Step 4

$$[reg := Pipeline_1(reg, mem)] \ =_{\mathcal{O}} \ [reg := VNCycle(reg, mem)] \tag{6.7}$$

is established applying the fundamental proof, as covered in chapter 3. DPS proofs are carried out with the prover tool.

Last step, Step 5, applies transitivity to equivalences 6.2 and

$$[reg := Pipeline_2(reg, mem)] \ =_{\mathcal{O}} \ [reg := Pipeline_1(reg, mem)] \tag{6.8}$$

obtaining the main goal, equivalence 6.1 of page 157.

6.3.2 State Vector Reduction

After proving the interface equivalence between the $Pipeline_2$ and *Von Neumann* processor models, we observe that most of the local variables of $Pipeline_2$ procedures have been removed due to the variable simplification step before obtaining

Procedure	Variables	Bits per Variable	Total Bits per Procedure
$VNCycle$	**pc**	32	
	ir	26	
			58

Table 6.7: Local variables of $VNCycle$

the $VNCycle$ model. Its variables are shown in the above table 6.7. The ones of $Pipeline_2$ are listed in table 6.8.

The variable, reg, is common to both processor models. It includes 32 general purpose registers of 32 bits each, then reg has 32×32=1024 bits.

The total number of bits of $Pipeline_2$ is 58 + 182 + 452 + 38 + 1024 = 1754. In the case of $VNCycle$ is 58 + 1024 = 1082. Note the state vector is reduced drastically, from 1754, for the parallel model, to 1082 bits. The upper bound reduction ratio is the following:

$$reduction\ ratio = \frac{\text{upper bound of num states of } VNCycle}{\text{upper bound of num states of } Pipeline_2} = \frac{2^{1082}}{2^{1754}} = 2^{-672}$$

6.3.3 Proof of $Pipeline_1$

The proof shows that the distributed pipelined architecture $Pipeline_1$ is equivalent to the simple *Von Neumann* processor model, $VNCycle$, which has neither inner parallelism nor communications. The interface equivalence, 6.7, to be proved is:

$$[reg := Pipeline_1(reg, mem)] =_{\mathcal{O}} [reg := VNCycle(reg, mem)]$$

Global Description:

The goal of this proof is to obtain the following basic *Von Neumann* iteration body.

$$VN_{body} :: \begin{bmatrix} ir := mem(pc); \\ pc := pc + 1; \\ reg(ir.rd) := alures(ir.func, reg(ir.rs1), reg(ir.rs2)) \end{bmatrix}$$

The proof starts by unfolding five times the indefinite loops of the four parallel processes in $Pipeline_1$. This is the number required to obtain the first VN iteration body. The unfoldings are necessary because the synchronous communication statements should not appear within indefinite iterations, otherwise the communication elimination reduction could not take place. After applying the communication elimination, parallelism to concatenation, concatenation commutativity, and variable elimination reductions the following form will be obtained:

Procedure	Variables	Bits per Variable	Total Bits per Procedure
IF	**pc**	32	
	instr	26	
			58
ID_{par}	**fd**	26	
	wd	38	
	ir	26	
	xdx	91	
	c	1	
			182
EX_{par}	**dx**	91	
	xxw	6	
	wx	37	
	rs1	5	
	rs2	5	
	rd	5	
	selectA	1	
	selectB	1	
	a	32	
	resA	32	
	selA	1	
	outA	32	
	b	32	
	resB	32	
	selB	1	
	outB	32	
	aluA	32	
	aluB	32	
	func	11	
	result	32	
			452
WB_{unh}	**xw**	38	
			38
			total ... *730*

Table 6.8: Variables of $Pipeline_2$

$$Pipeline_1 =_{\mathcal{O}} I; VN_{body}; M; E$$

where I denotes the variable initializations, VN_{body} is the intermediate form which contains the first *Von Neumann* iteration body after eliminating the redundant variables, and M is a resulting form which has neither parallelism nor inner communication. E is the tail statement which contains again the four parallel processes of $Pipeline_1$.

The proof continues by unfolding the indefinite loops of the four parallel processes of E once, and applying again the DPS proof. As a result, a new form which contains another VN_{body} and the same tail statements, $M; E$, is obtained, thus reaching the equivalence:

$$M;\ E =_{\mathcal{O}} VN_{body};\ M;\ E$$

Then, from the two last equivalences, by induction and substitution, the following equivalence will be obtained:

$$Pipeline_1 =_{\mathcal{O}} I;\ [VN_{body}]^n;\ M;\ E$$

for any finite integer n. If n is equal to the length of the program in the instruction memory, mem, the tail statements $M; E$ can be dropped since the state of the register file, reg, after the n^{th} instruction has written its result is the same in $Pipeline_1$ and in $I;\ [VN_{body}]^n$.

The final form is:

$$Pipeline_1 =_{\mathcal{O}} I;\ {VN_{body}}^n$$

which is the desired result, for programs of length at most n, completing this proof step.

The proof is partitioned in the following steps: constant replacement, unfolding, communication elimination, parallelism to concatenation transformation, concatenation commutativity, and redundant variable elimination. All these steps are carried out with the *interactive prover*.

step (i) - Variable Replacement:

The value of variable w is replaced at ID_{seq} of $Pipeline_1$, obtaining $Pipeline'_1$:

$$
\left[\begin{array}{l}
pc := 1;\\
(ir.rs1, ir.rs2, ir.rd, ir.func) := (0,0,0,0);\\
(dx.w, dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) := (false, 0,0,0,0,0,0);\\
(xw.w, xw.res, xw.rd) := (false, 0, 0);\\
(wx.w, wx.res, wx.rd) := (false, 0, 0);\\
\\
IF :: \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l} instr := mem(pc);\\ pc := pc + 1;\\ cfd \Leftarrow instr\end{array}\right]\end{array}\right] \;\;||\\
\\
ID_{seq} :: \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l} cwd \Rightarrow wd;\\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil;\\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd)\\ \quad :=\\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd);\\ cfd \Rightarrow ir;\\ cdx \Leftarrow xdx\end{array}\right]\end{array}\right] \;\;||\\
\\
EX_{seq} :: \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l} cwx \Rightarrow wx;\\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil;\\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil;\\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd);\\ cdx \Rightarrow dx;\\ cxw \Leftarrow xxw\end{array}\right]\end{array}\right] \;\;||\\
\\
WB :: \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l} cwd \Leftarrow xw;\\ cwx \Leftarrow xw;\\ cxw \Rightarrow xw\end{array}\right]\end{array}\right]
\end{array}\right]
$$

step (ii) - Unfolding Step:

First the unfolding law (see law 9 of page 39) is applied interactively:

Loop Forever Unfold: $[\,\textbf{loop forever do } S\,] \;\approx\; [\,S; \textbf{loop forever do } S]$

to each sequential parallel process of $Pipeline'_1$.

The indefinite loops of the four parallel processes are unfolded five times, as commented above, to unhide the synchronous communication statements.

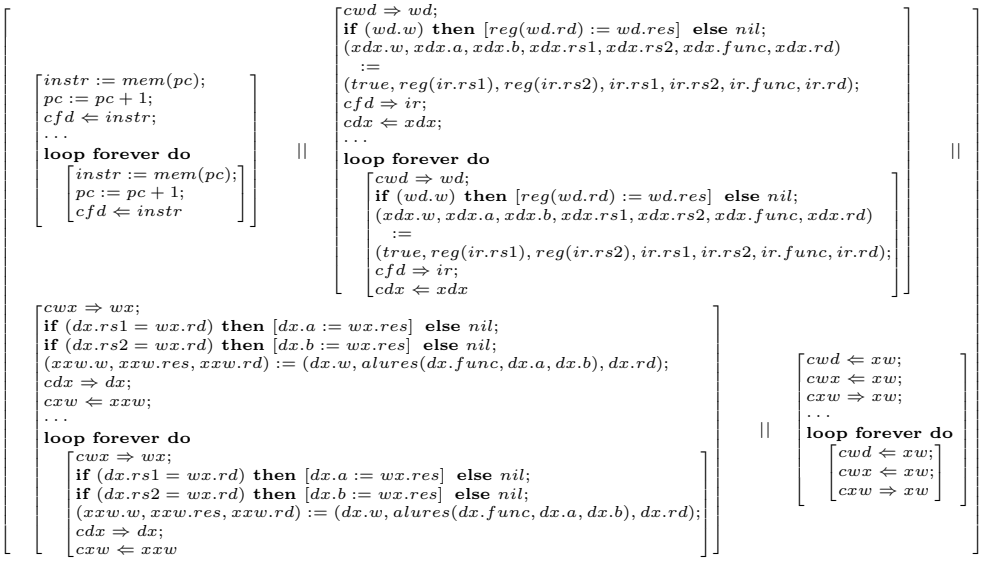
The following shows how the unfolding law is applied five times to the Instruction Fetch (IF) sequential procedure:

$$
\left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l} instr := mem(pc);\\ pc := pc + 1;\\ cfd \Leftarrow instr\end{array}\right]\end{array}\right]
$$

after applying it five times, one obtains,

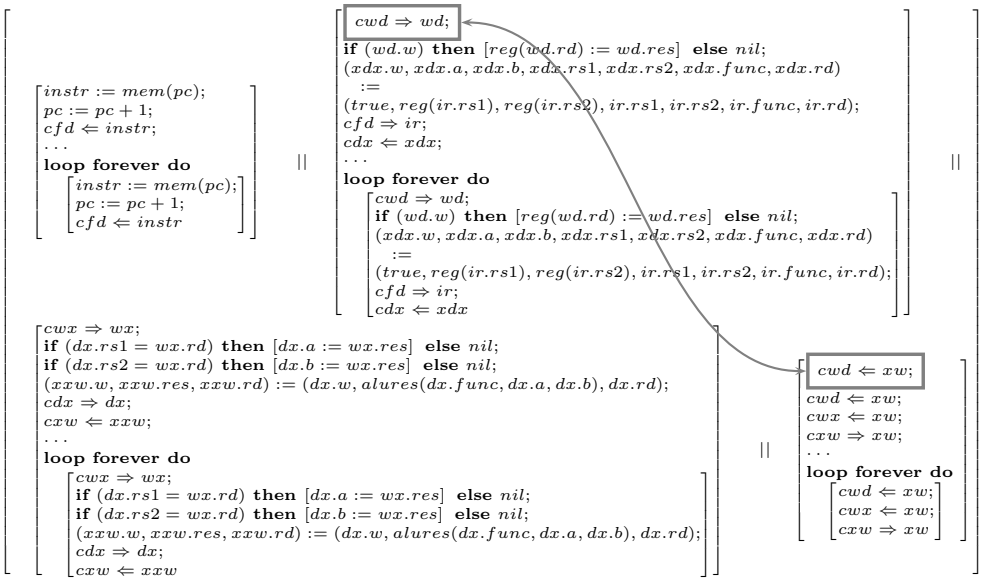
$$
\left\lceil
\begin{array}{l}
instr := mem(pc); \\
pc := pc + 1; \\
cfd \Leftarrow instr; \\
instr := mem(pc); \\
pc := pc + 1; \\
cfd \Leftarrow instr; \\
instr := mem(pc); \\
pc := pc + 1; \\
cfd \Leftarrow instr; \\
instr := mem(pc); \\
pc := pc + 1; \\
cfd \Leftarrow instr; \\
instr := mem(pc); \\
pc := pc + 1; \\
cfd \Leftarrow instr; \\
\textbf{loop forever do} \\
\quad \left\lceil
\begin{array}{l}
instr := mem(pc); \\
pc := pc + 1; \\
cfd \Leftarrow instr
\end{array}
\right\rceil
\end{array}
\right\rceil
$$

After applying to each parallel process, where the dots $(\cdots)$ represent the four repetitions of the prior program segments, the following is obtained:



step (iii) - Communication Elimination:

Once the synchronous communication statements are out of the indefinite iterations, the *iterative communication elimination* reduction procedure, GEN-COMELI of page 104, can be applied. This will eliminate automatically all the matching pairs of communication statements of the above SPL form. As an illustration next figure shows the first eliminable pair which will generate a communication event over the synchronous channel *cwd*.



When the *iterative communication elimination* reduction ends, all internal communications have been eliminated. The resulting form is $I_0; P_0; P_0; P_0; P_0; E_0$, therefore, the following equivalence holds:

$$Pipeline_1 \ =_{\mathcal{O}} \ I_0; P_0; P_0; P_0; P_0; E_0 \tag{6.9}$$

I_0 contains the variable initializations of $Pipeline_1$.

$$I_0 :: \left[\begin{array}{l} pc := 1; \\ w := true; \\ (ir.rs1, ir.rs2, ir.rd, ir.func) := (0,0,0,0); \\ (dx.w, dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) := (false,0,0,0,0,0,0); \\ (xw.w, xw.res, xw.rd) := (false,0,0); \\ (wx.w, wx.res, wx.rd) := (false,0,0) \end{array}\right]$$

P_0 has no references to the local synchronous channels $(cfd, cdx, cwd, cxw, cwx)$ which have been eliminated. It has the form:

$$P_0 :: \tag{6.10}$$

$$\left[\begin{array}{l} wd := xw; \\ \left[\begin{array}{l} \left[\begin{array}{l} \left[\begin{array}{l} \left[\begin{array}{l} instr := mem(pc); \\ pc := pc + 1 \end{array}\right] \\ ir := instr \end{array}\right] \ || \ \left[\begin{array}{l} \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right] \end{array}\right] ; \\ [wx := xw] \end{array}\right] \ || \ ; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ dx := xdx; \\ xw := xxw \end{array}\right]$$

The tail statement E_0 contains the four indefinite iterations of E, preceded by other statements.

$$E_0 :: \tag{6.11}$$

$$\left[\begin{array}{l}
wd := xw; \\
\left[\begin{array}{l}
\left[\begin{array}{l}
\left[\begin{array}{l}
\left[\begin{array}{l} instr := mem(pc); \\ pc := pc + 1 \end{array}\right] \\
ir := instr
\end{array}\right] \\
[wx := xw]
\end{array}\right] \\
\left[\begin{array}{l}
\textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\
(xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\
\quad := \\
(true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd)
\end{array}\right]
\end{array}\right] \\
\left[\begin{array}{l}
\textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\
\textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\
(xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\
dx := xdx; \\
\left[\begin{array}{l}
\left[\begin{array}{l}
xw := xxw; \\
\left[\begin{array}{l}
\textbf{loop forever do} \\
\left[\begin{array}{l}
cwx \Rightarrow wx; \\
\textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\
\textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\
(xxw.w, xxw.res, xxw.rd) \\
\quad := \\
(dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\
cdx \Rightarrow dx; \\
cxw \Leftarrow xxw
\end{array}\right]
\end{array}\right] \\
\left[\begin{array}{l}
\textbf{loop forever do} \\
\left[\begin{array}{l}
cwd \Leftarrow xw; \\
cwx \Leftarrow xw; \\
cxw \Rightarrow xw
\end{array}\right]
\end{array}\right]
\end{array}\right] \\
\left[\begin{array}{l}
\textbf{loop forever do} \\
\left[\begin{array}{l}
cwd \Rightarrow wd; \\
\left[\begin{array}{l}
\textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\
(xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\
\quad := \\
(true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\
cfd \Rightarrow ir; \\
cdx \Leftarrow xdx
\end{array}\right]
\end{array}\right]
\end{array}\right]
\end{array}\right] \\
\left[\begin{array}{l}
\textbf{loop forever do} \\
\left[\begin{array}{l}
\left[\begin{array}{l} instr := mem(pc); \\ pc := pc + 1; \\ cfd \Leftarrow instr \end{array}\right]
\end{array}\right]
\end{array}\right]
\end{array}\right]$$

step (iv) - Parallelism to Concatenation Transformation:

Some inner cooperation statements have appeared as a result of communication elimination. These statements can be transformed to statements in sequence, in the present step, with various applications of the *Cooperation and Concatenation* transformation procedure of section 5.4.2.9 of page 136, and law 10 of page 39. Thus obtaining a truly sequential form, with neither internal communication nor parallelism.

Basically this step applies the following three lemmas from left to right:

$$\begin{aligned} [\,[\,A;\ B\,] \,||\, C\,] \quad &=_{\mathcal{O}} \quad [\,A;\ [\,B \,||\, C\,]\,] \\ [\,[\,A \,||\, B\,];\ C\,] \quad &=_{\mathcal{O}} \quad [\,B;\ A;\ C\,] \\ [\,A \,||\, B\,] \quad &=_{\mathcal{O}} \quad [\,B;\ A\,] \end{aligned}$$

The **aim of the step** is the sequentialization of the parallel composition which occurs at the beginning of the two statements 6.10 and 6.11. This common parallelism is the following:

$$\left[\begin{array}{l} \left[\begin{array}{l} \left[\begin{array}{l} \left[\begin{array}{l} instr := mem(pc); \\ pc := pc + 1 \end{array}\right] \,||\, \left[\begin{array}{l} \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right] \end{array}\right]; \\ ir := instr \end{array}\right] \,||\, \right] \\ \left[wx := xw\right] \end{array}\right]$$

The goal of the transformation is to reach a form where the statements:

$$\left[\begin{array}{l} instr := mem(pc); \\ pc := pc + 1 \end{array}\right] \qquad \text{and} \qquad \left[ir := instr\right]$$

are adjacent in sequence. This is motivated by the *Von Neumann* form of the goal of the proof:

$$VN_{body} :: \left[\begin{array}{l} \overbrace{ir := mem(pc);} \\ \overbrace{pc := pc + 1;} \\ reg(ir.rd) := alures(ir.func, reg(ir.rs1), reg(ir.rs2)) \end{array}\right]$$

Realizing that variable *instr* will be ultimately eliminated, the match will be complete.

Reducing equivalence 6.9, as detailed in section B.1 of appendix B, one obtains:

$$Pipeline_1 \ =_{\mathcal{O}} \ I_0; P_1; P_1; P_1; P_1; E \tag{6.12}$$

where I_0 is detailed in page 168, and

$$P_1 :: \left[\begin{array}{l} wd := xw; \\ wx := xw; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ instr := mem(pc); \\ pc := pc + 1; \\ ir := instr; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ dx := xdx; \\ xw := xxw \end{array}\right] \tag{6.13}$$

$$E :: \left[IF \parallel ID \parallel EX \parallel WB\right]$$

From now on in the text of this proof, the four parallel processes of $Pipeline_1$ are replaced by their identifiers. These are the following:

$$IF :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{l} instr := mem(pc); \\ pc := pc + 1; \\ cfd \Leftarrow instr \end{array}\right] \end{array}\right]$$

$$ID :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{l} cwd \Rightarrow wd; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ cfd \Rightarrow ir; \\ cdx \Leftarrow xdx \end{array}\right] \end{array}\right]$$

$$EX :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{l} cwx \Rightarrow wx; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ cdx \Rightarrow dx; \\ cxw \Leftarrow xxw \end{array}\right] \end{array}\right]$$

$$WB :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{l} cwd \Leftarrow xw; \\ cwx \Leftarrow xw; \\ cxw \Rightarrow xw \end{array}\right] \end{array}\right]$$

The detailed explanation of each transformation can be found in section B.1 of appendix B.

step (v) - Concatenation Commutativity:

The **aim of the step** is to rearrange the body of P_1 so that the instruction fetch statements:

$$\left[\begin{array}{l} instr := mem(pc); \\ pc := pc + 1; \\ ir := instr \end{array}\right]$$

and machine instruction execution statement corresponding to the just fetched:

$$\left[\begin{array}{l} ... \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ ... \end{array}\right]$$

which is located in the next P_1 in sequence, become closer. This closening transformation is carried out in the first two P_1 statements in the sequence within 6.12. Then repeated on the form resulting from the second P_1 and the third P_1, and so forth until the fifth P_1 is transformed. This is possible since most of the statements of P_1 are disjoint, and simple concatenation permutation rules can be applied. Proceeding in this way, from equivalence 6.12 obtaining:

$$Pipeline_1 \ =_{\mathcal{O}} \ I_0; R_0; P_3; P_3; P_3; P_3; E_2; E \tag{6.14}$$

where,

$$R_0 :: \left[\begin{array}{l} wd := xw; \\ wx := xw; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ dx := xdx; \\ xw := xxw \end{array}\right]$$

$$P_3 :: \left[\begin{array}{l} wd := xw; \\ wx := xw; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ instr := mem(pc); \\ ir := instr; \\ pc := pc + 1; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ dx := xdx; \\ xw := xxw \end{array}\right]$$

$$E_2 :: \left[\begin{array}{l} instr := mem(pc); \\ ir := instr; \\ pc := pc + 1 \end{array}\right]$$

This is in preparation for late stages of the proof, where a statement that will match the third statement of the *Von Neumann* body is obtained:

$$VN_{body} :: \left[\begin{array}{l} ir := mem(pc); \\ pc := pc + 1; \\ reg(ir.rd) := alures(ir.func, reg(ir.rs1), reg(ir.rs2)) \end{array}\right]$$

The above closening transformations are achieved by applying the *concatenation commutativity* transformation procedure, section 5.4.2.6. It permutes disjoint and non-communicating statements with the law:

$$[\ A;\ B\ ] =_{\mathcal{O}} [\ B; A\ ]$$

For further details see section B.2 of appendix B.

step (vi) - Redundant Variable Elimination:

Some redundant variable assignments have appeared in the latter sequential composition due to the elimination of the synchronous channels done in the communication elimination step. The **aim of the stage** is the interactive application of the variable elimination reductions of section 5.4 to remove them and to reach a simpler sequential form.

The variable elimination is carried out within the interactive prover by applying the following *variable and assignment elim-intro* law (see law 13 of page 40):

$$[\ v := e; S_1(v); S_2\ ] \quad =_{\mathcal{O}} \quad [\ S_1(e); S_2\ ]$$

and *multiple variable and assignment elim-intro* law (see law 17 of page 41):

$$[\ (v.v_1, ..., v.v_n) := (e_1, ..., e_n);\ S_1(v.v_1, ..., v.v_n);\ S_2\ ] =_{\mathcal{O}} [\ S_1(e_1, ..., e_n);\ S_2\ ]$$

and *multiple variable and assignment partial elim-intro* law (see law 18 of page 41):

$$\begin{array}{c}[\ (v.v_1, ..., v.v_i, ..., v.v_j, ..., v.v_n) := (e_1, ..., e_i, ..., e_j, ..., e_n);\ S_1(v.v_i, ..., v.v_j);\ S_2\ ] \\ =_{\mathcal{O}} \\ [\ (v.v_1, ..., v.v_n) := (e_1, ..., e_n);\ S_1(e_i, ..., e_j);\ S_2\ ]\end{array}$$

They are applied from left to right.

In this step, variables $instr$, xw, xdx, wd, wx, $dx.w$, $xxw.w$, $dx.rs1$, and $dx.rs2$, are removed from I_0, R_0, and P_3 of equivalence 6.14. Also some simple '**if** ' statements with boolean conditions are simplified applying one of the following trivial congruences:

if '$true$' congruence: **if** $true$ **then** S_1 **else** $S_2 \approx S_1$

if '$false$' congruence: **if** $false$ **then** S_1 **else** $S_2 \approx S_2$

The new equivalence obtained from 6.14 is:

$$Pipeline_1 =_{\mathcal{O}} I_1; R_3; U_1; U_1; U_1; E_4; E \tag{6.15}$$

where,

$$I_1 :: \left[\begin{array}{l} pc := 1; \\ (ir.rs1, ir.rs2, ir.rd, ir.func) := (0,0,0,0); \\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) := (0,0,0,0,0,0); \\ (wx.w, wx.res, wx.rd) := (false, 0, 0) \end{array}\right]$$

$$R_3 :: \left[\begin{array}{l} \textbf{if } (false) \textbf{ then } [reg(0) := 0] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = 0) \textbf{ then } [dx.a := 0] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = 0) \textbf{ then } [dx.b := 0] \textbf{ else } nil; \\ (xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd) \end{array}\right]$$

$$U_1 :: \left[\begin{array}{l} (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ reg(xxw.rd) := xxw.res; \\ \textbf{if } (ir.rs1 = xxw.rd) \textbf{ then } [dx.a := xxw.res] \textbf{ else } nil; \\ \textbf{if } (ir.rs2 = xxw.rd) \textbf{ then } [dx.b := xxw.res] \textbf{ else } nil; \\ (xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1 \end{array}\right]$$

$$E_4 :: \left[\begin{array}{l} (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ (wd.w, wd.res, wd.rd) := (true, xxw.res, xxw.rd); \\ (wx.w, wx.res, wx.rd) := (true, xxw.res, xxw.rd); \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (true, alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ dx := xdx; \\ xw := xxw; \\ instr := mem(pc); \\ ir := instr; \\ pc := pc + 1 \end{array}\right]$$

The details can be found in section B.3 of appendix B.

step (vii) - Data Forwarding Elimination:

The essence of data forwarding is still present in U_1. Observe its two consecutive **if** statements. These pass the correct values to the two ALU inputs, $dx.a$ and $dx.b$, when they have been incorrectly read from the registers $ir.rs1$ and $ir.rs2$ in the first multiple assignment of U_1. The correct value is $xxw.res$, which was stored in register $xxw.rd$ at the assignment preceding the **if** statements of U_1. The assignments, $dx.a := xxw.res$ and $dx.b := xxw.res$, take place when an instruction reads a register value before it has been updated by the previous instruction, which is at the WB stage. The equalities $dx.rs1 = xxw.rd$ and $dx.rs2 = xxw.rd$, check for the incorrect read situation between source register indexes, variables $dx.rs1$ and

$dx.rs2$, of the current instruction and the destination register index of the previous instruction, variable $xxw.rd$.

The **aim of the stage** is the elimination of the The data forwarding applying the following lemma, from left to right:

$$\left[\begin{array}{l}(a,b,c,d,e,f) := (r(i),r(j),i,j,t,s);\\ r(k) := q;\\ [\,\textbf{if}\ \ i = k\ \ \textbf{then}\ \ a := q\ \ \textbf{else}\ \ nil];\\ [\,\textbf{if}\ \ j = k\ \ \textbf{then}\ \ b := q\ \ \textbf{else}\ \ nil]\end{array}\right] =_{\mathcal{O}} \left[\begin{array}{l}r(k) := q;\\ (a,b,c,d,e,f) := (r(i),r(j),i,j,t,s)\end{array}\right]$$

Justification Basically the lemma removes both **if** statements due to the movement of the assignment $r(k) := q$. The equivalence guarantees that the variables a and b have the same value in both sides. If the assignment $r(k) := q$ is placed before the multiple assignment, the variables a and b always have the correct value, independent of the value of i and j, and the **if** statements are not longer needed.

□

The lemma is applied to U_1 with the following matchings:

$$\begin{array}{rcl}(a,b,c,d,e,f) := (r(i),r(j),i,j,t,s) & :: & \left[\begin{array}{l}(dx.a,dx.b,dx.rs1,dx.rs2,dx.func,dx.rd)\\ \quad :=\\ (reg(ir.rs1),reg(ir.rs2),ir.rs1,ir.rs2,ir.func,ir.rd)\end{array}\right]\\ r(k) := q & :: & reg(xxw.rd) := xxw.res\\ \textbf{if}\ \ i = k\ \ \textbf{then}\ \ a := q\ \ \textbf{else}\ \ nil & :: & \textbf{if}\ (ir.rs1 = xxw.rd)\ \textbf{then}\ [dx.a := xxw.res]\ \textbf{else}\ nil\\ \textbf{if}\ \ j = k\ \ \textbf{then}\ \ b := q\ \ \textbf{else}\ \ nil & :: & \textbf{if}\ (ir.rs2 = xxw.rd)\ \textbf{then}\ [dx.b := xxw.res]\ \textbf{else}\ nil\end{array}$$

After the reduction one obtains U_1':

$$U_1' :: \left[\begin{array}{l}reg(xxw.rd) := xxw.res;\\ (dx.a,dx.b,dx.rs1,dx.rs2,dx.func,dx.rd)\\ \quad :=\\ (reg(ir.rs1),reg(ir.rs2),ir.rs1,ir.rs2,ir.func,ir.rd);\\ (xxw.res,xxw.rd) := (alures(dx.func,dx.a,dx.b),dx.rd);\\ ir := mem(pc);\\ pc := pc + 1\end{array}\right]$$

Observe that now the instruction fetch and the machine instruction execution statements, located in two U_1' in sequence, are much closer. Notice that the bypass mechanism has disappeared since the register, $reg(xxw.rd) := xxw.res$, is written first.

The overall equivalence obtained from 6.15 is:

$$Pipeline_1 =_{\mathcal{O}} I_1; R_3; U_1'; U_1'; U_1'; E_4; E \tag{6.16}$$

step (viii) - Obtaining the first Von Neumann Body:

The desired form will be obtained after applying again the concatenation commutativity, the redundant variable elimination and the simple '**if** ' statements elimination in the same way as the previous steps to equivalence 6.16. The resulting equivalence contains the first *Von Neumann* body, it is the following:

$$Pipeline_1 =_{\mathcal{O}} I; VN_{body}; M; E \tag{6.17}$$

where,

$$I :: \left[\begin{array}{l} pc := 1; \\ reg(0) := (alures(0,0,0); \\ reg(0) := alures(0, reg(0), reg(0)) \end{array}\right]$$

$$VN_{body} :: \left[\begin{array}{l} ir := mem(pc); \\ pc := pc + 1; \\ reg(ir.rd) := alures(ir.func, reg(ir.rs1), reg(ir.rs2)) \end{array}\right]$$

$$M :: \left[\begin{array}{l} ir := mem(pc); \\ pc := pc + 1; \\ (xxw.res, xxw.rd) := (alures(ir.func, reg(ir.rs1), reg(ir.rs2)), ir.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ (wd.w, wd.res, wd.rd) := (true, xxw.res, xxw.rd); \\ (wx.w, wx.res, wx.rd) := (true, xxw.res, xxw.rd); \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (true, alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ dx := xdx; \\ xw := xxw; \\ instr := mem(pc); \\ ir := instr; \\ pc := pc + 1 \end{array}\right]$$

$$E :: \quad [\, IF \,||\, ID \,||\, EX \,||\, WB \,]$$

Each transformation applied is detailed in section B.4 of appendix B.

step (ix) - Tail Statements:

The goal of this step is to prove the following equivalence:

$$M;\ E\ =_{\mathcal{O}}\ VN_{body};\ M;\ E \tag{6.18}$$

obtaining another VN_{body} from tail statements. M and E are the tail statements of equivalence 6.17. The details of each step of the next proof are explained in section B.5 of appendix B, hence only a brief description is given.

Unfolding the indefinite loops of the four parallel processes of E once, and applying the communication elimination reduction, one obtains:

$$M;\ E\ =_{\mathcal{O}}\ M;\ E_0$$

E_0, statement 6.11, is transformed as in the first steps of the proof, page 235 of appendix B, obtaining the following new equivalence:

$$M;\ E\ =_{\mathcal{O}}\ M;\ P_0;\ E \tag{6.19}$$

where P_0 corresponds to statement 6.10 of page 168.

The proof continues by reducing [M; P_0] applying the following transformations:

- concatenation commutativity
- redundant variable elimination
- data forwarding elimination
- simple '**if** ' simplification

the steps are detailed in section B.5 of appendix B.

In the resulting form:

$$M;\ P_0\ =_{\mathcal{O}}\ VN_{body};\ M$$

a new VN_{body} statement is obtained, then the equivalence 6.18 results from 6.19.

step (x) - Induction Step:

The previous step proves that for each unfolding of the indefinite loops of the four parallel processes of E, and applying again the DPS proof, a new VN_{body} statement is obtained, equivalence 6.18 establishes it. Thus, after $n-1$ unfoldings, the following equivalence will be obtained:

$$M;\ E\ =_{\mathcal{O}}\ [VN_{body}]^{n-1};\ M;\ E \tag{6.20}$$

After substituting equivalence 6.20 in 6.17 one obtains:

$$Pipeline_1\ =_{\mathcal{O}}\ I;[VN_{body}]^n;\ M;\ E$$

for any finite integer n.

Therefore for programs in the instruction memory, mem of length $l = n$, the tail statements $M; E$ can be dropped as commented in the previous global description paragraph. The final equivalence may be reduced to:

$$Pipeline_1\ =_{\mathcal{O}}\ I;[VN_{body}]^n \tag{6.21}$$

6.3.4 Proof of ID_{par}

Goal:

The proof shows how the process ID_{par} of subsection 6.2.3.3 is reduced to the following sequential version:

$$(reg, cdxW, cdxA, cdxB, cdxRS1, cdxRS2, cdxFUNC, cdxRD) ::= ID_{seq\_unh}(reg, cfd, cwdW, cwdRES, cwdRD) ::$$

$$\left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l} cwdW \Rightarrow wd.w;\\ cwdRES \Rightarrow wd.res;\\ cwdRD \Rightarrow wd.rd;\\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil;\\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd)\\ \quad :=\\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd);\\ cfd \Rightarrow ir;\\ cdxW \Leftarrow xdx.w;\\ cdxA \Leftarrow xdx.a;\\ cdxB \Leftarrow xdx.b;\\ cdxRS1 \Leftarrow xdx.rs1;\\ cdxRS2 \Leftarrow xdx.rs2;\\ cdxFUNC \Leftarrow xdx.func;\\ cdxRD \Leftarrow xdx.rd \end{array}\right] \end{array}\right]$$

ID_{seq_unh} is obtained after applying indefinite loop unfoldings, and the DPS proof (communication elimination, parallelism to concatenation, concatenation commutativity, and variable elimination). The proof is similar to the previous one, only some of the steps are outlined.

The interface equivalence, 6.2 of section 6.3.1, to be proved is:

$$[(reg, coutID) := ID_{par}(reg, cfd, cinID)]$$

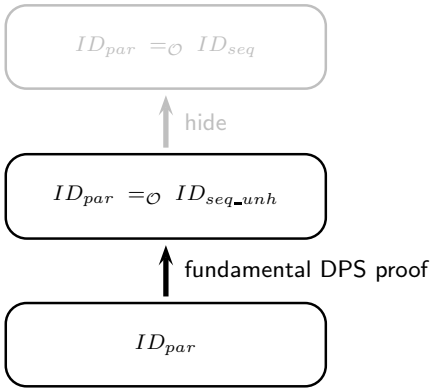
$$=_{\mathcal{O}}$$

$$[(reg, coutID) := ID_{seq\_unh}(reg, cfd, cinID)]$$

where:

$$\begin{array}{lcl} coutID & = & (cdxW, cdxA, cdxB, cdxRS1, cdxRS2, cdxFUNC, cdxRD)\\ cinID & = & (cwdW, cwdRES, cwdRD)\end{array}$$

It corresponds to the following nodes of the tree-schema shown in figure 6.8:



Unfolding and Communication Elimination:

The proof starts by applying twice the indefinite loop unfolding law, this is the number needed to obtain the first sequential body of ID_{seq_unh}. After the unfoldings, when the *iterative communication elimination* reduction, GEN-COMELI of page 104, ends, the following equivalence is obtained:

$$ID_{par} =_{\mathcal{O}} P_0; E_0 \tag{6.22}$$

P_0 has no references to the local channels, ($fdRS1, fdRS2, fdRD, fdFUNC$, *control*), which have been eliminated. It has the form:

$$P_0 ::$$

$$
\left[\begin{array}{l}
\left[\begin{array}{ll} [c := true] & \| \quad \left[\begin{array}{l} cwdW \Rightarrow wd.w; \\ cwdRES \Rightarrow wd.res; \\ cwdRD \Rightarrow wd.rd \end{array}\right] \end{array}\right]; \\
w := c; \\
\left[\begin{array}{ll} [cfd \Rightarrow fd] & \| \quad \left[\begin{array}{l} \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right] \end{array}\right]; \\
ir.rs1 := fd.rs1; \\
ir.rs2 := fd.rs2; \\
ir.rd := fd.rd; \\
ir.func := fd.func; \\
\left[\begin{array}{ll} [c := true] & \| \quad \left[\begin{array}{l} cdxW \Leftarrow xdx.w; \\ cdxA \Leftarrow xdx.a; \\ cdxB \Leftarrow xdx.b; \\ cdxRS1 \Leftarrow xdx.rs1; \\ cdxRS2 \Leftarrow xdx.rs2; \\ cdxFUNC \Leftarrow xdx.func; \\ cdxRD \Leftarrow xdx.rd; \\ cwdW \Rightarrow wd.w; \\ cwdRES \Rightarrow wd.res; \\ cwdRD \Rightarrow wd.rd \end{array}\right] \end{array}\right]; \\
w := c
\end{array}\right]
$$

The tail statement E_0 is the following:

$$E_0 ::$$

$$
\left[\begin{array}{l}
\left[\begin{array}{l}
\left[\begin{array}{l}
\left[cfd \Rightarrow fd\right] \quad \| \quad \left[\begin{array}{l}
\textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\
(xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\
\quad := \\
(true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd)
\end{array}\right]
\end{array}\right]; \\
ir.rs1 := fd.rs1; \\
ir.rs2 := fd.rs2; \\
ir.rd := fd.rd; \\
ir.func := fd.func; \\
\left[\begin{array}{l}
\left[\begin{array}{l}
cfd \Rightarrow fd; \\
\left[\begin{array}{l}
\textbf{loop forever do} \\
\left[\begin{array}{l}
cfd \Rightarrow fd; \\
fdRS1 \Leftarrow fd.rs1; \\
fdRS2 \Leftarrow fd.rs2; \\
fdRD \Leftarrow fd.rd; \\
fdFUNC \Leftarrow fd.func
\end{array}\right]
\end{array}\right]
\end{array}\right] \quad \| \quad
\left[\begin{array}{l}
cdxW \Leftarrow xdx.w; \\
cdxA \Leftarrow xdx.a; \\
cdxB \Leftarrow xdx.b; \\
cdxRS1 \Leftarrow xdx.rs1; \\
cdxRS2 \Leftarrow xdx.rs2; \\
cdxFUNC \Leftarrow xdx.func; \\
cdxRD \Leftarrow xdx.rd; \\
\left[\begin{array}{l}
\textbf{loop forever do} \\
\left[\begin{array}{l}
cwdW \Rightarrow wd.w; \\
cwdRES \Rightarrow wd.res; \\
cwdRD \Rightarrow wd.rd; \\
control \Rightarrow w; \\
\textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\
(xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\
\quad := \\
(w, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\
fdRS1 \Rightarrow ir.rs1; \\
fdRS2 \Rightarrow ir.rs2; \\
fdRD \Rightarrow ir.rd; \\
fdFUNC \Rightarrow ir.func; \\
cdxW \Leftarrow xdx.w; \\
cdxA \Leftarrow xdx.a; \\
cdxB \Leftarrow xdx.b; \\
cdxRS1 \Leftarrow xdx.rs1; \\
cdxRS2 \Leftarrow xdx.rs2; \\
cdxFUNC \Leftarrow xdx.func; \\
cdxRD \Leftarrow xdx.rd
\end{array}\right]
\end{array}\right]
\end{array}\right]
\end{array}\right]
\end{array}\right] \\
\quad \| \quad \left[\begin{array}{l}
\textbf{loop forever do} \\
\quad \left[\begin{array}{l}
c := true; \\
control \Leftarrow c
\end{array}\right]
\end{array}\right]
\end{array}\right]
$$

Obtaining the first ID_{seq_unh} Body:

The inner cooperation statements in P_0, which appeared as a result of communication elimination, and the redundant variables are reduced by applying *Cooperation and Concatenation* and *Variable Elimination* transformations. The resulting equivalence from 6.22 is:

$$ID_{par} \; =_{\mathcal{O}} \; ID_{seq\_unh\_body}; P_1; E_0 \tag{6.23}$$

where $ID_{seq\_unh\_body}$ and P_1 have the forms:

$$ID_{seq\_unh\_body} :: \begin{bmatrix} cwdW \Rightarrow wd.w; \\ cwdRES \Rightarrow wd.res; \\ cwdRD \Rightarrow wd.rd; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ cfd \Rightarrow ir; \\ cdxW \Leftarrow xdx.w; \\ cdxA \Leftarrow xdx.a; \\ cdxB \Leftarrow xdx.b; \\ cdxRS1 \Leftarrow xdx.rs1; \\ cdxRS2 \Leftarrow xdx.rs2; \\ cdxFUNC \Leftarrow xdx.func; \\ cdxRD \Leftarrow xdx.rd \end{bmatrix}$$

$$P_1 :: \begin{bmatrix} cwdW \Rightarrow wd.w; \\ cwdRES \Rightarrow wd.res; \\ cwdRD \Rightarrow wd.rd; \\ c := true; \\ w := c \end{bmatrix}$$

For further details see section C.1 of appendix C.

Induction Step:

The previous step proves that for each unfolding of the indefinite loops of the four parallel processes of E, and applying again the DPS proof, a new VN_{body} statement is obtained, equivalence 6.18 establishes it. Thus, after $n-1$ unfoldings, one obtains:

Applying a similar reduction to $P_1; E_0$ the following equivalence is obtained:

$$P_1; E_0 \ =_{\mathcal{O}} \ [ID_{seq\_unh\_body}]^{n-1}; P_1; E_0$$

After $n-1$ steps, the following form will be obtained from equivalence 6.23:

$$ID_{par} \ =_{\mathcal{O}} \ [ID_{seq\_unh\_body}]^{n}; P_1; E_0$$

For programs of length $l = n$, the equivalence may be reduced to:

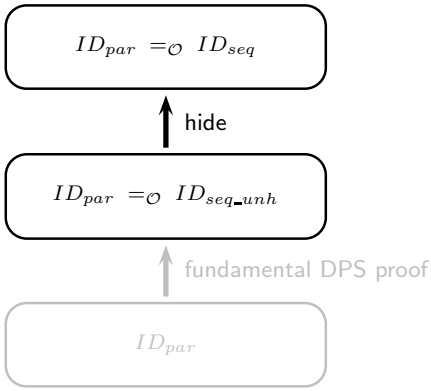
$$ID_{par} \ =_{\mathcal{O}} \ [ID_{seq\_unh\_body}]^{n}$$

6.3.5 Proof of ID_{seq_unh}

Establishing the equivalence 6.4 of subsection 6.3.1:

$$[(reg, coutID) := ID_{seq\_unh}(reg, cfd, cinID)]$$
$$=_{\mathcal{O}}$$
$$[(reg, cdx) := ID_{seq}(reg, cfd, cwd)]$$

where $coutID$ and $cinID$ are the same as above, leads into the equivalence at the top of the following schema, from figure 6.8:



ID_{seq} is shown in subsection 6.2.4.2. This part of the proof consists of a hiding simplification that groups channels of ID_{seq_unh}, shown in page 180. The *hide* function of subsection 3.3.4, is applied with the next correspondences:

$$\begin{array}{lcl} cdx & \longleftarrow & (cdxW, cdxA, cdxB, cdxRS1, cdxRS2, cdxFUNC, cdxRD) \\ cwd & \longleftarrow & (cwdW, cwdRES, cwdRD) \end{array}$$

The type of cdx is Typ_DX, which matches the cartesian product of the types of $cdxW$, $cdxA$, $cdxB$, $cdxRS1$, $cdxRS2$, $cdxFUNC$, $cdxRD$ channels. cwd channel is of type Typ_XW matching the product of the types of $cwdW$, $cwdRES$, $cwdRD$.

6.3.6 Proof of EX_{par}

Goal:

The interface equivalence, 6.3 of subsection 6.3.1, to be proved is:

$$[(cxwW, cxwRES, cxwRD) := EX_{par}(cinEX)]$$

$$=_{\mathcal{O}}$$

$$[(cxwW, cxwRES, cxwRD) := EX_{seq\_unh}(cinEX)]$$

where $cinEX$ is the list of input channels:

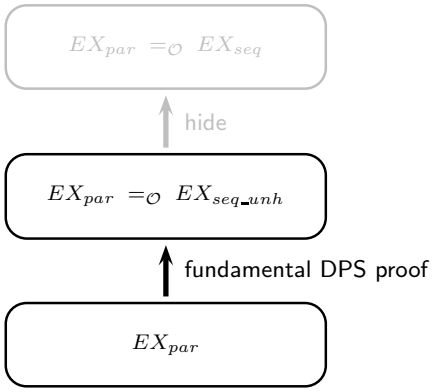
$$cinEX = (cdxW, cdxA, cdxB, cdxRS1, cdxRS2, cdxFUNC, cdxRD, cwx)$$

and

$$(cxwW, cxwRES, cxwRD) := EX_{seq\_unh}(cdxW, cdxA, cdxB, cdxRS1, cdxRS2, cdxFUNC, cdxRD, cwx)$$

$$\left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l}cwx \Rightarrow wx;\\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil;\\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil;\\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd);\\ cdxW \Rightarrow dx.w;\\ cdxA \Rightarrow dx.a;\\ cdxB \Rightarrow dx.b;\\ cdxRS1 \Rightarrow dx.rs1;\\ cdxRS2 \Rightarrow dx.rs2;\\ cdxFUNC \Rightarrow dx.func;\\ cdxRD \Rightarrow dx.rd;\\ cxwW \Leftarrow xxw.w;\\ cxwRES \Leftarrow xxw.res;\\ cxwRD \Leftarrow xxw.rd\end{array}\right]\end{array}\right]$$

As it was done for the decode stage, EX_{par} is reduced to its sequential version EX_{seq_unh}. This step corresponds to the highlighted part of the following piece of figure 6.8:



The main steps are summarized. The proof is similar to the previous one.

Unfolding and Communication Elimination:

The proof starts by applying three times the indefinite loop unfolding law to obtain the first sequential body of EX_{seq_unh}.

After the *iterative communication elimination* reduction, GEN-COMELI of page 104, ends, obtaining the equivalence:

$$EX_{par} =_{\mathcal{O}} P_0; Q_0; E_0 \tag{6.24}$$

where P_0 has the form:

$$\left[\begin{array}{l}
a := dx.a; \\
\left[\begin{bmatrix} cwx \Rightarrow wx; \\ resA := wx.res \end{bmatrix} \;\; || \;\; \big[b := dx.b\big]\right]; \\
\left[\big[resB := wx.res\big] \;\; || \;\; \begin{bmatrix} rs1 := dx.rs1; \\ rs2 := dx.rs2 \end{bmatrix}\right]; \\
rd := wx.rd; \\
selectA := (rs1 = rd); \\
selectB := (rs2 = rd); \\
selA := selectA; \\
\left[\big[selB := selectB\big] \;\; || \;\; \begin{bmatrix} \textbf{if } selA \textbf{ then } [outA := resA] \textbf{ else } [outA := a]; \\ aluA := outA \end{bmatrix}\right]; \\
\textbf{if } selB \textbf{ then } [outB := resB] \textbf{ else } [outB := b]; \\
aluB := outB; \\
func := dx.func; \\
xxw.w := dx.w;\; xxw.rd := dx.rd; \\
cdxW \Rightarrow dx.w;\; cdxA \Rightarrow dx.a;\; cdxB \Rightarrow dx.b; cdxRS1 \Rightarrow dx.rs1;\; cdxRS2 \Rightarrow dx.rs2; \\
cdxFUNC \Rightarrow dx.func;\; cdxRD \Rightarrow dx.rd; \\
cxwW \Leftarrow xxw.w;\; cxwRD \Leftarrow xxw.rd
\end{array}\right]$$

Q_0 is the following:

$$
\left[\begin{array}{l}
a := dx.a; \\
\left[\begin{array}{l} [cwx \Rightarrow wx; \\ resA := wx.res] \end{array}\right] \quad || \quad \left[b := dx.b\right]; \\
\left[\begin{array}{l} [resB := wx.res] \end{array}\right] \quad || \quad \left[\begin{array}{l} rs1 := dx.rs1; \\ rs2 := dx.rs2 \end{array}\right]; \\
rd := wx.rd; \\
selectA := (rs1 = rd); \\
selectB := (rs2 = rd); \\
selA := selectA; \\
\left[\begin{array}{l} \\ [selB := selectB] \quad || \\ \\ \end{array}\right. \left[\begin{array}{l} \left[\begin{array}{l} \left[\begin{array}{l} \textbf{if } selA \textbf{ then } [outA := resA] \textbf{ else } [outA := a] \end{array}\right] \\ \quad || \quad \left[\begin{array}{l} xxw.res := alures(func, aluA, aluB); \\ cxwRES \Leftarrow xxw.res \end{array}\right] \end{array}\right]; \\ aluA := outA; \end{array}\right]; \\
\textbf{if } selB \textbf{ then } [outB := resB] \textbf{ else } [outB := b]; \\
aluB := outB; \\
func := dx.func; \\
xxw.w := dx.w; \\
xxw.rd := dx.rd; \\
cdxW \Rightarrow dx.w; \\
cdxA \Rightarrow dx.a; \\
cdxB \Rightarrow dx.b; \\
cdxRS1 \Rightarrow dx.rs1; \\
cdxRS2 \Rightarrow dx.rs2; \\
cdxFUNC \Rightarrow dx.func; \\
cdxRD \Rightarrow dx.rd; \\
cxwW \Leftarrow xxw.w; \\
cxwRD \Leftarrow xxw.rd
\end{array}\right]
$$

and the tail statement is:

$$E_0 ::$$

$$
\left[\begin{array}{l}
a := dx.a; \\
\left[\begin{array}{l}\left[\begin{array}{l} cwx \Rightarrow wx; \\ resA := wx.res \end{array}\right] \quad || \quad \left[b := dx.b\right]\end{array}\right]; \\
\left[\begin{array}{l}\left[resB := wx.res\right] \quad || \quad \left[\begin{array}{l} rs1 := dx.rs1; \\ rs2 := dx.rs2 \end{array}\right]\end{array}\right]; \\
rd := wx.rd; \\
\left[\begin{array}{l}
\left[\begin{array}{l} selectA := (rs1 = rd); \\ selectB := (rs2 = rd); \\ selA := selectA; \\ \left[\begin{array}{l}\left[selB := selectB\right] \quad || \quad \left[\begin{array}{l}\left[\begin{array}{l}\left[\begin{array}{l}\textbf{if } selA \textbf{ then } [outA := resA] \textbf{ else } [outA := a]\end{array}\right] \\ \quad || \quad \left[\begin{array}{l} xxw.res := alures(func, aluA, aluB); \\ cxwRES \Leftarrow xxw.res \end{array}\right]\end{array}\right]; \\ aluA := outA;\end{array}\right]\end{array}\right]; \\
\left[\begin{array}{l}\textbf{if } selB \textbf{ then } [outB := resB] \textbf{ else } [outB := b]; \\ aluB := outB; \\ \left[\begin{array}{l}\left[\begin{array}{l} func := dx.func; \\ \left[\begin{array}{l}\left[\begin{array}{l} xxw.w := dx.w; \\ xxw.rd := dx.rd; \\ cdxW \Rightarrow dx.w; \\ cdxA \Rightarrow dx.a; \\ cdxB \Rightarrow dx.b; \\ cdxRS1 \Rightarrow dx.rs1; \\ cdxRS2 \Rightarrow dx.rs2; \\ cdxFUNC \Rightarrow dx.func; \\ cdxRD \Rightarrow dx.rd; \\ cxwW \Leftarrow xxw.w; \\ cxwRD \Leftarrow xxw.rd; \\ T_0 \end{array}\right] \quad || \quad \left[\begin{array}{l} xxw.res := alures(func, aluA, aluB); \\ cxwRES \Leftarrow xxw.res; \\ T_1 \end{array}\right]\end{array}\right] \\ || \\ \left[T_2\right]\end{array}\right] \\ || \\ \left[\left[T_3\right] \quad || \quad \left[T_4\right]\right]\end{array}\right] \\ || \\ \left[T_5\right]\end{array}\right]
\end{array}\right]
$$

where:

$$T_0 :: \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l}dxA \Leftarrow dx.a;\ dxB \Leftarrow dx.b;\ dxRS1 \Leftarrow dx.rs1;\ dxRS2 \Leftarrow dx.rs2;\\ \qquad dxFUNC \Leftarrow dx.func;\\ xxw.w := dx.w;\ xxw.rd := dx.rd;\\ cdxW \Rightarrow dx.w;\ cdxA \Rightarrow dx.a;\ cdxB \Rightarrow dx.b;\ cdxRS1 \Rightarrow dx.rs1;\\ \qquad cdxRS2 \Rightarrow dx.rs2;\\ cdxFUNC \Rightarrow dx.func;\ cdxRD \Rightarrow dx.rd;\ cxwW \Leftarrow xxw.w;\\ \qquad cxwRD \Leftarrow xxw.rd\end{array}\right]\end{array}\right]$$

$$T_1 :: \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l}caluA \Rightarrow aluA; caluB \Rightarrow aluB;\\ dxFUNC \Rightarrow func;\\ xxw.res := alures(func, aluA, aluB);\\ cxwRES \Leftarrow xxw.res\end{array}\right]\end{array}\right]$$

$$T_2 :: \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l}dxB \Rightarrow b;\ wxRESb \Rightarrow resB;\ selMuxA \Rightarrow selB;\\ \textbf{if}\ \ selB\ \ \textbf{then}\ \ [outB := resB]\ \ \textbf{else}\ \ [outB := b];\\ caluB \Leftarrow outB\end{array}\right]\end{array}\right]$$

$$T_3 :: \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l}dxRS1 \Rightarrow rs1;\ dxRS2 \Rightarrow rs2;\\ wxRD \Rightarrow rd;\\ selectA := (rs1 = rd);\ selectB := (rs2 = rd);\\ selMuxA \Leftarrow selectA;\ selMuxB \Leftarrow selectB\end{array}\right]\end{array}\right]$$

$$T_4 :: \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l}dxA \Rightarrow a;\ wxRESa \Rightarrow resA;\ selMuxA \Rightarrow selA;\\ \textbf{if}\ \ selA\ \ \textbf{then}\ \ [outA := resA]\ \ \textbf{else}\ \ [outA := a];\\ caluA \Leftarrow outA\end{array}\right]\end{array}\right]$$

$$T_5 :: \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l}cwx \Rightarrow wx;\\ wxRESa \Leftarrow wx.res;\ wxRESb \Leftarrow wx.res;\ wxRD \Leftarrow wx.rd\end{array}\right]\end{array}\right]$$

Obtaining the first EX_{seq_unh} Body:

The following form is obtained from equivalence 6.24 after removing the inner cooperation statements and the redundant variables from $P_0; Q_0$:

$$EX_{par} \ =_{\mathcal{O}}\ EX_{seq\_unh\_body}; Q_1; E_0 \tag{6.25}$$

where $EX_{seq\_unh\_body}$, and $Q1$ have the forms:

$$EX_{seq\_unh\_body} ::$$

$$\left[\begin{array}{l} cwx \Rightarrow wx; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ cdxW \Rightarrow dx.w; \\ cdxA \Rightarrow dx.a; \\ cdxB \Rightarrow dx.b; \\ cdxRS1 \Rightarrow dx.rs1; \\ cdxRS2 \Rightarrow dx.rs2; \\ cdxFUNC \Rightarrow dx.func; \\ cdxRD \Rightarrow dx.rd; \\ cxwW \Leftarrow xxw.w; \\ cxwRES \Leftarrow xxw.res; \\ cxwRD \Leftarrow xxw.rd \end{array}\right]$$

$$Q_1 :: \left[\begin{array}{l} a := dx.a; \\ b := dx.b; \\ cwx \Rightarrow wx; \\ resA := wx.res; \\ resB := wx.res; \\ rs1 := dx.rs1; \\ rs2 := dx.rs2; \\ rd := wx.rd; \\ selectA := (rs1 = rd); \\ selectB := (rs2 = rd); \\ selA := selectA; \\ \textbf{if } selA \textbf{ then } [outA := resA] \textbf{ else } [outA := a]; \\ aluA := outA; \\ selB := selectB; \\ \textbf{if } selB \textbf{ then } [outB := resB] \textbf{ else } [outB := b]; \\ aluB := outB; \\ func := dx.func; \\ xxw.w := dx.w; \\ xxw.rd := dx.rd; \\ cdxW \Rightarrow dx.w; \\ cdxA \Rightarrow dx.a; \\ cdxB \Rightarrow dx.b; \\ cdxRS1 \Rightarrow dx.rs1; \\ cdxRS2 \Rightarrow dx.rs2; \\ cdxFUNC \Rightarrow dx.func; \\ cdxRD \Rightarrow dx.rd; \\ cxwW \Leftarrow xxw.w; \\ cxwRD \Leftarrow xxw.rd \end{array}\right]$$

The details of the reductions can be found in section C.2 of appendix C.

Induction Step:

Applying a similar process to $Q_1; E_0$ the following equivalence is obtained:

$$Q_1; E_0 \ =_{\mathcal{O}} \ [EX_{seq\_unh\_body}]^{n-1}; Q_1; E_0$$

After $n-1$ unfoldings, the following form will be obtained from equivalence 6.25:

$$EX_{par} \ =_{\mathcal{O}} \ [EX_{seq\_unh\_body}]^{n}; Q_1; E_0$$

The equivalence for programs of length $l = n$ may be reduced to:

$$EX_{par} \ =_{\mathcal{O}} \ [EX_{seq\_unh\_body}]^{n}$$

6.3.7 Proof of EX_{seq_unh}

The interface equivalence, 6.5 of subsection 6.3.1, will be proved:

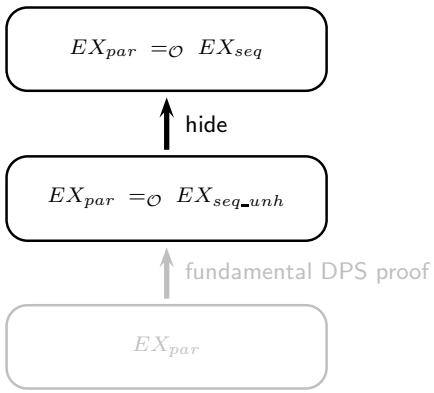
$$[(cxwW, cxwRES, cxwRD) := EX_{seq\_unh}(cinEX)]$$

$$=_{\mathcal{O}}$$

$$[cxw := EX_{seq}(cdx, cwx)]$$

where $cinEX$ can be found in the previous subsection.

This step corresponds to the highlighted part of the following portion of figure 6.8:



EX_{seq} is shown in subsection 6.2.4.3. In this step the *hide* function, subsection 3.3.4, is applied to EX_{seq_unh}, 186, with the following grouping channel correspondences:

$$\begin{array}{lcl} cxw & \longleftarrow & (cxwW, cxwRES, cxwRD) \\ cdx & \longleftarrow & (cdxW, cdxA, cdxB, cdxRS1, cdxRS2, cdxFUNC, cdxRD) \end{array}$$

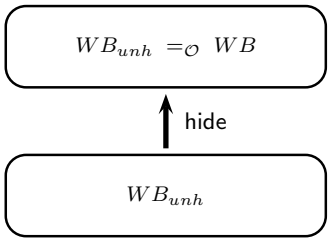
The type of cwd is Typ_XW, which matches the cartesian product of the types of $cwdW$, $cwdRES$, $cwdRD$. cdx is of type Typ_DX matching the product of the types of $cdxW$, $cdxA$, $cdxB$, $cdxRS1$, $cdxRS2$, $cdxFUNC$, $cdxRD$ channels.

6.3.8 Proof of WB_{unh}

Last step establishes the equivalence 6.6 of subsection 6.3.1:

$$\begin{array}{c} (cwdW, cwdRES, cwdRD, cwx) := WB_{unh}(cxwW, cxwRES, cxwRD) \\ =_{\mathcal{O}} \\ (cwd, cwx) := WB(cxw) \end{array}$$

The following part of figure 6.8 corresponds to this proof step:



WB_{unh} is shown in subsection 6.2.3.5 and WB in subsection 6.2.4.4. The *hide* function is applied to WB_{unh} with the following grouping channel correspondences:

$$\begin{array}{lcl} cwd & \longleftarrow & (cwdW, cwdRES, cwdRD) \\ cxw & \longleftarrow & (cxwW, cxwRES, cxwRD) \end{array}$$

The type of cwd and cxw is Typ_XW, which matches the cartesian product of the types of $cwdW$, $cwdRES$, $cwdRD$.

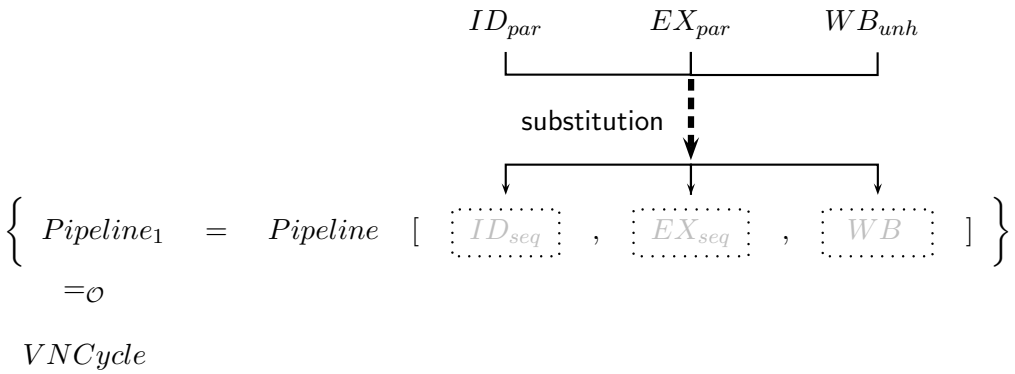
6.3.9 Final Step. Application of the Substitution Rule

The substitution can take place only when all the previous interface equivalences have been proved. On the one hand, the equivalence:

$$[reg := Pipeline_1(reg, mem)] =_{\mathcal{O}} [reg := VNCycle(reg, mem)]$$

where $Pipeline_1 = Pipeline[ID_{seq}, EX_{seq}, WB]$ was proved in subsection 6.3.3

and, on the other hand, the equivalences between parallel and sequential implementations of the pipeline stages were also proved. The substitution in parallelism, lemma 2 of page 37, allows the replacement of all sequential stages by their parallel equivalent ones, assuming that the deadlock-freeness condition holds. Note that the substitution must replace ID_{seq}, EX_{seq}, and WB at once. Next schema shows it:



After the substitution, obtaining the following equivalence:

$$\left\{ Pipeline_2 = Pipeline [ID_{par} , EX_{par} , WB_{unh}] \right\}$$

$$=_{\mathcal{O}}$$

$$VNCycle$$

In the above, procedure $Pipeline$, with references to ID_{par}, EX_{par}, and WB_{unh} denotes $Pipeline_2$, which is equivalent to $VNCycle$. The global result now is established proving the goal interface equivalence, 6.1:

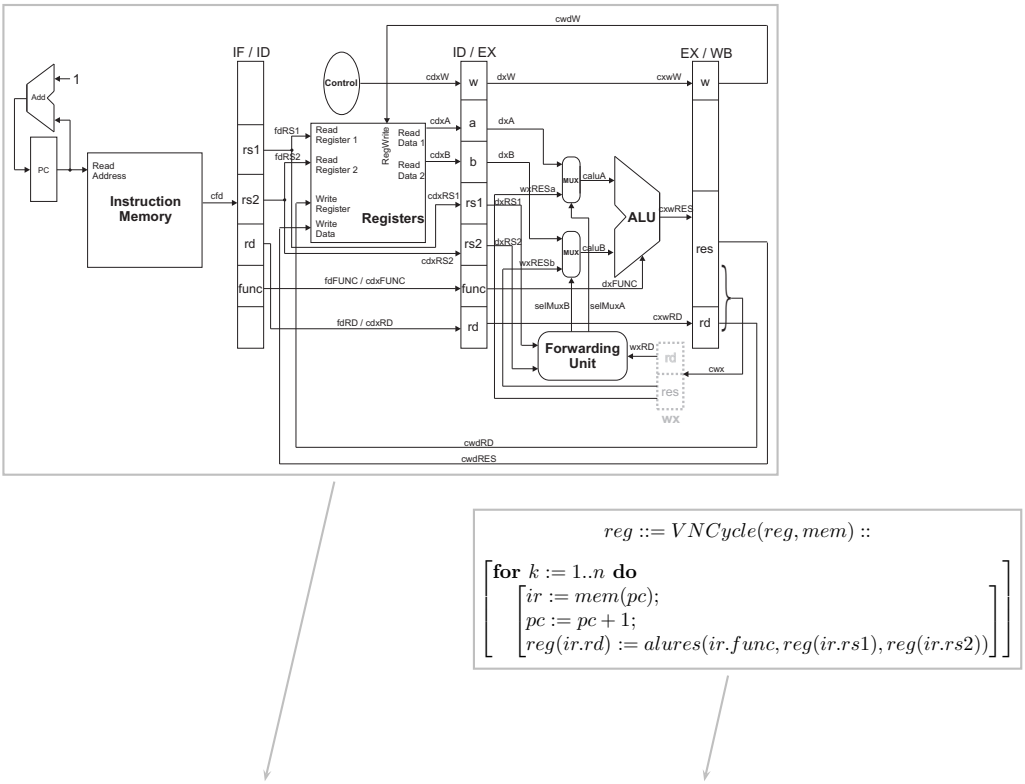
$$[reg := Pipeline_2(reg, mem)] =_{\mathcal{O}} [reg := VNCycle(reg, mem)]$$

valid for programs of finite length.

6.4 Conclusion

In this chapter a complete equivalence proof of a pipelined processor model have been presented. A non trivial model with forwarding circuits, inner parallelism, and multiple communication statements, $Pipeline_2$, has been proved equivalent to a simple sequential *Von Neumann* processor model, $VNCycle$, which has neither inner parallelism nor communications. The equivalence is valid only when both models execute finite length programs of arithmetical register-register instructions stored in the instruction memory.

The following figure illustrates the equivalence:



$$reg ::= VNCycle(reg, mem) ::$$

$$\left[\begin{array}{l}\textbf{for } k := 1..n \textbf{ do} \\ \quad \left[\begin{array}{l} ir := mem(pc); \\ pc := pc + 1; \\ reg(ir.rd) := alures(ir.func, reg(ir.rs1), reg(ir.rs2)) \end{array}\right] \end{array}\right]$$

$$[reg := Pipeline_2(reg, mem)] \ =_{\mathcal{O}} \ [reg := VNCycle(reg, mem)]$$

where the observed set, $\mathcal{O}$, is $\{reg, mem\}$.

Since the program stored in $mem(\cdot)$ never changes, the equivalence indicates that the histories of the value changes of the registers $reg(\cdot)$ of both models are the same. Consequently, for the same initial values of the registers, and the same finite length programs, stored in *mem*, two equivalent models end with the same register file values.

In going from the left to the right hand side of the equivalence, the achieved reduction on the upper bound on the state vector was 2^{-672}, as subsection 6.3.2 has shown.

Chapter 7

CONCLUSIONS AND FUTURE WORK

7.1 Summary and Conclusions

Having completed the detailed presentation of the results of this dissertation, a summary of what has been accomplished, and of what remains to be undertaken, is in order. This section concentrates on the former; pending work is reviewed in next section. The thesis dealt with formal equivalence transformations of distributed and parallel system models: communication elimination, sequentialization, and redundant variable elimination, all with a simplification aim. These where viewed as equivalence proofs constructed via application of laws as reductions in a system model rewriting process. Systems were expressed in an imperative notation with explicit parallelism, cooperation, synchronous communication statements, and storage variables. Chapter 2 introduced the two notations which have been used throughout the dissertation, a variant of SPL and PADD. Two syntactic forms with the same semantics.

7.1.1 Ground Notions

The theory needed to develop the proper dissertation results, has been summarized, without proofs, in chapters 2 and 3. It includes a suitable equivalence between statements, a set of basic program transformation laws for this equivalence, a procedure reference unhiding rule, and a procedure reference substitution rule. A semantics for the notation forms the ground layer for this theory. It is a small extension of the Manna and Pnueli fair transition systems. It makes explicit, in both computations and reduced behaviors, the values communicated via synchronous channels. These two notions become interface computations and interface behaviors in the extension. Altogether this has been overviewed in chapter 2. Some other notions, in the same packet, summarized in chapter 3, are hierarchical proofs, organized around modular procedures, and the possibility of hiding a set of channels under a common abstract

channel. Both have been used in the pipelined processor sequentialization proof.

7.1.2 Applicability Conditions for the Laws

For general system models, an infinite set of proper communication elimination laws is required. These are collected in a recursive schema. A contribution of this dissertation, which is elaborated in chapter 3, has been the formulation of *applicability conditions* of the schema of proper communication elimination laws. They are necessary in order not to introduce deadlock, since these laws have to sequentialize certain parallel substatements in some situations as the price for eliminating inner communication substatements.

The structure of the conditions reflects the recursive nature of the communication elimination law schema. Pairs of substatements have to be examined for both the presence and the order of communication operations within them. For certain given substatement at one level, substatements at all the levels have to be examined.

7.1.3 Communication Elimination Procedures

A procedure for the *elimination of a single pair* of matching communications from a *bounded communication* (BC) statement has been developed as another contribution. It transforms the statement only by the application of laws as reductions. It determines the order of the proper communication elimination law required by the statement and the pair, transforms the statement for structure matching via the introduction of nil statements, and checks the applicability conditions. It has been used within another procedure for the automatic *elimination of all matching pairs* of a statement. It tries to find a suitable sequence of laws which, applied as reductions of a rewriting process, eliminates all the inner communication operations from a given BC statement, whose inner communications are outside the scope of selection statements. It has been proved that when the procedure terminates, in the sense that all the required applicability conditions hold, deadlock freeness of the original program can be decided from the termination state. When the procedure does not terminate nothing can be said about this question. Although the procedure to eliminate a single pair was covered in chapter 4, the iterative procedure to eliminate all pairs of matching communication operations is dealt with in chapters 3 and 4. It has polynomial complexity, as shown in the main text.

7.1.4 An Interactive Prover

Communication elimination and sequentialization proofs are too cumbersome to be carried out manually. The help of a suitable tool is needed. Therefore, as another contribution, an *interactive tool* for the construction of these proofs has been developed. Chapter 5 is devoted to it. The prover guarantees that transformations conform to the accepted laws; it has commands for the invocation of complex procedures, such as communication elimination, and for simpler transformation procedures as well.

A set of such procedures carries out permutations on k-ary parallelisms, associations, flattening of parallelisms with associations in them, etc. The basic transformation with a non-communication elimination law is carried out with procedure *apply*. The prover can be expanded with relative ease, thanks to its modular design.

7.1.5 An Example of Sequentialization Proof

As an application, a *sequentialization proof of a pipelined DLX processor model*, incorporating four stages and forwarding circuits, has been constructed with the help of the interactive prover. The proof involves proof decomposition around procedures, channel hiding under abstract channels, the communication elimination procedure, sequentialization, and redundant variable elimination; all of these items have been either overviewed or summarized in the main text. The sequential model obtained after the proof, which is interface equivalent to the original pipelined model, with respect to the processor registers as the observed set $\mathcal{O}$, is a simple loop whose body has the four steps of a processor operating on its register file only: instruction fetch, operand register read, ALU operation, and writing the result into the destination register; a very intuitive equivalence. This is the function of a correct pipeline implementation of a processor. The reduction of the upper bound on the number of states of the model, attained in this transformation, is of 2^{-672}, a quite impressive result.

7.1.6 Difficulty of Sequentialization

A general observation in DPS proofs has been the size of the statement resulting from the communication elimination stage. The pipelined processor proof is an example. It is believed, however, that automatic construction of the needed simplification parts of proofs will be possible in many cases; as it seems to be for the same example. Nevertheless new examples will need new automatic simplification procedures, with varying goals. Will this go on forever? This touches the issue of completeness, which has not been even attempted in this work.

7.1.7 A Final Concluding Word

This dissertation has contributed to essential formalization aspects of communication elimination and distributed program sequentialization proofs, and has made them possible in practice; for statements whose inner communications are not selection embedded. It has allowed to envisage that further automatic proof construction procedures are possible, and has also established the ground layer for the development of general communication elimination and sequentialization automatic proof construction for statements whose inner communications are selection embedded.

In spite of the huge amount of effort already devoted to the area, before and during this thesis, much work still needs to be done until communication elimination and sequentialization become a practical method. Nevertheless the results of this thesis have enlarged its foundations and given the necessary encouragement for continuing the effort.

7.2 Future Work

7.2.1 Further Automation of DPS Proofs

The communication elimination step of DPS has already been automatized; but the remaining parts are also candidates for automatization. For instance, TPs combining parallelism to concatenation transformations with redundant variable elimination would be applicable to partially automatize some steps of the DLX processor proof. Some of the possibilities are outlined in this section. Using them, the fundamental DPS proof of Step 4, figure 6.8 of page 158, would be carried out with about ten interactive steps.

7.2.1.1 Cooperation Substatement Closening

This transformation procedure would try to mechanize the multiple applications of the *Cooperation to Concatenation* reduction, as in step (iv) of the proof of *Pipeline*1 of page 169. The goal would be to reach a predetermined sequential form starting from a parallel one, trying to eliminate parallelism from the original statement.

Basically the user would indicate a pair of statements within a statement formed with cooperation and concatenations. The transformation procedure would apply only *Cooperation to Concatenation* reductions in order to reach a sequential composition, where the two statements of the pair are closer in sequence and the inner parallelism is removed.

The transformation would proceed as far as the parallel substatements are dis-

joint and have no inner communication statements.

Procedure COOPCLOSEN – cooperation substatement closening

Input: S is the input statement in expanded PADD notation, having parallelism and concatenation top substatements only,

a is the point of the first statement of the pair within S,

b is the point of the second statement of the pair within S,

p is the desired sequential order: $[a;\ ...;\ b]$ or $[b;\ ...;\ a]$, this order can not be necessarily achieved in all situations.

Output: S', a statement without cooperation operators equivalent to S after applying several *Cooperation to Concatenation* reductions to it. So that statements a and b are closer in sequence within S', and in the sequential order indicated by p.

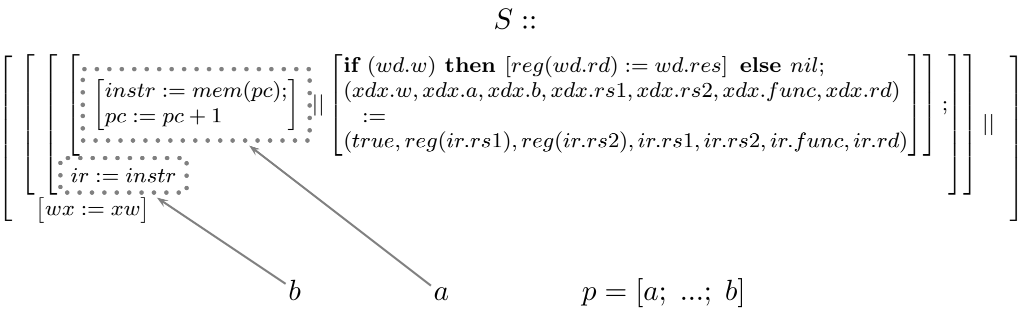
The cooperation to concatenation laws are applied whenever the disjointness and non-communication applicability conditions are fulfilled.

The desired order of a and b is achieved only if it is possible.

The position of these two statements within the obtained sequential form does not matter.

When no transformation can be applied no transformation is carried out, and a failure indication is reported.

For instance, step (iv) of *Pipeline*1 proof, page 169, now could be reduced in one step applying COOPCLOSEN transformation procedure, where the inputs are:



after the reduction, the following sequential form would be obtained.

$$S' ::$$

$$\left[\begin{array}{l} wx := xw; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ a \quad instr := mem(pc); \\ \quad pc := pc + 1; \\ b \quad ir := instr; \end{array}\right]$$

Observe that the obtained S', statements a and b are in the desired sequential order, as parameter p establishes, and the inner parallelism is removed. In this example the transformation applies twice *Cooperation to Concatenation* reduction.

7.2.1.2 Concatenation Substatement Closening

The multiple applications of *Concatenation Commutativity* reduction of step (v) of the proof of *Pipeline*1 of page 172, can be mechanized in a transformation procedure. The concatenation closening transformation procedure would try that two statements of sequential composition become closer. This is possible since most of the statements of the initial sequential form are disjoint, and simple concatenation permutation rules can be applied.

The user would be prompted to indicate two sequential statements within the given input cooperation statement, and as a result a new sequential form would be obtained. The transformation would have applicability conditions. The statements to be permuted would have to be disjoint and have no communication statements.

Procedure CONCATCLOSEN − concatenation substatement closening

Input: S is the input statement in expanded PADD notation,
a is the point of the first statement within S,
b is the point of the second statement within S.

Output: S', a sequential composition equivalent to S after applying several *Concatenation Commutativity* reductions to it, so that statements a and b are closer in sequence within S', The concatenation commutativity laws are applied whenever the disjointness applicability conditions are fulfilled.
When no transformation can be applied no transformation is carried out, and a failure indication is reported.

7.2.1.3 Iterative Redundant Variable Elimination

The redundant variable elimination step, for instance in the proof of *Pipeline*1 of page 173 (step (vi)), is another candidate to be mechanized. In our proofs, after applying communication elimination, some redundant variable assignments appear in the resulting form. These can be eliminated applying *variable and assignment elim-intro* law 13, page 40, and obtaining a simpler form.

The procedure, shown below, searches assignments within S by calling iteratively procedure $NextVarAssign$ which would search for the next assignment as a candidate to be removed and would return p, the point to the assignment. Next it would apply $VarElim$ transformation procedure, *Elimination of Redundant Variables* of section 5.4, page 138, which would try to remove it and, if successful, would obtain an equivalent statement S'.

Procedure ITEVARELIM − iterative redundant variable elimination

Input: S is the input statement in expanded PADD notation.

Output: S' a statement equivalent to S, where some of the redundant variable assignments within S have been removed.

When an assignment can not be eliminated, VARELIM exits with failure, but the procedure loops and searches for the next one.

$S' := S$
$(p, existp) :=$ NEXTVARASSIGN(S')
while *existp* **do**
  $(S', failure) :=$ VARELIM(S', p)
  $(p, existp) :=$ NEXTVARASSIGN(S')

7.2.1.4 General Substatement Closening

A higher level transformation procedure could be defined on top of the above TP's. It would automatize the last DPS steps, those applied after the communication elimination: cooperation to concatenation, concatenation commutativity, and redundant variable elimination.

The goal of this new TP would be to reach a predetermined simple sequential form. Given two substatements within an input statement, the TP would try to put these two substatements closer in sequence and in a predetermined order. It would apply iteratively cooperation and concatenation closening, and simple simplification TP's until the desired sequential form is reached. Simple simplification would

apply iterative redundant variable elimination, and basic arithmetic and boolean expression simplifications.

7.2.2 Further Generalization of Communication Elimination Laws

7.2.2.1 Introduction

The communication elimination law described in chapter 3 is not the unique elimination law, there exist more general laws; nevertheless they are more complex. Basically the communication elimination laws are based on the definition of the standard form of top statements, lemma 4 of page 45. It could be generalizated by partitioning the P statements, as the next subsection details.

7.2.2.2 Top Statements with P Partition

The form of the top level statements of lemma 4 is:

$$G_k^x = H_{k-1}^x; [G_{k-1}^x || P_{k-1}^x]; T_{k-1}^x$$

This is used successfully in many examples, but there are other possibilities. The recursive definition of the elimination laws, is shown below for an arbitrary $k \geq 0$:

$$\left[\begin{array}{c} H_k^l; \\ [\ G_k^l \ ||\ P_k^l\ ]; \\ T_k^l \end{array}\right] \ || \ \left[\begin{array}{c} H_k^r; \\ [\ G_k^r \ ||\ P_k^r\ ]; \\ T_k^r \end{array}\right] =_{\mathcal{O}} \left[\begin{array}{c} [\ H_k^l \ ||\ H_k^r\ ]; \\ [\ G_k \ ||\ P_k^l \ ||\ P_k^r\ ]; \\ [\ T_k^l \ ||\ T_k^r\ ] \end{array}\right]$$

Observe that P_k^l and T_k^r are in parallel in the l.h.s. of the equivalence, but not in the other side. In case of P_k^l and T_k^r communicate, the above laws can not be applied due to applicability conditions not being satisfied.

In some situations where statements P and T may communicate, P may be partitioned as a binary concatenation as:

$$P_k^l =_{\mathcal{O}} [\ P_k^{ml};\ P_k^{tl}\ ] \qquad \text{and} \qquad P_k^r =_{\mathcal{O}} [\ P_k^{mr};\ P_k^{tr}\ ]$$

Then the new elimination laws, for $k \geq 0$, become:

$$\left[\begin{array}{c} H_k^l; \\ \left[\begin{array}{ccc} G_k^l & \| & P_k^l \end{array}\right]; \\ T_k^l \end{array}\right] \| \left[\begin{array}{c} H_k^r; \\ \left[\begin{array}{ccc} G_k^r & \| & P_k^r \end{array}\right]; \\ T_k^r \end{array}\right] =_{\mathcal{O}} \left[\begin{array}{c} \left[\begin{array}{ccc} H_k^l & \| & H_k^r \end{array}\right]; \\ \left[\begin{array}{ccccc} G_k & \| & P_k^{ml} & \| & P_k^{mr} \end{array}\right]; \\ \left[\left[\begin{array}{c} P_k^{tl}; \\ T_k^l \end{array}\right] \| \left[\begin{array}{c} P_k^{tr}; \\ T_k^r \end{array}\right]\right] \end{array}\right]$$

In the above new laws, the P_k^{tl}'s, the tail parts of the P_k^l's, are in sequence with the T_k^l's and in parallel with the T_k^r's in both sides of the equivalence, while the P_k^{ml}'s, the heading part of the P_k^l's remains in parallel with the G_k^l's and in sequence with the T_k^l's and T_k^r's in the r.h.s. Similarly for the P_k^{tr}'s and P_k^{mr}'s.

These new laws are more general than Theorem 4, but they have some extra complexity. For instance, the P's would have to be partitioned. However, in some cases the binary partition does not help. The applicability conditions of the above laws, similar to Theorems 1, 2, and 3 of chapter 3, would have to be rewritten to include the new P_k^{mx} and P_k^{tx} substatements in the elimination restrictions.

Example: Given the following statement S, we would like to eliminate the communication over channel α:

$$S :: \left[\begin{array}{ccc} \left[\begin{array}{c} c_0 \Rightarrow \alpha; \\ c_1 \Rightarrow \beta \end{array}\right] & \| & \left[\begin{array}{ccc} c_2 \Leftarrow \alpha & \| & c_3 \Leftarrow \beta \end{array}\right] \end{array}\right]$$

After transforming S, the top level form of lemma 4 for $k = 0$ would be reached, obtaining S':

$$S' :: \left[\begin{array}{ccc} \left[\begin{array}{c} \textbf{nil}; \\ \left[\begin{array}{ccc} c_0 \Rightarrow \alpha & \| & \textbf{nil} \end{array}\right]; \\ c_1 \Rightarrow \beta \end{array}\right] & \| & \left[\begin{array}{c} \textbf{nil}; \\ \left[\begin{array}{ccc} c_2 \Leftarrow \alpha & \| & c_3 \Leftarrow \beta \end{array}\right]; \\ \textbf{nil} \end{array}\right] \end{array}\right]$$

Since the pair (P_0^r, T_0^l) communicates, the elimination law of chapter 3 could not be applied due to the applicability conditions, Theorem 1.

In this case, we could apply the new communication elimination law introduced in this subsection. After the P partition, the new top level form would be the following:

$$S'' :: \left[\begin{array}{ccc} \left[\begin{array}{c} \textbf{nil}; \\ \left[\begin{array}{ccc} c_0 \Rightarrow \alpha & \| & \left[\begin{array}{c} \textbf{nil}; \\ \textbf{nil} \end{array}\right] \end{array}\right]; \\ c_1 \Rightarrow \beta \end{array}\right] & \| & \left[\begin{array}{c} \textbf{nil}; \\ \left[\begin{array}{ccc} c_2 \Leftarrow \alpha & \| & \left[\begin{array}{c} \textbf{nil}; \\ c_3 \Leftarrow \beta \end{array}\right] \end{array}\right]; \\ \textbf{nil} \end{array}\right] \end{array}\right]$$

Now channel α would be eliminated applying the new laws.

The transformation obtained with the new laws would be:

$$S''' :: \left[\begin{array}{c} \left[\ \mathbf{nil} \quad || \quad \mathbf{nil}\ \right]; \\ \\ \left[\ c_2 := c_0 \ \ || \ \ \mathbf{nil} \ \ || \ \ \mathbf{nil}\ \right]; \\ \\ \left[\left[\begin{array}{l} \mathbf{nil}; \\ c_1 \Rightarrow \beta \end{array}\right] \ || \ \left[\begin{array}{l} c_3 \Leftarrow \beta; \\ \mathbf{nil} \end{array}\right]\right] \end{array}\right]$$

Observe that in the above form P_0^{tr} and T_0^l remain in parallel after the elimination, so they could communicate in the resulting form and no deadlock would be introduced. After simplifying the **nil** statements, one obtains:

$$S'''' :: \left[\begin{array}{c} \left[\ c_2 := c_0\ \right]; \\ \\ \left[\ c_1 \Rightarrow \beta \ \ || \ \ c_3 \Leftarrow \beta\ \right] \end{array}\right]$$

7.2.2.3 Alternative Construction of Top Statements

The construction of the standard forms of the top statements, G^l and G^r, is defined in Theorem 5 of page 54. Basically it adds outermost layers of **nil** statements around the G statements. This is not the only possible construction, for instance an innermost insertion also could be applied, as the next example illustrates.

Example: Given the following statements in parallel:

$$S_l :: \left[\begin{array}{c} H_1^l; \\ \\ \left[\begin{array}{c} \left[\begin{array}{c} H_0^l; \\ \left[\ c_1 \Rightarrow \alpha \ \ || \ \ P_0^l\ \right]; \\ T_0^l \end{array}\right] \ || \ P_1^l \end{array}\right]; \\ \\ T_1^l \end{array}\right] \ || \ S_r :: \left[\begin{array}{c} H_0^r; \\ \left[\ c_2 \Leftarrow \alpha \ \ || \ P_0^r\ \right]; \\ T_0^r \end{array}\right]$$

The goal is the elimination of the matching pair over channel α. S_l has a form of order 2 and S_r of order 1, then we make $n = 2$ and construct G_2^l and G_2^r. In this case $G_2^l = S_l$, and by inserting one innermost layer of **nil** statements we would obtain G_2^r from S_r:

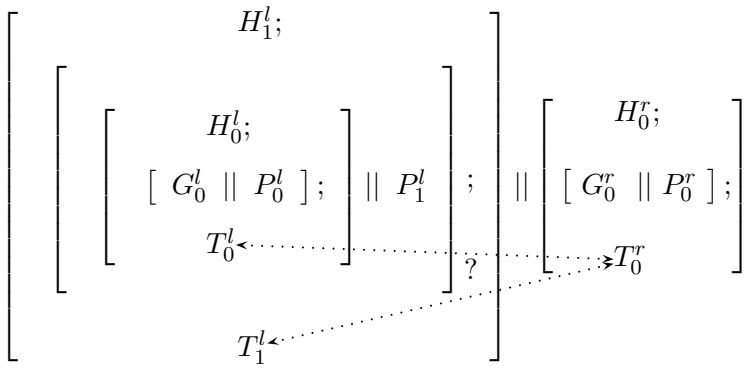
$$G_2^r :: \left\lfloor \begin{array}{c} H_0^r; \\ \left\lfloor \begin{array}{c} \left\lfloor \begin{array}{c} nil; \\ \left[\; c_2 \Leftarrow \alpha \;\|\; nil \;\right]; \\ nil \end{array} \right\rfloor \;\|\; P_0^r \end{array} \right\rfloor; \\ T_0^l \end{array} \right\rfloor$$

Now we would apply the communication elimination to $G_2^l \parallel G_2^r$, obtaining the following:

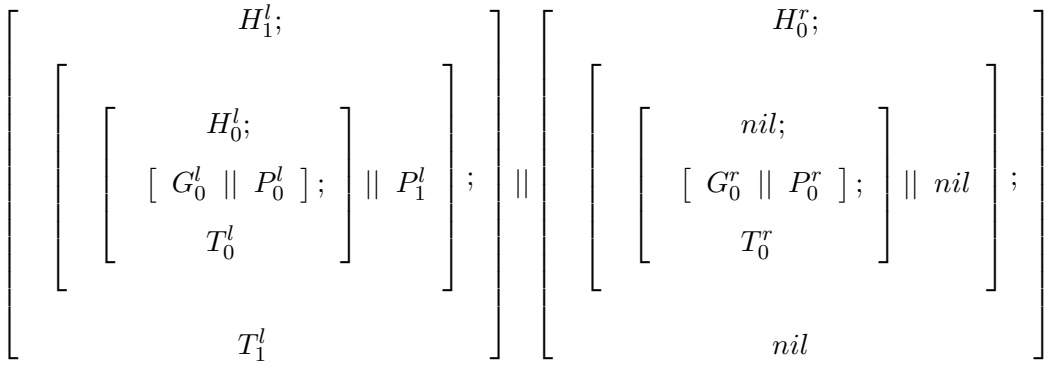
$$\left\lfloor \begin{array}{c} \left[\; H_1^l \;\|\; H_0^r \;\right]; \\ \left\lfloor \begin{array}{c} \left\lfloor \begin{array}{c} \left[\; H_0^l \;\|\; nil \;\right]; \\ \left[\; c_2 := c_1 \;\|\; P_0^l \;\|\; nil \;\right]; \\ \left[\; T_0^l \;\|\; nil \;\right] \end{array} \right\rfloor \;\|\; P_1^l \;\|\; P_0^r \end{array} \right\rfloor; \\ \left[\; T_1^l \;\|\; T_0^r \;\right] \end{array} \right\rfloor$$

The difference between the outermost addition and the innermost insertion lies mainly in the construction of the G top statements. Procedure BIN-COMELI has been implemented as a special case of Theorem 5, which adds outermost layers. However, for orders $n > 1$, there are many possible ways of inserting **nil** statements. All would be valid if they satisfy lemma 4 and the applicability conditions of Theorem 5.

In general the insertion of **nil** statements would depend on which pairs of T statements, (T_i^l, T_j^r) where $i, j \in [0, n-1]$, communicate. For instance, in the example below, if T_0^r communicates with T_0^l then an outermost *nil* addition would be the right choice. On the other hand, if T_0^r communicates with T_1^l then an innermost *nil* insertion would be the right choice for the applicability conditions to hold.



In case of T_0^l communicates with T_0^r, we would construct the top statements of the above example as follows:



This construction combines outermost and innermost **nil** insertion. This is one of the possible ways to achieve the communication elimination when the applicability conditions of Theorem 5 are not satisfied. With this construction the communication pair could be eliminated.

7.2.3 Deadlock Situations

The communication elimination can not be applied always. Sometimes it fails due to the applicability conditions, which guarantee that no deadlock is introduced after the elimination.

The communication elimination procedure, *PElim* of page 55, can end in one of the following situations:

a) Successful termination. Then the initial statement S is deadlock-free.

b) Unsuccessful termination. Then there is still some communication left in the communication front of S, $\textsc{ComFront}(I, S)$. The remaining unmatched communication statements indicate that the initial statement S is not deadlock-free.

c) Exit with failure. Then the applicability conditions are not fulfilled. The resulting statement would deadlock after applying the elimination. Nothing can be decided from initial model deadlock-freeness.

Currently, failure may indicate either that the input statement deadlocks or that after applying the law a deadlock would be introduced in the resulting statement. Both situations are not distinguished now. For instance, the following statement S_1 deadlocks:

$$S_1 :: \left[\begin{array}{c} H_0^l; \\ \left[\begin{array}{c} \alpha \Leftarrow a \end{array}\right] \;\|\; \left[\begin{array}{c} \beta \Leftarrow b; \\ \gamma \Leftarrow c \end{array}\right] \end{array}\right]; \;\|\; \left[\begin{array}{c} H_0^r; \\ \left[\begin{array}{c} \alpha \Rightarrow d \end{array}\right] \;\|\; \left[\begin{array}{c} \gamma \Rightarrow e \end{array}\right]; \\ \beta \Rightarrow f \end{array}\right]$$

where H_0^l, H_0^r, and T_0^l do not have any communication statement. The deadlock arises when $\beta \Leftarrow b$ tries to communicate with $\beta \Rightarrow f$, and $\gamma \Leftarrow c$ with $\gamma \Rightarrow e$. In the above example, one of the applicability conditions of procedure BIN-COMELI, page 66, checks only that P_0^l does not communicates with T_0^r. Adding the following condition:

$\textsc{commprecede}(P_0^l, T_0^r, P_0^r)$

where $\quad P_0^l :: \left[\begin{array}{c} \beta \Leftarrow b; \\ \gamma \Leftarrow c \end{array}\right], \quad P_0^r :: \left[\, \gamma \Rightarrow e \right] \quad$ and $\quad T_0^r :: \left[\, \beta \Rightarrow f \right]$.

Procedure COMMPRECEDE, page 76, checks whether within P_0^l, the communications with T_0^r precede the communications with P_0^r, and decides failure in the law application. Actually, deadlock in the initial model could be decided with a more elaborated processing.

This condition suffices to detect a deadlock situation. In general, we should further study how to detect deadlock in input statements. Based on the whole structure of the top parallel statements, new set of *deadlock conditions* could be defined. They would warn about these anomalous situations.

7.2.4 Elimination of Communications within Selection Scopes and Non-disjoint Pairs

The communication elimination procedure, *PElim* of page 55, does not apply when the synchronous communications to be eliminated are under the scope of selection statements neither when the competing pairs, of the same front, share a communication statement, thus not being disjoint. This subsection introduces briefly some notions needed to handle these cases. Their solution endows the application domain of formal sequentialization with more generality.

The new communication elimination procedure would be implemented on top of *PElim*. It would try to eliminate inner communications which may be under the scope of selections and with possibly non-disjoint communication pairs. The goal would be to build a top selection composition whose alternatives would be selection-free BCS, where procedure *PElim* would be applied. In general, there are three possible location cases for a communication c of a pair p in *CompPairs(I,S)* within the scope of a selection construct:

a) as a guard of a communications selection

b) within an alternative, $A[c]$, of a communications selection whose communication guard is an external communication ext

c) within an alternative, $A[c]$, of a normal selection

where the communication c has the forms $\alpha \Rightarrow v$ or $\alpha \Leftarrow v$.

Corresponding to the above three situations, a new set of elimination laws should be defined. An example of the first case is:

$$\left[\cdots \alpha_1 \Leftarrow v_1; \cdots \;||\; \cdots [b_1, \alpha_1 \Rightarrow v_2; A_1 \;\; \textbf{or} \;\; b_2, \alpha_2 \Leftarrow v_3; A_2]; \cdots \;||\; \cdots \alpha_2 \Rightarrow v_4 \cdots\right]$$

where the b_i's are boolean guards, and the communications via channels α_1 and α_2 stand as guards of a communications selection. The expression has two distinct competing pairs : $(\alpha_1 \Leftarrow v_1\ ,\ \alpha_1 \Rightarrow v_2)$ and $(\alpha_2 \Leftarrow v_3\ ,\ \alpha_2 \Rightarrow v_4)$.

The following is an example of the second case:

$$\left[\cdots \alpha_1 \Leftarrow v_1; \cdots \;||\; \cdots [b_1, cext \Rightarrow v_2; \cdots \alpha_1 \Rightarrow v_3 \cdots \;\; \textbf{or} \;\; R]; \cdots \;||\; \cdots\right]$$

where R stands for the rest of the selection statement. Here the pair is over channel α_1.

For the third case one has the statement:

$$\left[\cdots \alpha_1 \Leftarrow v_1; \cdots \;||\; \cdots [b_1, \cdots \alpha_1 \Rightarrow v_2 \cdots \;\; \textbf{or} \;\; R]; \cdots \;||\; \cdots\right]$$

7.2.4.1 Example

As an illustration of the elimination under selections, the following procedure models a step of a stop and wait communications protocol:

$$(\alpha, ack) ::= Step\ (m)\ ::\ \left[\begin{array}{l} \textbf{local}\quad \delta,\ \eta : \quad \textbf{channel of message} \\ \textbf{local}\quad \gamma \qquad\ \textbf{channel of boolean} \\ \textbf{local}\quad \varepsilon : \qquad \textbf{channel of nil} \\ \\ \quad Emitter :: \big[\eta \Leftarrow m; \gamma \Rightarrow ack\big] \\ \quad || \\ \quad DataChannel :: \left[\begin{array}{ll} \textbf{local} & d : \quad \textbf{message} \\ \eta \Rightarrow d; & \left[\begin{array}{l} true, \delta \Leftarrow d; \textbf{nil} \\ \textbf{or} \\ true, \varepsilon \Leftarrow; \textbf{nil} \end{array}\right] \end{array}\right] \\ \quad || \\ \quad Receiver :: \left[\begin{array}{l} \textbf{local}\quad r : \quad \textbf{message} \\ \quad \big[true, \delta \Rightarrow r; \alpha \Leftarrow r; \gamma \Leftarrow T\big] \\ \quad \textbf{or} \\ \quad \big[true, \varepsilon \Rightarrow; \gamma \Leftarrow F\big] \end{array}\right] \end{array}\right]$$

There are three statements connected in parallel: *Emitter*, *DataChannel*, and *Receiver*. The message to be sent is input to *Step* in variable m of global memory within *Emitter*, which sends it to *DataChannel* via η and waits on channel γ for acknowledgement. The message is delivered to *Receiver* via channel δ. A transmission error is simulated, non-deterministically, by communicating to *Receiver* through channel ε of type **nil**. Only the implicit synchronization suffices, no value passing is needed. The two options are the alternatives of the communications selection within *DataChannel*, matched by a communications selection within *Receiver*. After outputting the message from *Step* via channel α, *Receiver* acknowledges to *Emitter* by passing a true value through channel γ. In case of an erroneous reception, a false value is sent instead. This boolean is stored in variable *ack* of global memory as a result of *Step*.

The interface set of the above example, *Step*, would be $\mathcal{O} : \{\alpha, ack, m\}$. The equivalent program resulting from a DPS proof would be the following:

$$(\alpha, ack) ::= SimpleStep\ (m)\ ::\ \left[\begin{array}{l} \quad \big[true, [\alpha \Leftarrow m; ack := T]\big] \\ \textbf{or} \\ \quad \big[true, [ack := F]\big] \end{array}\right]$$

Observe that parallelism and inner channels, δ, η, γ, and ε, have been eliminated, only remains a selection top statement as desired.

The overall equivalence would be: $Step =_{\{\alpha, ack, m\}} SimpleStep$.

7.2.4.2 Difficulties of general elimination

In order to illustrate the difficulties of the elimination within selection scopes, the following statement, *UndSS*:

$$
UndSS :: \left[\begin{array}{l}
\left[\begin{array}{l} H_1; \\ \left[\begin{array}{l}[b_1, c_1 \Rightarrow; A_1] \\ \textbf{or} \\ [b_2, c_2 \Leftarrow; A_2]\end{array}\right]; \\ T_1\end{array}\right] \\
\quad || \\
\left[\begin{array}{l} H_2; \\ \left[\begin{array}{l}\left[\begin{array}{l} b_3; \\ \left[\begin{array}{l} H_3; \\ \left[\begin{array}{l}[b_{31}, c_2 \Rightarrow; B_2] \\ \textbf{or} \\ [b_{32}; B_3]\end{array}\right]; \\ T_3\end{array}\right]\end{array}\right] \\ \textbf{or} \\ [b_4; H_4; c_1 \Leftarrow; B_4]\end{array}\right]; \\ T_2\end{array}\right]
\end{array}\right]
$$

is a parallelism with two top statements which have selections. The set of internal communications is $I : \{c_1, c_2\}$. The notations $c \Rightarrow$ and $c \Leftarrow$ denote send and receive over a **nil** typed channel. Neither the H_i nor B_4, T_2 and T_3 statements have internal communications. Then

$$\textit{ComFront(I,UndSS)}: \{<> c_1, []c_2, <> c_2, []c_1\}$$

and

$$\textit{CompPairs(I,UndSS)}: \{(<> c_1, []c_1), ([]c_2, <> c_2)\}.$$

Notice that if, say, T_3 had an internal communication it would be in the front when b_{32} evaluates to true and B_3 has no inner communication. The execution of the two associated communication events depends on the truth value of boolean guards b_1, b_2, b_3, b_4, b_{31} and b_{32}. Thus it could be said, using the terminology of [Tay83], that the two pairs are in the *possible rendez-vous* set. The choice of alternative in the

upper communication selection depends on both boolean guard evaluation and on the readyness of communications.

Since communication events depend on boolean guard evaluation, deadlock also depends on that, adding an extra difficulty. For instance, in *UndSS*, the state where b_1, b_3, and b_{31} are true, when control visits them, gives a deadlock when b_2 evaluates to false but does not when it evaluates to true; since then pair $([]c_2, <> c_2)$ will be always taken. Similarly for the state where b_2 and b_4 are true when b_1 is or is not true.

The following illustrates boolean deadlock for case *c*) at the beginning of section 7.2.4,

$$S_b :: [\ [\ [b_{11}, A_{11}[c_1]]\ \textbf{or}\ [b_{12}, A_{12}[c_2]]]\ ||\ [\ [b_{21}, A_{21}[\bar{c}_1]]\ \textbf{or}\ [b_{22}, A_{22}[\bar{c}_2]]]]$$

Here the choice of an alternative does not depend on the communications, but on boolean guards only. Now, boolean deadlock may take place even if one of the conditions $b_{11} \wedge b_{21}$ and $b_{12} \wedge b_{22}$ are true. For instance, when $b_{11} \wedge b_{21} \wedge b_{12} \wedge \neg b_{22}$, the left parallel process may choose non-deterministically its second alternative, the one whose guard is b_{12}, while the one at the right can only choose its first alternative, thus giving deadlock. But, in addition, when $\neg b_{21} \wedge \neg b_{22} \wedge \neg b_{12}$, the deadlock single alternative behavior $A_{11}(c_1)$ is possible. Similar behaviors may occur when $\neg b_{11} \wedge \neg b_{12}$.

In order to assess the complexity of the general elimination problem, some bounds based on the complexities of deciding whether a given pair is in the possible rendezvous set (PR set for short) of [Tay83] can be given. In that reference, the program is assumed to have only one level of parallelism, with no recursive call. This is a subclass of BC statements. Clearly, any pair in the PR set is eliminated by a communication elimination procedure. Therefore, the complexity of elimination cannot be lower than the complexities of the PR problem. Translating the results of [Tay83] to the terminology used in this thesis, the general elimination from BC statements has non-polynomial, NP-complete, complexity for the following classes of BC statement:

1. Neither branches nor communication selection substatements allowed, but allowing non-disjoint pairs.

2. Neither branches nor non-disjoint pairs allowed, but allowing communication selection substatements.

Also, linear complexity proportional to the number of basic statements (nodes in the flow-graphs of [Tay83]) is attained for BC statements with neither branches nor communication elimination substatements, and having no non-disjoint pair. Notice

that the polynomial complexity of the communication elimination procedure given in this thesis is due to the removal of the single level of parallelism restriction. Therefore, the cause of the NP complexity is the non-determinism common to cases 1 and 2 above. It is important to say that, in all these bounds, it has been assumed that the elimination algorithm terminates, all applicability conditions being met, for the classes of statements considered.

Let us turn our attention now to non-BC statements. A specific class of them has been treated in this thesis: linear recursion or, equivalently, indefinite iteration. The complexity would be polynomial for the proof of the pipelined processor. In spite of the simplicity of this case, it is encountered very often in the applications.

Unfortunately, the elimination complexity for the general class, having arbitrary structures of recursive procedure references is not decidable. This inference is based on the work reported in [Ram00]. There, as in [Tay83], the problem of statically deciding whether a pair belongs to the possible rendez-vous set is treated; and shown to be undecidable. This gives a lower-bound, as in the BC statement case, on the complexity of the general communication elimination procedure. Since, should it terminate, it would give an effective way to compute the PR set.

In spite of these complexity results, further work has to be undertaken; since most applications have relatively simple structures. In addition, deadlock situations caused by improper boolean guards are assumed to be non-existent in the current stage. All this has to be further studied. First for general BC statements with any structure of selections embedding internal communications. This will involve sequence and parallelism distributive laws, over selections. After that, the fundamental sequentialization proof for non-BC statements with linear recursion has to be analyzed.

7.2.5 Completeness

Until now, only the soundness of the laws has been established. The work is in a too early stage to attempt the study of completeness of a set of laws. More proofs have to be done, and extension to selection embedded inner communications has to be finished. Nevertheless, some comments on that will be provided in this section.

Informally, completeness may mean that for each statement belonging to a certain well defined class of statements S_P, an equivalent sequential form can always be derived applying laws belonging to a certain set L. This seems to be a sort of minimal notion, since nothing is said about the sequential form, needing to be transformed to a concrete form desired by the user. But these transformations may be treated in the pertinent literature which would have to be searched and studied.

The first difficulty is encountered in the non-termination possibility of the com-

munication elimination procedure. For which class of models is termination guaranteed? In section 7.2.2 above some changes to give more generality to the current recursive communication elimination law schema have been suggested. Therefore, the terminating class would be enlarged then.

A simple class of models for which termination seems to be guaranteed is *communication-closed-layered* models. In them communications are organized in time layers. As execution proceeds, the communication events of a layer never occur before those of preceding layers.

Assuming that the class of models, within BC statements, for which communication elimination terminates has been clarified, then obtaining a sequential form after communication elimination seems to be guaranteed, by application of parallelism to sequence transformation laws. This is so since parallel statements are disjoint. More specifically, the heading statements H, of the communication elimination law schema, are disjoint and free of internal communications. Also, the tail statements T are eventually reduced to either **nil** or to disjoint non-communicating statements by the terminating communication elimination proof construction procedure.

These preliminary thoughts pend to be validated, modified, formalized, and proved true in the course of a future effort.

Appendix A

SOUNDNESS OF THE LAWS

The soundness proofs of the basic laws given in subsections 2.9.2.1 and 2.9.2.2, with a proof of a law of the communication elimination schema, and of the unboundedness of the schema, are given in this appendix. This is carried out in the semantic framework of the Manna and Pnueli books. The justification of the need of avoiding strong fairness is also proved (Theorem 6).

The appendix contains an updated version of part of [BBCN01]. It does not belong to the thesis contribution; it has been added to make this account more self-contained.

A.1 Simple Cases

The first set of auxiliary laws is based in the following congruences. Let $p_m(k)$, where $k = 1..m$, denote the *k-th* integer of a permutation of the list $\langle 1, 2, \ldots, m \rangle$. Then:

$$\mathbf{nil}; S \approx S \qquad S; \mathbf{nil} \approx S \qquad S_1 || \ldots || S_m \;\approx\; S_{p_m(1)} || \ldots || S_{p_m(m)}$$

$$S_1; \ldots; S_k; \ldots; S_l; \ldots; S_m \;\approx\; S_1; \ldots; [S_k; \ldots; S_l]; \ldots; S_m$$

$$c_1, S_1 \;\mathbf{or}\; \ldots \;\mathbf{or}\; c_m, S_m \;\approx\; c_{p_m(1)}, S_{p_m(1)} \;\mathbf{or}\; \ldots \;\mathbf{or}\; c_{p_m(m)}, S_{p_m(m)}$$

$$\begin{array}{l} c_1, com(\alpha_1)\,, S_1 \;\mathbf{or}\; \ldots \;\mathbf{or}\; c_m, com(\alpha_m)\,, S_m \\ \approx \\ c_{p_m(1)}, com(\alpha_{p_m(1)})\,, S_{p_m(1)} \;\mathbf{or}\; \ldots \;\mathbf{or}\; c_{p_m(m)}, com(\alpha_{p_m(m)})\,, S_{p_m(m)} \end{array}$$

The justification of these congruences is simple, since from the semantic definition of the notation, given in section 2.5, it can be seen that both sides of their congruence symbols have the same associated transitions.

A.1.1 Justification of the auxiliary laws

Rules allowing the introduction and the elimination of the skip statement are needed in the applications. The following remarks make their mathematical justification hard.

Remark 1 (Skip Concatenation Non-congruences): Let S and $\tilde{S}$ be statements. Then

$$S \not\approx S;\mathbf{skip} \qquad S;\tilde{S} \not\approx S;\mathbf{skip};\tilde{S}$$

Hence, in general

$$S \not\approx \mathbf{skip};S$$

As an intuitive clue to justify this remark, deleting an skip statement may enable transitions associated with the statement which immediately follows it, particularly joint synchronous communication transitions formed with a statement parallel with it in some program context. This leads into an infinite number of enablings when the skip statement is within an infinite loop in the program context, and then some computation may be excluded from one side due to the fairness requirements with respect to the enabled transition, but not in the side where the **skip** is present.

Justification In order to prove the first relation, define the program context $P[S]$ as follows:

$$
\left[\begin{array}{l}
\textbf{local}\quad x,y,z,v\quad :\textbf{boolean where } x=\mathrm{T},y=\mathrm{T},z=\mathrm{T},v=\mathrm{T}\\
\textbf{local}\quad \alpha,\beta,\gamma,\delta\quad :\textbf{channel of boolean}\\
\\
\left[\begin{array}{l}
P_1::\left[\begin{array}{l}k_0\text{: }\textbf{while } x \textbf{ do}\\ \quad\left[\begin{array}{l}k_1\text{: } S;\ \ k_2\text{: }\left[\begin{array}{l}k_3\text{: }\beta\Rightarrow x; k_4\text{: }\alpha\Leftarrow \mathrm{F}; k_5\text{: }\gamma\Leftarrow\mathrm{F}\\ \textbf{or}\\ k_6\text{: }\gamma\Leftarrow\mathrm{T}\end{array}\right]\\ k_7\text{:}\end{array}\right];\end{array}\right]\\
\|\\
P_2::\left[\begin{array}{l}\ell_0\text{: }\textbf{while } y \textbf{ do}\\ \quad[\ell_1\text{: }\gamma\Rightarrow y]\end{array}\right]\\
\|\\
P_3::\left[\begin{array}{l}m_0\text{: }\textbf{while } z \textbf{ do}\\ \quad\left[\begin{array}{l}m_1\text{: }\alpha\Rightarrow z;\ \ m_2\text{: }\left[\begin{array}{l}m_3\text{: }\beta\Leftarrow\mathrm{F}; m_4\text{: }\alpha\Rightarrow z; m_5\text{: }\delta\Leftarrow\mathrm{F}\\ \textbf{or}\\ m_6\text{: }\delta\Leftarrow\mathrm{T}\end{array}\right]\\ m_7\text{:}\end{array}\right];\end{array}\right]\\
\|\\
P_4::\left[\begin{array}{l}n_0\text{: }\textbf{while } v \textbf{ do}\\ \quad[n_1\text{: }\delta\Rightarrow v]\end{array}\right]
\end{array}\right]
\end{array}\right]
$$

Then, all the computations of $P[\alpha \Leftarrow \mathrm{T}]$ always terminate under the fairness assumptions of section 2.5. This is due to the synchronization between P_1 and P_3 imposed by the synchronous communication via channel α. Just after this transition is taken, the joint transition between the same two processes, but via β, is enabled. Since both communication statements occur within an indefinite loop in both processes, this enabling occurs an indefinite number of times. But, since the joint transition associated with synchronous communication statements are in the compassion set (strongly fair), the synchronous communication via β has to be taken eventually. Once this occurs, the variables x, z, v, and y take the value false and the four processes terminate.

The communication via β may not occur in the program $P[\alpha \Leftarrow \mathrm{T}; \mathbf{skip}]$ since the presence of the new skip statement allows the existence of indefinite computations having no enabling of the communication via β. This is so since now, due to the skip statement, the synchronization via channel α does not necessarily activate simultaneously the control locations corresponding to labels k_2 and m_2, consequently the joint transition of the synchronous communication via channel β may never be enabled. Therefore program $P[\alpha \Leftarrow \mathrm{T}; \mathbf{skip}]$ has a non-terminating computation. This proves the first non-congruence. The other two are consequences of the first. □

Remark 2 (Skip and Nil Cooperation Non-congruences): Let S be a statement. Then

$$S \not\approx S \,\|\, \mathbf{skip} \qquad S \not\approx S \,\|\, \mathbf{nil}$$

Justification Let

$$S = \left[\begin{array}{l} k_0\text{: } \beta \Rightarrow x; k_1\text{: } \alpha \Leftarrow \mathrm{F}; k_2\text{: } \gamma \Leftarrow \mathrm{F} \\ \textbf{or} \\ k_3\text{: } \gamma \Leftarrow \mathrm{T} \end{array}\right]$$

Define the program context $P[\ \tilde{S}\ ]$ as follows:

$$\left[\begin{array}{l} \textbf{local} \quad x, y, z, v \quad : \textbf{boolean where } x = \mathrm{T}, y = \mathrm{T}, z = \mathrm{T}, v = \mathrm{T} \\ \textbf{local} \quad \alpha, \beta, \gamma, \delta \quad : \textbf{channel of boolean} \\ \\ \left[\begin{array}{l} P_1 :: \left[\begin{array}{l} k_0\text{: } \textbf{while } x \textbf{ do} \\ \quad [k_1\text{: } \alpha \Leftarrow \mathrm{T}; k_2\text{: } \tilde{S}\ ; k_3\text{: }\ ] \end{array}\right] \\ \| \\ P_2 :: \left[\begin{array}{l} \ell_0\text{: } \textbf{while } y \textbf{ do} \\ \quad [\ell_1\text{: } \gamma \Rightarrow y] \end{array}\right] \\ \| \\ P_3 :: \left[\begin{array}{l} m_0\text{: } \textbf{while } z \textbf{ do} \\ \quad \left[\begin{array}{l} m_1\text{: } \alpha \Rightarrow z;\ \ m_2\text{: } \left[\begin{array}{l} m_3\text{: } \beta \Leftarrow \mathrm{F}; m_4\text{: } \alpha \Rightarrow z; m_5\text{: } \delta \Leftarrow \mathrm{F} \\ \textbf{or} \\ m_6\text{: } \delta \Leftarrow \mathrm{T} \end{array}\right]; \\ m_7\text{:} \end{array}\right] \end{array}\right] \\ \| \\ P_4 :: \left[\begin{array}{l} n_0\text{: } \textbf{while } v \textbf{ do} \\ \quad [n_1\text{: } \delta \Rightarrow v] \end{array}\right] \end{array}\right] \end{array}\right]$$

Then $P[S]$ always terminates under the fairness assumptions of section 2.5, but $P[S||\textbf{skip}]$ and $P[S||\textbf{nil}]$ have a non-terminating computation due to the existence of the skip-type entry transition of the cooperation statement. The reasoning is similar to the one of the previous remark. □

The following two theorems are important since they identify, within the SPL notation of section 2.5, the strong fairness assumptions about communication statements as being responsible for the irregular behavior of the **skip** and **nil** statements, as in the non-congruences of the above remarks.

Theorem 6 (Concatenated Skip Deletion): Let S be a statement. Let S_{ncs} be a statement which is neither a communication selection whose communication guards are not all asynchronous sends, nor synchronous communications. Then

$$S; S_{ncs} \ \approx\ S; \textbf{skip}; S_{ncs}$$

This congruence holds without the above restrictions upon S_{ncs} when no transition associated with communication statements is in the compassion set $\mathcal{C}$.

Notice that the congruence holds when S_{ncs} is a selection all of whose guards are asynchronous sends. As appendix A.2 shows, the restrictions of the lemma avoid the activation of a front transition of S_{ncs}, in a computation of the right hand side, which is not taken in it. This would take place when the **skip** is deleted.

Theorem 7 (Parallel Skip and Nil Deletion): Let S be a statement. Let S_{ncs} and S'_{ncs} be statements which are neither communication selections whose communication guards are not all asynchronous sends, nor synchronous communications. Let $\tilde{S}$ be an arbitrary statement. Then

$$S; S_{ncs}; \tilde{S}; S'_{ncs} \;\approx\; S; [\mathbf{skip}|||[S_{ncs}; \tilde{S}]]; S'_{ncs}$$

and

$$S; S_{ncs}; \tilde{S}; S'_{ncs} \;\approx\; S; [\mathbf{nil}|||[S_{ncs}; \tilde{S}]]; S'_{ncs}$$

These congruences hold without the restrictions upon S_{ncs} when no transition associated with communication statements is in the compassion set $\mathcal{C}$.

The justification is similar to the one of the previous theorem. When deleting the parallel **skip** or **nil**, the binary cooperation disappears together with its entry and exit transitions. Here these entry and exit transitions of the cooperation statement, which are of the skip type, play the same role as the transition associated with the skip statement in the previous theorem. Notice that no restriction is needed when the order of the cooperation is greater than two. The detailed justification would follow the same reasoning given in appendix A.2 for Theorem 6.

Lemma 12 (Associativity of Cooperation): Let k and l be integers such that $1 \leq k < m$ and $1 < l \leq m$. Then

$$[S_1||\ldots||S_k||\ldots||S_l||\ldots||S_m\ ] \;\approx\; [\ S_1||\ldots||\ \ [\ S_k||\ldots||S_l\ ]\ \ ||\ldots||S_m\ ]$$

provided that the front statements of $S_k, \cdots, S_l$ are neither synchronous communication statements nor communication selection statements whose communication guards are not all asynchronous sends.

This congruence also holds without the restriction upon the front statements of $S_k, \cdots, S_l$, when no transition associated with communication statements is in the compassion set $\mathcal{C}$.

Notice that the entry and exit transitions associated with the main cooperation statement are present in the computations of both sides. However, the entry and exit transitions of the inner cooperation statement are present in one side only, therefore

in moving from one side to the other these inner skip-type transitions are deleted from the corresponding computations.

When deleting the entry transition of the inner cooperation, from a computation of the right hand side, some front transition of either $S_k, \cdots$, or S_l, which is not taken in the computation, may be activated. The outer exit transition prevents this activation when the exit transition of the inner cooperation is deleted. The detailed justification would follow the same reasoning given in appendix A.2 for Theorem 6. Nevertheless, its main line is given now.

Justification Let S^l and S^r denote the statements to the left and to the right of the general congruence of the lemma. Let τ_l^E and τ_l^X denote the entry and exit transitions of the cooperation statement S^l, and τ_r^E and τ_r^X denote the entry and exit transitions of the main cooperation statement in S^r. Let τ_{kl}^E and τ_{kl}^X denote the entry and exit transitions associated with the cooperation substatement in S^r. Let $P[S]$ be an arbitrary program context.

1. Let σ be a computation of $P[S^l]$. We construct from σ a computation σ' of $P[S^r]$ by replacing in σ any occurrence of transition τ_l^E by τ_r^E followed immediately by τ_{kl}^E. Similarly, any occurrence of transition τ_l^X in S^l is replaced by τ_{kl}^X, followed immediately by τ_r^X. This last replacement is consistent since whenever τ_l^X is enabled in σ, τ_{kl}^X should also be enabled in σ'. In this situation, after taking τ_{kl}^X, transition τ_r^X becomes enabled and can be taken immediately. Clearly, $\sigma^r = \sigma'^r$ since skip-type transitions are replaced by skip-type transitions or by two consecutive skip-type transitions and, by construction, the number of times a transition τ is enabled or taken in both σ and σ' are the same. This is due to the fact that insertion of skip-type transitions in a computation does not enable any new transition.

2. Let σ be a computation of $P[S^r]$. We construct from σ a computation σ' of $P[S^l]$ by replacing any occurrence of transition τ_r^E in σ by transition τ_l^E, and by deleting from σ every occurrence of transition τ_{kl}^E. Similarly, we delete from σ every occurrence of transition τ_{kl}^X, and we replace every occurrence of transition τ_r^X in σ by transition τ_l^X.

   We have to show that σ' satisfies the standard fairness requirements after having deleted transitions τ_{kl}^E and τ_{kl}^X. In the case of τ_{kl}^E, the reasoning leading to the impossibility of violating the standard fairness requirements would be identical to the one made in connection with the τ_ℓ skip-type transition in the skip statement deletion Theorem 6, above. Now, however, statement S_2 of appendix A.2 can be the front statement of either $S_l, \cdots$, or S_r. This justifies the restrictions upon these statements imposed in the lemma. Finally, the case where transition τ_{lk}^X is deleted is simpler since its consecutive transition,

which follow next to it in sequence, is the skip-type transition τ_r^X, which is replaced by skip-type transition τ_l^X. Since a skip-type transition can not give two front transitions, transition τ of lemma 16 of appendix A.2, which caused the problem, can not exist.

□

A.1.2 Communication Elimination Laws

After the study of a simple communication elimination law, it is proven that no finite set of laws suffices for communication elimination in general programs.

Lemma 13 (Simple Communication Elimination and Introduction): Let H^l and H^r be statements which do not have communication statements through synchronous channel α, and T^l and T^r be statements. Then

$$[H^l; \alpha \Leftarrow e; T^l]||[H^r; \alpha \Rightarrow u; T^r] \;\approx\; [H^l||H^r]; u := e; [T^l||T^r]$$

provided that either no transition of a communication statement is strongly fair or a **skip** is inserted automatically before communication or communication selection statements, as explained above in this subsection.

This congruence is a special case of the recursive schema of subsection 3.2.2 of the main text. The following congruence

$$[\alpha \Leftarrow e||\alpha \Rightarrow u] \;\approx\; u := e$$

is a special case of the lemma, obtained by making $H^l = H^r = T^l = T^r = \mathbf{nil}$. As in Theorems 6, 7, and lemma 12, the problem here is the deletion from the right hand side computations of the entry transition of the cooperation $[T^l||T^r]$.

Justification Let S^l and S^r denote the statements to the left and to the right of the congruence symbol. Let τ_l^E and τ_l^X denote the entry and exit transitions associated with the cooperation statement S^l. Let τ_h^E and τ_h^X denote the same transitions for the cooperation substatement $H^l||H^r$ of S^r. Let τ_t^E and τ_t^X denote the same transitions for the cooperation substatement $T^l||T^r$ of S^r. Let $P[S]$ be an arbitrary program context.

1. Let σ be a computation of $P[S^l]$. A computation σ' of $P[S^r]$ is constructed as follows: replace every occurrence of transitions τ_l^E and τ_l^X in σ by transitions τ_h^E and τ_t^X in σ', respectively. We replace every occurrence of joint transition $\tau_{<\alpha>}$ in σ by transitions τ_h^X, $\tau_{u:=e}$, and τ_t^E taken consecutively in σ' in this

order. We can guarantee that this joint transition occurs in σ since H^l and H^r have no communication statement over α; hence these communications are at the front of the cooperation statement.

2. Let σ be a computation of $P[S^r]$. A computation σ' of $P[S^l]$ is constructed as follows: replace every occurrence of transitions τ_h^E and τ_t^X in σ by transitions τ_l^E and τ_l^X in σ', respectively. Transitions τ_h^X, $\tau_{u:=e}$, and τ_t^E will appear in this order in any computation σ of $P[S^r]$, possibly having between them transitions of parallel statements. We delete transitions τ_h^X and τ_t^E from each of these subsequences in σ and replace $\tau_{u:=e}$ by joint transition $\tau_{<\alpha>}$. This is done in a single operation. This operation is consistent since when transition $\tau_{u:=e}$ is taken in σ the locations $post(H^l)$ and $post(H^r)$ are activated, and after taking transition $\tau_{u:=e}$ and deleting transition τ_t^E control locations $pre(T^l)$ and $pre(T^r)$ are activated. A reasoning similar to the one made for the skip-deletion lemma above would show that, in the restricted notation, standard fairness requirements are satisfied by σ'.

The following theorem is related to lemma 4 of section 3.2. The general line of the proofs of both are the same.

Theorem 8 (Incompleteness of any Finite Set of Laws): No finite set of laws, congruences or refinement relations, suffices to syntactically eliminate a pair of synchronous communication statements from restricted SPL statements.

Justification It suffices to prove the theorem in the subset of statements constructed with basic statements and concatenation and cooperation operators only. The congruence in lemma 13 does not eliminate matching synchronous communication operations from general statements. Consider, for instance, the following

$$[H^l; \alpha \Leftarrow e; T^l]||[H^r; [\alpha \Rightarrow u||P^r]; T^r]$$

which is a simple extension of the left hand side of the congruence above, and where P^r is an arbitrary statement. The communication through synchronous channel α can not be eliminated with the communications elimination lemma above, structure matching with its left-hand side is not possible due to P^r. Therefore, in order to eliminate the synchronous communication we need to introduce a new law. Theorem 4 of section 3.2 gives a possible form for this law. Assume that such a law is

$$[H^l; \alpha \Leftarrow e; T^l||H^r; [\alpha \Rightarrow u||P^r]; T^r] \;=\; [H^l||H^r]; [u := e||P^r]; [T^l||T^r]$$

which eliminates the communication pair. Actually, the exact form of the expression is not essential for the reasoning. Restricting the statements to those constructed with concatenation and cooperation operators only, assume that we have found a relation

$$[\ L\ ||\ R\ ]\ =\ G$$

such that G has not the synchronous communication pair that communicates L and R. Statements G, L, and R match the obvious statements in the previous elimination situation. Nevertheless, this second elimination law is not sufficient either. In order to show this, let us complicate slightly anyone of the two parallel statements. Let us select R. The situation is symmetric.

Prefixing or postfixing statements in concatenation with R does not change the structure matching scenario, due to the associativity of concatenation. Composing a new statement in cooperation with R has the same effect, since the associativity and commutativity laws for cooperation allow its removal, leaving the same binary cooperation schema as before. The only remaining way to obtain an essentially different statement, using the two operators only, is to introduce a statement in cooperation with R and to prefix and postfix statements in concatenation with this new binary cooperation, obtaining

$$L\ ||\ [\ H\ ;\ [R||P]\ ;\ T\ ]$$

where H does not contain synchronous communication statements through α, and the asynchronous communication to be eliminated is within R. With the stated restrictions, there is no other way to obtain another expression which is not reducible to the form $L||R$ by structure matching. The assumed new law which eliminated the communication from $L||R$ obtaining statement G does not match with the new statement, for the same reasons as before. Therefore, a new communication elimination law is needed. This reasoning can be iterated indefinitely, introducing new statements H, P, and T at each iteration step, thus creating the need of a new law at each step, reaching the desired conclusion.

□

A.2 Proof of Theorem 6

The theorem is repeated here for easy reference

Concatenated Skip Deletion Theorem *Let S be a statement. Let S_{ncs} be a statement which is neither a communication selection whose communication guards are not all asynchronous sends, nor synchronous communications. Then*

$$S; S_{ncs}\ \approx\ S; \mathbf{skip}; S_{ncs}$$

This congruence holds without the above restrictions upon S_{ncs} when no transition associated with communication statements is in the compassion set $\mathcal{C}$.

We begin with some auxiliary definitions and lemmas. Let us forget, for a moment, about the fairness requirements of computations and work with runs instead, as defined in section 2.5. A *reduced run* of a FTS is obtained from a run in the same way that a reduced behavior was obtained there from a computation.

Definition 14 (Reduced Run): Let M be a FTS, and $\mathcal{O}$ a set of observed variables, where π is not in $\mathcal{O}$. Then a reduced run r^r is obtained from a run r of M by retaining the observable part of all the states appearing in r and deleting any state which is equal to its predecessor but not equal to its successors.

Therefore, stuttering steps are removed from the observable part of a run provided that they do not correspond to a terminal state. Notice also that a reduced run does not need to satisfy any fairness requirements. Then, reduced behaviors would be the subset of reduced runs which satisfy the fairness requirements. The concept of reduced run can be extended to programs, as the reduced run of the FTS associated with the program.

Lemma 14 (Skip Congruence in a Wide Sense): Let $P[.]$ be an arbitrary program context. Then, the sets of reduced runs of

$$P[\ S_1;\ S_2\ ] \quad \text{and} \quad P[\ S_1;\ \mathbf{skip};\ S_2\ ]$$

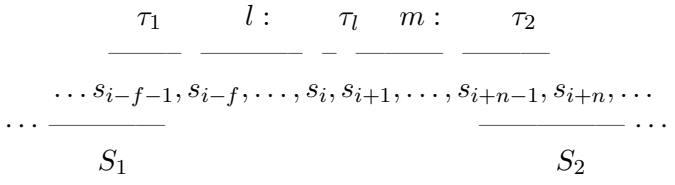
are identical. We express this fact by saying that the two concatenation statements are *congruent in the wide sense*.

Justification Introduce m as the post-label of S_1 in $P[S_1; m : S_2]$ and of **skip** in $P[S_1; \ell : \mathbf{skip}; m : S_2]$. Also, ℓ will be the post-label of S_1 in $P[S_1; \ell : \mathbf{skip}; m : S_2]$

1. Consider a run r of $P[S_1; m : S_2]$. We obtain a run r' of $P[S_1; \ell : \mathbf{skip}; m : S_2]$ by requiring that whenever control reaches m, which is now relabeled as ℓ, a skip transition τ_ℓ is taken immediately. Clearly, r' is a run of $P[S_1; \ell : \mathbf{skip}; m : S_2]$ and the reduced runs of both r and r' are identical, since their only differences are at the transitions τ_ℓ which have been introduced in r', corresponding to the skip statement. These transitions have identical initial and final states. For each such pair of states in r', the run r has only one state. But the first of the two equal states of such pairs will be deleted when the corresponding reduced run is constructed, as it has been defined above. Then, the reduced runs of both r and r' will be equal.

2. Consider now a run r of $P[S_1; \ell : \mathbf{skip}; m : S_2]$. We obtain a run r' of $P[S_1; m : S_2]$ by deleting the initial state of every transition corresponding to the skip statement, when control reaches l. The same reasoning of the previous case shows that the reduced runs obtained from r and r' are identical. □

Let us describe the deletion of skip transitions τ_ℓ with some detail. The rest of this appendix will need and refer to it. A computation σ of $P[S_1; \ell : \mathbf{skip}; m : S_2]$ will have state subsequences (there may be more than one instance, and perhaps infinitely many) of the following form



where $s_{i-f}, \ldots, s_i$ are ℓ-states, in the sense that the control location corresponding to ℓ belongs to π in these states, s_{i-f-1} is not an ℓ-state. Also, states $s_{i+1}, \ldots, s_{i+n-1}$ are m-states, and the transition taken at s_i is the skip transition τ_ℓ, s_{i+n} is not an m-state. Hence, the last transition corresponding to S_1 is taken at state s_{i-f-1}, and a front transition of S_2 is taken at state s_{i+n-1}. They will be referred to as transitions τ_1 and τ_2. The l-states correspond to transitions of statements parallel to S_1 and S_2. The same is true for m-states. Let us construct now the state sequence σ' by deleting all τ_ℓ transitions from σ. This entails replacing ℓ by m in all the l-states of σ. This sequence of states will have subsequences of the form

$$\ldots s_{i-f-1}, s_{i-f}, \ldots, (s_i \equiv s_{i+1}), \ldots, s_{i+n}, \ldots$$

which will correspond to the above subsequences of σ. The states $s_{i-f}, \ldots, (s_i \equiv s_{i+1}), \ldots, s_{i+n-1}$ become now m-states by construction. States s_i and s_{i+1} collapse into the same state. The rest of the sequence remains the same.

For realistic schedulers we would like that the congruence of lemma 14 was true in a strict sense. In other words, that for an arbitrary program context $P[.]$, the set of *reduced behaviors* of $P[S_1; m : S_2]$ and of $P[S_1; \ell : \mathbf{skip}; m : S_2]$ were identical. This is not true due to remark 1 of appendix A. However, the following lemma expresses the fact that it is true in one direction.

Lemma 15 (Sequential **Skip** Insertion): Let $P[.]$ be an arbitrary program context. Then, any reduced behavior of $P[\ S_1;\ S_2\ ]$ is also a reduced behavior of $P[\ S_1;\ \mathbf{skip};\ S_2\ ]$.

Justification Consider a computation σ of $P[S_1; m : S_2]$. We obtain a computation σ' of $P[S_1; \ell : \mathbf{skip}; m : S_2]$ by requiring that whenever control reaches m, a skip transition τ_ℓ is taken immediately. The first m-state becomes now an l-state by relabeling. The remaining m-states remain as such. Clearly, σ', in addition to being a run, is also a computation of $P[S_1; \ell : \mathbf{skip}; m : S_2]$ since the construction does not change the satisfaction of the fairness requirements for any transition. This

is due to the fact that the positions in which a transition τ is enabled or taken in both σ and σ' are the same when only skip transitions are inserted; since no variable value changes by the skip insertion. The reduced behaviors corresponding to σ and to σ' are identical, $\sigma^r = \sigma'^r$. Since any reduced behavior is obtained from a specific computation, and from any computation of $P[S_1; m : S_2]$ a computation of $P[S_1; \ell : \textbf{skip}; m : S_2]$ can be constructed in such a way that the two corresponding reduced behaviors are identical, as it has just been shown, the lemma is proved. □

The reverse of lemma 15 is not true since, when deleting skip transitions from a computation, there is the possibility that some transition which was not enabled in the final state of the skip transition becomes enabled in the same state obtained after the deletion. Hence, the satisfaction of fairness requirements may change for such transition. This is due to the fact that the l-states change now to m-states; this may enable some transition of S_2. Therefore, in order to prove Theorem 6 we have to identify the cases in which the fairness requirements are satisfied by a computation with the skip statement but are not satisfied when the transition corresponding to the skip statement is deleted from such computation. More specifically, the types of S_2 statement giving raise to the transitions violating the fairness requirements have to be identified. The following lemma characterizes the cases where the above may occur.

Lemma 16 (Unfairness Scenario): The only way in which σ, a computation of $P[S_1; \ell : \textbf{skip}; m : S_2]$, can be fair to a transition τ but σ', the state sequence of $P[S_1; m : S_2]$ constructed by deleting τ_ℓ transitions as detailed above, be unfair to τ is if τ is enabled infinitely often in σ' but only finitely often in σ, and τ is taken only finitely many times in both.

Justification By construction of the sequence σ', a transition τ is taken the same number of times in σ and in σ'. This is true, in particular, for a transition τ of S_2. Consequently, the case of τ being taken infinitely many times in both σ and σ' is excluded, since then there would be no possible way to violate any fairness requirement by enabling the transition. The number of times that a transition τ is enabled is not necessarily the same in both σ and σ', since, as commented before, the deletion of the transition corresponding to the skip statement modifies states $s_{i-f}, ..., s_i$, and this may enable transitions in these states which were not enabled in the corresponding states of σ. Therefore, fairness is violated only when τ is enabled in a finite number of positions of σ and in an infinite number of positions of σ'. Then, since the transition is taken in a finite number of positions of both state sequences, fairness with respect to this transition will be violated in σ'. □

The infinite number of enabling positions may be either consecutive or not. In the first case we have continuous enabling and weak fairness would be violated. In the second case strong fairness would be violated.

Lemma 17 (Compassionate Transition): If the state sequence σ' is unfair with respect to transition τ, which is enabled finitely often in $P[S_1; \ell : \mathbf{skip}; m : S_2]$ but infinitely often in $P[S; m : S_2]$, then the control location corresponding to label m is visited an indefinite number of times, in both σ and σ'. Also, τ has to be a front transition of S_2 and compassionate (strongly fair).

Justification The only possible cause of the indefinite number of enablings in σ' is the deletion of the transitions corresponding to the **skip**. As detailed above, the states s_{i-f-1} and those in the sequence $s_{i+1}, ..., s_{i+n}$ cannot be τ-enabling states since they are not affected by the deletion. Therefore, the new enablings take place when replacing the location of l for that of m in at least one of the states of the sequence $s_{i-f}, ..., s_{i-1}$. Notice that this sequence has to have at least one state and it is preceded and followed, in principle, by states where τ is not enabled. Also, τ has to be a front transition of S_2 since it is enabled when control is at the location corresponding to m.

In addition, for an infinite number of new enablings when deleting just one **skip**, either S_2 is within an indefinite loop together with S_1, or it is a loop itself, or it is a terminal statement. In the latter case, the m-state would have to be a terminal state repeating infinitely often. But for that to be possible, location $[m]$ should also be a terminal location. This could only happen if S_2 was the nil statement. But then transition τ would not exist since this statement has no associated transition.

The case of S_2 being an indefinite loop, and hence generating an indefinite sequence, is excluded since then the first transition of S_2 would be a transition from a while statement which is of the skip-type with a boolean condition; and, as such, a while statement can never have two enabled front transitions. Therefore τ and τ_2, as defined above, could not exist.

The above facts show that both S_1 and S_2 have to be within the same indefinite loop, and exclude the possibility of an infinite sequence of continuous new enablings of τ, leaving only the possibility of a sequence of intermittent new enablings of it. Hence, τ can be a strongly fair transition only.

□

Lemma 18 (Disabling Transition): A τ-disabling transition τ'' occurs in both σ and σ' in one of the states of their state subsequence $s_{i-f}, ..., s_{i-1}$. This transition should be competing with transition τ but should be parallel to the transitions associated with statement S_2.

Justification This has to be so since transition τ is enabled but not taken only in a subsequence of the state sequence $s_{i-f}, ..., s_{i-1}$. Since it was caused by replacing an ℓ by an m location in the states of this sequence. Furthermore, τ is disabled in the sequence of states $s_{i+1}, ..., s_{i+n-1}$ which does not change in going

from σ to σ'. Therefore some transition τ'' is taken in one of the states of the first sequence, disabling transition τ. Transition τ'' can not belong to S_2, but it has to be parallel to the transitions associated with S_2, since the first transition of S_2 which is taken is a front transition of this statement. τ_2, is taken in state s_{i+n-1} as it was pointed out before, and τ'' has to be taken in a prior state without moving control from the location corresponding to m. Transition τ'' disables transition τ, hence it should be competing with it.

□

Lemma 19 (Prohibited Statements): Within the reduced notation, The only possibilities for statement S_2 to give rise to the three transitions τ, τ_2 and τ'' identified above are to be a synchronous communication statement and a communication selection statement.

Notice that if this lemma is true then the concatenated skip deletion theorem is also true.

Justification After a review of the semantics of the notation as defined in appendix 2.6, the only statements with transitions in the compassion set are the synchronous communication, the asynchronous receive, and the request statement. The communication selection statement gives rise to compassionate transitions when either synchronous communication or asynchronous receive statements stand at its front.

Statement S_2 of lemma 17, or equivalently statement S_{ncs} of Theorem 6, can be neither an asynchronous receive nor a request statement, since then it could not contribute to the two front transitions τ and τ_2, as it should, according to lemma 17 and the justification of of lemma 18. Therefore, since τ has to be in the compassion set, the only possibilities left for statement S_2 are the ones stated in the lemma. In the following we give details on the only possibilities:

1. Two possibilities when S_2 is a synchronous communication statement:

   (a) The synchronous communication statement, abbreviated as α in the program schema

$$\left[\begin{array}{l} \left[\,\ldots\,;\quad S_2 :: \left[\alpha(\textit{may contribute to } \tau \textit{ or to } \tau_2)\right];\ \ldots\,\right] \\ \| \\ \left[\begin{array}{ll} & \left[\begin{array}{l} \bar{\alpha}(\textit{contributes to } \tau)\ ;\ \ldots \\ \textbf{or} \\ \tau''\ ;\ \ldots \end{array}\right] \\ \ldots\ ; & \end{array}\right]; \ldots \\ \| \\ \left[\,\ldots\ ;\ \bar{\alpha}(\textit{contributes to } \tau_2)\ ;\ \ldots\,\right] \end{array}\right]$$

forms two joint transitions: transition τ with a matching communications statement ($\bar{\alpha}$) of a parallel selection statement, and transition τ_2 with another matching parallel communication statement, $\bar{\alpha}$ in the bottom process. Then transition τ'' corresponds to another front transition of the parallel selection statement.

(b) The synchronous communication statement α, corresponding to S_2, forms two joint transitions, τ and τ_2, with two parallel matching communication statements $l : \bar{\alpha}$, and $m : \bar{\alpha}$ parallel between themselves.

$$\left[\begin{array}{l} \left[\,\ldots\ ;\quad S_2 :: \left[\alpha(\textit{may contribute to } \tau \textit{ or to } \tau_2)\right]\ ;\ \ldots\,\right] \\ \quad || \\ \left[\,\ldots\ ;\ l : \bar{\alpha}(\textit{contributes to } \tau_2)\ ;\ \ldots\,\right] \\ \quad || \\ \left[\,\ldots\ ;\ m : \bar{\alpha}(\textit{may contribute to } \tau \textit{ or to } \tau'');\ \ldots\,\right] \\ \quad || \\ \left[\,\ldots\ ;\ \alpha(\textit{contributes to } \tau'')\ ;\ \ldots\,\right] \end{array}\right]$$

Then the disabling transition τ'' corresponds to a joint transition between the communication statement $m : \bar{\alpha}$ contributing to transition τ, and a fourth synchronous communication statement which is parallel to the three, at the lower process.

2. Two cases for S_2 being a communications selection statement.

(a) All communication statements are synchronous. A second selection statement is parallel to S_2.

$$\left[\begin{array}{l} \left[\begin{array}{l} \left[\,\ldots\ ;\quad S_2 :: \left[\begin{array}{l} \alpha(\textit{contributes to } \tau)\ ;\ \ldots \\ \textbf{or} \\ \tau_2\ ;\ \ldots \end{array}\right]\ ;\ \ldots\,\right] \\ \quad || \\ \left[\,\ldots\ ;\quad \left[\begin{array}{l} \bar{\alpha}(\textit{contributes to } \tau);\ \ldots \\ \textbf{or} \\ \tau''\ ;\ \ldots \end{array}\right]\ ;\ \ldots\,\right] \end{array}\right] \end{array}\right]$$

Joint transition τ is contributed by a communication statement of each selection statement. Transition τ_2 is contributed by S_2 also. Transition τ'' is contributed by the selection statement which is parallel to S_2.

(b) All communication statements are asynchronous. Two of them are parallel to S_2 and between themselves.

$$\left[\begin{array}{l} \left[\begin{array}{l} \ldots\ ; \quad S_2 :: \left[\begin{array}{l} receive(\alpha_{as})(gives\ transition\ \tau)\ ;\ \ldots \\ \textbf{or} \\ \tau_2\ ;\ \ldots \end{array}\right] \end{array}\right] \ ;\ \ldots \\ \quad || \\ \quad [\ \ldots\ ; receive(\alpha_{as})(gives\ transition\ \tau'');\ \ldots\ ] \\ \quad || \\ \quad [\ \ldots\ ; send(\alpha_{as})\ ;\ \ldots\ ] \end{array}\right]$$

A receive statement belonging to S_2 gives rise to transition τ. Transition τ_2 corresponds to a competing transition contributed by S_2. A second receive statement parallel to S_2 gives rise to transition τ'' which, when taken, disables τ when there is one data item in asynchronous channel α_{as}. A send statement, parallel to both, puts a data item on the empty asynchronous channel α_{as}, which is always received when transition τ'' is taken, thus disabling τ. □

As a final remark, in the case where S_2 is a communication selection statement, all its communication guards can not be asynchronous sends, since these statements contribute with weakly fair transitions.

Appendix B

PROOF OF $PIPELINE_1$

B.1 Parallelism to Concatenation Transformation

This appendix details the steps after applying the *iterative communication elimination* reduction, GEN-COMELI of page 104, and starts from the following equivalence:

$$Pipeline_1 \;=_{\mathcal{O}}\; I_0; P_0; P_0; P_0; P_0; E_0$$

$$I_0 :: \left[\begin{array}{l} pc := 1; \\ w := true; \\ (ir.rs1, ir.rs2, ir.rd, ir.func) := (0,0,0,0); \\ (dx.w, dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) := (false,0,0,0,0,0,0); \\ (xw.w, xw.res, xw.rd) := (false,0,0); \\ (wx.w, wx.res, wx.rd) := (false,0,0) \end{array}\right]$$

$$P_0 ::$$

$$\left[\begin{array}{l} wd := xw; \\ \left[\begin{array}{l} \left[\begin{array}{l} \left[\begin{array}{l} \left[\begin{array}{l} instr := mem(pc); \\ pc := pc + 1 \end{array}\right] \\ ir := instr \end{array}\right] \;||\; \left[\begin{array}{l} \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right] \end{array}\right]; \\ [wx := xw] \end{array}\right] \;||\; ; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ dx := xdx; \\ xw := xxw \end{array}\right]$$

$$
\begin{array}{l}
E_0 :: \\
\left[\begin{array}{l}
wd := xw; \\
\left[\begin{array}{l}
\left[\begin{array}{l}
\left[\begin{array}{l}
\left[\begin{array}{l} instr := mem(pc); \\ pc := pc + 1 \end{array}\right] \\
ir := instr
\end{array}\right] \\
[wx := xw]
\end{array}\right] \\
\left[\begin{array}{l}
\textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\
(xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\
\quad := \\
(true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd)
\end{array}\right]
\end{array}\right] \\
\left[\begin{array}{l}
\textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\
\textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\
(xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\
dx := xdx; \\
\left[\begin{array}{l}
\left[\begin{array}{l}
xw := xxw; \\
[\, EX \;\; || \;\; WB \,]
\end{array}\right] \\
\quad || \quad ID
\end{array}\right]
\end{array}\right] \\
\quad || \quad IF
\end{array}\right]
\end{array}\right]
\end{array}
$$

From now on in the text of this proof, the four parallel processes of $Pipeline_1$ are replaced by their identifiers. These are the following:

$$
IF :: \left[\begin{array}{l}
\textbf{loop forever do} \\
\left[\begin{array}{l}
instr := mem(pc); \\
pc := pc + 1; \\
cfd \Leftarrow instr
\end{array}\right]
\end{array}\right]
$$

$$
ID :: \left[\begin{array}{l}
\textbf{loop forever do} \\
\left[\begin{array}{l}
cwd \Rightarrow wd; \\
\textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\
(xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\
\quad := \\
(true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\
cfd \Rightarrow ir; \\
cdx \Leftarrow xdx
\end{array}\right]
\end{array}\right]
$$

$$
EX :: \left[\begin{array}{l}
\textbf{loop forever do} \\
\left[\begin{array}{l}
cwx \Rightarrow wx; \\
\textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\
\textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\
(xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\
cdx \Rightarrow dx; \\
cxw \Leftarrow xxw
\end{array}\right]
\end{array}\right]
$$

$$
WB :: \left[\begin{array}{l}
\textbf{loop forever do} \\
\left[\begin{array}{l}
cwd \Leftarrow xw; \\
cwx \Leftarrow xw; \\
cxw \Rightarrow xw
\end{array}\right]
\end{array}\right]
$$

Some inner cooperation statements have appeared as a result of the communication elimination. These can be transformed to sequential statements with various applications of the *Cooperation and Concatenation* transformation procedure of law 11 of page 39, thus obtaining a truly sequential form, with neither internal communication nor parallelism.

Reduction of E_0:

Statement E_0, the form obtained in the previous proof step, is reduced first. For clarity E_0 is rewritten as:

$$E_0 :: \tag{B.1}$$

$$\left[\begin{array}{l}
wd := xw; \\
\left[\begin{array}{l}
\left[\begin{array}{l}
\left[\begin{array}{l}
\left[\begin{array}{l} instr := mem(pc); \\ pc := pc + 1 \end{array}\right] \; || \; \left[\begin{array}{l} \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right]
\end{array}\right]; \\
ir := instr
\end{array}\right] \; || \\
\left[wx := xw\right]
\end{array}\right]; \\
\left[\begin{array}{l}
\left[\begin{array}{l}
\textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\
\textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\
(xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\
dx := xdx; \\
\left[\left[\begin{array}{l} xw := xxw; \\ [\, EX \;\; || \;\; WB \,] \end{array}\right] \;\; || \;\; ID \right]
\end{array}\right] \;\; || \;\; IF
\end{array}\right]
\end{array}\right]$$

Next step applies the special case of law 10 (*binary cooperation and concatenation*) of page 39:

$$[\, [\, A;\; B\,] \,||\, C\,] \;=_{\mathcal{O}}\; A;\; [\, B \,||\, C\,]$$

where A, B, C are disjoint processes, and $(S_1 = A), (S_3 = B), (S_4 = C)$. This lemma is applied, from left to right, to the dot line framed substatement of E_0. The matchings are,

$$A :: \quad \left[\begin{array}{l}
\textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\
\textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\
(xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\
dx := xdx
\end{array}\right]$$

$$B :: \quad \left[\left[\begin{array}{l} xw := xxw; \\ [\, EX \;\; || \;\; WB \,] \end{array}\right] \;\; || \;\; ID \right]$$

$$C :: \quad [\, IF\,]$$

The resulting form E_0', which is interface equivalent to E_0, is:

$$E_0' ::$$

$$\left[\begin{array}{l}
wd := xw; \\
\left[\begin{array}{l}
\left[\begin{array}{l}
\left[\begin{array}{l} instr := mem(pc); \\ pc := pc + 1 \end{array}\right] \;||\; \left[\begin{array}{l} \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right]; \\
ir := instr
\end{array}\right] \\
[wx := xw]
\end{array}\right] \;||\; \Bigg]; \\
\textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\
\textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\
(xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\
dx := xdx; \\
\left[\begin{array}{l}
\left[\begin{array}{l}
\left[\begin{array}{l} xw := xxw; \\ [\,EX \;||\; WB\,] \end{array}\right] \;||\; ID
\end{array}\right] \;||\; IF
\end{array}\right]
\end{array}\right]$$

Applying again the above *binary cooperation and concatenation* lemma from left to right and within the dotted frame of E_0'. The matchings are: $A :: [\, xw := xxw \,]$, $B :: [\, EX \;||\; WB \,]$, and $C :: [\, ID \,]$. The resulting form E_0'' is:

$$E_0'' ::$$

$$\left[\begin{array}{l}
wd := xw; \\
\left[\begin{array}{l}
\left[\begin{array}{l}
\left[\begin{array}{l} instr := mem(pc); \\ pc := pc + 1 \end{array}\right] \;||\; \left[\begin{array}{l} \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right]; \\
ir := instr
\end{array}\right] \\
[wx := xw]
\end{array}\right] \;||\; \Bigg]; \\
\textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\
\textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\
(xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\
dx := xdx; \\
\left[\begin{array}{l}
\left[\begin{array}{l}
\left[\begin{array}{l} xw := xxw; \\ [\,[\,EX \;||\; WB\,]\,] \end{array}\right] \;||\; ID
\end{array}\right] \;||\; IF
\end{array}\right]
\end{array}\right]$$

Applying again the lemma to E_0'', with the matchings:

$$A :: [\, xw := xxw \,]$$

$$B :: [\, [\, EX \;||\; WB \,] \;||\; ID \,]$$

$$C :: [\, IF \,]$$

obtaining E_0''':

$$E_0''' ::$$

$$\left[\begin{array}{l} wd := xw; \\ \left[\begin{array}{l} \left[\begin{array}{l} \left[\begin{array}{l} \left[\begin{array}{l} instr := mem(pc); \\ pc := pc + 1 \end{array}\right] \\ ir := instr \end{array}\right] \\ [wx := xw] \end{array}\right] \\ \left[\begin{array}{l} \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right] \end{array}\right] \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ dx := xdx; \\ xw := xxw; \\ \left[\begin{array}{l} [\ [\ [\ EX \ ||\ WB\ ]\ ||\ ID\ ]\ ||\ IF\ ] \end{array}\right] \end{array}\right]$$

Finally the tail statements (EX, WB, ID, and IF) are arranged applying the *parallelism flattening* and *parallelism permutation* transformations of section 5.4, obtaining the following equivalence:

$$[\ [\ [\ EX \ ||\ WB\ ]\ ||\ ID\ ]\ ||\ IF\ ] \ =_{\mathcal{O}}\ [\ IF \ ||\ ID \ ||\ EX \ ||\ WB\ ]$$

The following interface equivalence for E_0 is obtained after these applications:

$$E_0 \ =_{\mathcal{O}}\ P_0; E$$

where $E :: [\ IF \ ||\ ID \ ||\ EX \ ||\ WB\ ]$.

Therefore,

$$Pipeline_1 \ =_{\mathcal{O}}\ I_0; P_0; P_0; P_0; P_0; P_0; E$$

It is important to remark that the numbers of repetitions of P_0 in the equivalence is the same as the number of unfolding rules applied in the first step of the proof.

Reduction of P_0:

The proof continues with the *parallelism to concatenation* transformation of each P_0. Applying, within the interactive prover, the *cooperation and concatenation* transformation procedure of section 5.4 of page 122, that basically applies the following lemma from left to right:

$$[\ [\ A \ ||\ B\ ];\ C\ ] \ =_{\mathcal{O}}\ [\ B;\ A;\ C\ ]$$

For the given P_0, the matchings are:

$$A :: \quad \left[\begin{array}{l} instr := mem(pc); \\ pc := pc + 1 \end{array}\right]$$

$$B :: \quad \left[\begin{array}{l} \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right]$$

$$C :: \quad [\ ir := instr\ ]$$

Obtaining P_0':

$$P_0' :: \left[\begin{array}{l} wd := xw; \\ \left[\begin{array}{l} \left[\begin{array}{l} \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ instr := mem(pc); \\ pc := pc + 1; \\ ir := instr \end{array}\right] \quad || \quad [wx := xw] \end{array}\right]; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ dx := xdx; \\ xw := xxw \end{array}\right]$$

A purely sequential form of P_0' is obtained after applying the law 11 for $m = 2$ and the permutation $\langle 2, 1 \rangle$:

$$[\ A \ || \ B\ ] \ =_{\mathcal{O}} \ [\ B;\ A\ ]$$

from left to right. The matchings are:

$$A :: \quad \left[\begin{array}{l} \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ instr := mem(pc); \\ pc := pc + 1; \\ ir := instr \end{array}\right]$$

$$B :: \quad [wx := xw]$$

The sequential form which is obtained is:

$$P_1 :: \left[\begin{array}{l} wd := xw; \\ wx := xw; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ instr := mem(pc); \\ pc := pc + 1; \\ ir := instr; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ dx := xdx; \\ xw := xxw \end{array}\right]$$

Therefore $P_1 =_{\mathcal{O}} P_0$, and the resulting interface equivalence is:

$$Pipeline_1 =_{\mathcal{O}} I_0; P_1; P_1; P_1; P_1; P_1; E$$

B.2 Concatenation Commutativity

The goal of the proof is to reach the *Von Neumann* iteration. In order to reach this goal, the body of P_1, shown above, must be rearranged. This is possible since some of the statements of P_1 are disjoint, and simple concatenation permutation rules can be applied. Thus, the body of P_1 can be transformed into an equivalent sequential form. The *concatenation commutativity* law (see law 4 of page 38) permutes disjoint and non-communicating statements as follows:

$$[\ A_x;\ B_x\ ] =_{\mathcal{O}} [\ B_x; A_x\ ]$$

Applying the lemma from left to right to the substatements $[\ pc := pc + 1;\ ir := instr\ ]$ of P_1, where

$$A_1 :: \quad [\ pc := pc + 1\ ]$$
$$B_1 :: \quad [\ ir := instr\ ]$$

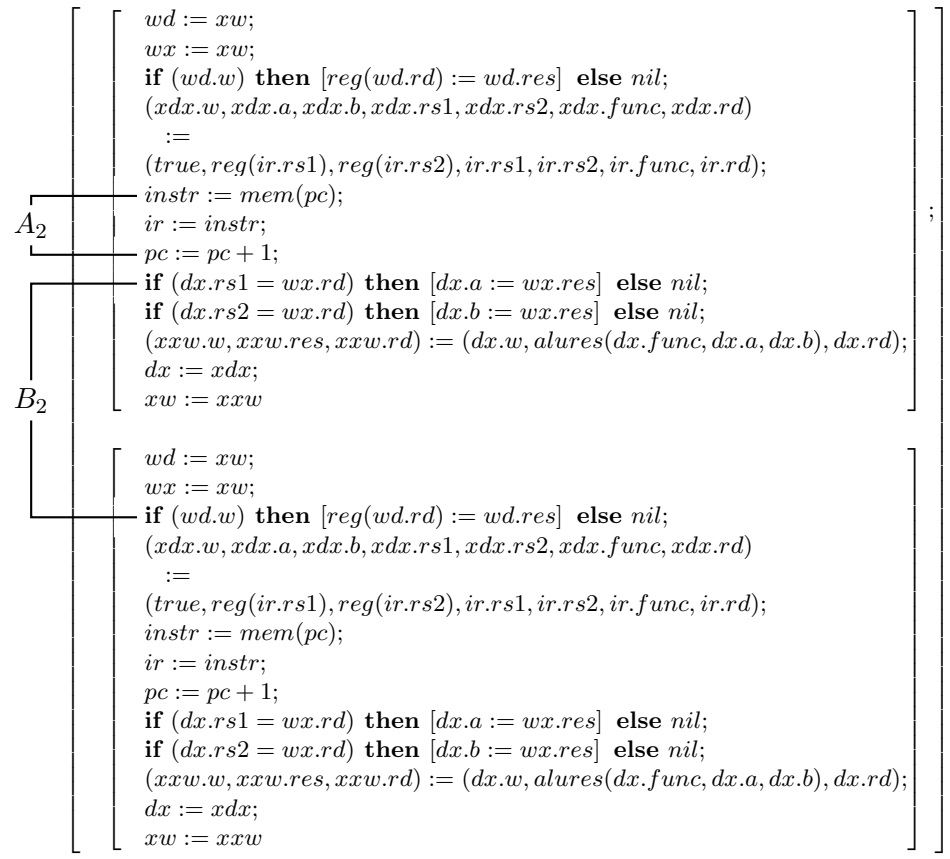
P_1' is obtained:

$$P_1' :: \left[\begin{array}{l} wd := xw; \\ wx := xw; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ instr := mem(pc); \\ ir := instr; \\ pc := pc + 1; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ dx := xdx; \\ xw := xxw \end{array}\right] \tag{B.2}$$

The resulting interface equivalence is:

$$Pipeline_1 \ =_{\mathcal{O}} \ [\ I_0; P_1; P_1; P_1; P_1; P_1; E\ ] \ =_{\mathcal{O}} [\ I_0; P_1'; P_1'; P_1'; P_1'; P_1'; E\ ]$$

Next step applies the above commutative transformation to the concatenation pair $[P_1'; P_1']$ expanded below:



The matchings are:

$$A_2 :: \left[\begin{array}{l} instr := mem(pc); \\ ir := instr; \\ pc := pc + 1 \end{array}\right]$$

$$B_2 :: \left[\begin{array}{l} \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ dx := xdx; \\ xw := xxw; \\ wd := xw; \\ wx := xw; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil \end{array}\right]$$

Observe that B_2 contains statements from two consecutive P_1' statements. The resulting interface equivalence is:

$$Pipeline_1 =_{\mathcal{O}} [\ I_0; P_1'; P_1'; P_1'; P_1'; P_1'; E\ ] =_{\mathcal{O}} [\ I_0; R_0; P_2; P_2; P_2; P_2; E_2; E\ ]$$

where:

$$R_0 :: \left[\begin{array}{l} wd := xw; \\ wx := xw; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ dx := xdx; \\ xw := xxw \end{array}\right]$$

$$P_2 :: \left[\begin{array}{l} wd := xw; \\ wx := xw; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ instr := mem(pc); \\ ir := instr; \\ pc := pc + 1; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ dx := xdx; \\ xw := xxw \end{array}\right]$$

$$E_2 :: \left[\begin{array}{l} instr := mem(pc); \\ ir := instr; \\ pc := pc + 1 \end{array}\right]$$

The last concatenation commutativity reduction is applied to P_2 with the following matchings:

$$A :: \left[\begin{array}{l} instr := mem(pc); \\ ir := instr; \\ pc := pc + 1; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right]$$

$$B :: \left[\begin{array}{l} \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd) \end{array}\right]$$

obtaining P_3:

$$P_3 :: \left[\begin{array}{l} wd := xw; \\ wx := xw; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ instr := mem(pc); \\ ir := instr; \\ pc := pc + 1; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ dx := xdx; \\ xw := xxw \end{array}\right] \tag{B.3}$$

and

$$Pipeline_1 =_{\mathcal{O}} [\ I_0; R_0; P_2; P_2; P_2; P_2; E_2; E\ ] =_{\mathcal{O}} [\ I_0; R_0; P_3; P_3; P_3; P_3; E_2; E\ ]$$

B.3 Redundant Variable Elimination

Some redundant variable assignments have appeared in the above sequential composition, $[\ I_0; R_0; P_3; P_3; P_3; P_3; E_2; E\ ]$, due to the elimination of the synchronous channels done in the communication elimination step. The interactive application of the variable elimination reductions of section 5.4 will remove them and a more simpler sequential form will be reached.

The variable elimination is carried out within the interactive prover by applying the following *variable and assignment elim-intro* law (see law 13 of page 40):

$$[\ v := e; S_1(v); S_2\ ] =_{\mathcal{O}} [\ S_1(e); S_2\ ]$$

and *multiple variable and assignment elim-intro* law (see law 17 of page 41):

$$[\ (v.v_1, ..., v.v_n) := (e_1, ..., e_n);\ S_1(v.v_1, ..., v.v_n);\ S_2\ ] =_{\mathcal{O}} [\ S_1(e_1, ..., e_n);\ S_2\ ]$$

which can be written more succinctly as:

$$[\ (\bar{v}) := (\bar{e}); S_1(\bar{v}); S_2\ ] \quad =_{\mathcal{O}} \quad [\ S_1(\bar{e}); S_2\ ]$$

Both lemmas are applied from left to right.

Elimination of *instr*:

Variable *instr* is removed from each P_3, statement B.3, applying the variable assignment elimination. The matchings for law 13 are:

$$\begin{array}{rcl} v := e & :: & instr := mem(pc) \\ S_1(v) & :: & [\ ir := instr\ ] \\ S_2 & :: & \text{all the statements in sequence after } S_1(v) \\ & & \text{ending in a new assignment to } instr \end{array}$$

Obtaining P_3':

$$P_3' :: \left[\begin{array}{l} wd := xw; \\ wx := xw; \\ \textbf{if}\ (wd.w)\ \textbf{then}\ [reg(wd.rd) := wd.res]\ \textbf{else}\ nil; \\ \textbf{if}\ (dx.rs1 = wx.rd)\ \textbf{then}\ [dx.a := wx.res]\ \textbf{else}\ nil; \\ \textbf{if}\ (dx.rs2 = wx.rd)\ \textbf{then}\ [dx.b := wx.res]\ \textbf{else}\ nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ dx := xdx; \\ xw := xxw \end{array}\right]$$

Elimination of *xw*:

This elimination is done in several steps. First xw is removed from $[\ I_0; R_0\ ]$, applying the multiple assignment elimination. The matchings for law 17, where $n = 3$, are:

$$\begin{array}{rcl} (v.v_1, v.v_2, v.v_3) := (e_1, e_2, e_3) & :: & (xw.w, xw.res, xw.rd) := (false, 0, 0) \\ S_1(\bar{v}) & :: & \left[\begin{array}{l} wd := xw; \\ wx := xw \end{array}\right] \\ S_2 & :: & \text{all the statements in sequence after } S_1(\bar{v}) \\ & & \text{until a new assignment to } xw \end{array}$$

Obtaining $[\ I'_0; R'_0\ ]$,

$$\left[\begin{array}{ll} I'_0 :: & \left[\begin{array}{l} pc := 1; \\ w := true; \\ (ir.rs1, ir.rs2, ir.rd, ir.func) := (0,0,0,0); \\ (dx.w, dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) := (false, 0,0,0,0,0,0); \\ (wx.w, wx.res, wx.rd) := (false, 0, 0) \end{array}\right]; \\ R'_0 :: & \left[\begin{array}{l} (wd.w, wd.res, wd.rd) := (false, 0, 0); \\ (wx.w, wx.res, wx.rd) := (false, 0, 0); \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ dx := xdx; \\ xw := xxw \end{array}\right] \end{array}\right]$$

where variables wd and wx of R'_0 have been expanded in terms of their components.

Next step removes variable xw from $[\ R'_0; P_3\ ]$ applying the variable elimination lemma. The matchings are:

$$\begin{array}{rcl} v := e & :: & xw := xxw \\ S_1(v) & :: & \left[\begin{array}{l} wd := xw; \\ wx := xw \end{array}\right] \\ S_2 & :: & \text{all the statements in sequence after } S_1(v) \\ & & \text{until a new assignment to } xw \end{array}$$

Obtaining $[\ R''_0; P'_3\ ]$,

$$R''_0 :: \left[\begin{array}{l} (wd.w, wd.res, wd.rd) := (false, 0, 0); \\ (wx.w, wx.res, wx.rd) := (false, 0, 0); \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ dx := xdx \end{array}\right]$$

$$P'_3 :: \left[\begin{array}{l} wd := xxw; \\ wx := xxw; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ dx := xdx; \\ xw := xxw \end{array}\right]$$

and

$$Pipeline_1 =_{\mathcal{O}} [\ I'_0; R''_0; P'_3; P_3; P_3; P_3; E_2; E\ ]$$

Remaining assignments to variable xw are removed applying the lemma to all concatenations $[\ P'_3; P_3\ ]$. The matching are the same as above. Obtaining $[\ P''_3; P'_3\ ]$, where P''_3 is:

$$P''_3 :: \left[\begin{array}{l} wd := xxw; \\ wx := xxw; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ dx := xdx \end{array}\right]$$

and

$$Pipeline_1 =_{\mathcal{O}} [\ I_0; R''_0; P''_3; P'_3; P_3; P_3; E_2; E\ ] =_{\mathcal{O}} [\ I_0; R''_0; P''_3; P''_3; P''_3; P'_3; E_2; E\ ]$$

Elimination of xdx:

The multiple variable elimination is applied to remove xdx from R''_0 and P''_3 statements. The matchings are:

$$(v.v_1, v.v_2, ..., v.v_7) := (e_1, e_2, ..., e_7) \;::\; \left[\begin{array}{l}(xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd)\end{array}\right]$$

$$S_1(\bar{v}) \;::\; [\;\; dx := xdx \;\;]$$

$$S_2 \;::\; \text{all the statements in sequence after } S_1(\bar{v}) \text{ until a new assignment to } xdx$$

The obtained interface equivalence is:

$$Pipeline_1 \;=_{\mathcal{O}}\; [\; I_0'; R_0''; P_3'''; P_3'''; P_3'''; P_3''; E_2; E\;] \;=_{\mathcal{O}}\; [\; I_0'; R_1; P_4; P_4; P_4; P_8; E_3; E\;]$$

where:

$$R_1 :: \left[\begin{array}{l}(wd.w, wd.res, wd.rd) := (false, 0, 0); \\ (wx.w, wx.res, wx.rd) := (false, 0, 0); \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ (dx.w, dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd)\end{array}\right]$$

$$P_4 :: \left[\begin{array}{l}wd := xxw; \\ wx := xxw; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (dx.w, dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd)\end{array}\right]$$

$$P_8 :: \left[\begin{array}{l}wd := xxw; \\ wx := xxw; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ dx := xdx\end{array}\right]$$

$$E_3 :: \left[\begin{array}{l}xw := xxw; \\ instr := mem(pc); \\ ir := instr; \\ pc := pc + 1\end{array}\right]$$

E_3 can also be expressed as $[\ [xw := xxw];\ E_2\ ]$. Observe that variable xdx can not be eliminated from P_3''.

Elimination of wd:
Variable wd is removed from R_1 and P_4 applying again variable elimination. The matchings for R_1 are:

$$
\begin{array}{rcl}
(v.v_1, v.v_2, v.v_3) := (e_1, e_2, e_3) & :: & (wd.w, wd.res, wd.rd) := (false, 0, 0) \\
S_1(\bar{v}) & :: & \left[\begin{array}{l} (wx.w, wx.res, wx.rd) := (false, 0, 0); \\ \textbf{if}\ (wd.w)\ \textbf{then}\ [reg(wd.rd) := wd.res]\ \textbf{else}\ nil \end{array}\right] \\
S_2 & :: & \text{all the statements in sequence after } S_1(\bar{v}) \\
 & & \text{until a new assignment to } wd
\end{array}
$$

and for P_4:

$$
\begin{array}{rcl}
v := e & :: & wd := xxw \\
S_1(v) & :: & \left[\begin{array}{l} wx := xxw; \\ \textbf{if}\ (wd.w)\ \textbf{then}\ [reg(wd.rd) := wd.res]\ \textbf{else}\ nil \end{array}\right] \\
S_2 & :: & \text{all the statements in sequence after } S_1(v) \\
 & & \text{until a new assignment to } wd
\end{array}
$$

Obtaining R_1' and P_4':

$$
R_1' :: \left[\begin{array}{l}
(wx.w, wx.res, wx.rd) := (false, 0, 0); \\
\textbf{if}\ (false)\ \textbf{then}\ [reg(0) := 0]\ \textbf{else}\ nil; \\
\textbf{if}\ (dx.rs1 = wx.rd)\ \textbf{then}\ [dx.a := wx.res]\ \textbf{else}\ nil; \\
\textbf{if}\ (dx.rs2 = wx.rd)\ \textbf{then}\ [dx.b := wx.res]\ \textbf{else}\ nil; \\
(xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\
(dx.w, dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\
\quad := \\
(true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd)
\end{array}\right]
$$

$$
P_4' :: \left[\begin{array}{l}
wx := xxw; \\
\textbf{if}\ (xxw.w)\ \textbf{then}\ [reg(xxw.rd) := xxw.res]\ \textbf{else}\ nil; \\
\textbf{if}\ (dx.rs1 = wx.rd)\ \textbf{then}\ [dx.a := wx.res]\ \textbf{else}\ nil; \\
\textbf{if}\ (dx.rs2 = wx.rd)\ \textbf{then}\ [dx.b := wx.res]\ \textbf{else}\ nil; \\
(xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\
ir := mem(pc); \\
pc := pc + 1; \\
(dx.w, dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\
\quad := \\
(true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd)
\end{array}\right]
$$

Observe that the overall interface equivalence is now:

$$Pipeline_1 \ =_{\mathcal{O}} \ [\ I_0''; R_1'; P_4'; P_4'; P_4'; P_8; E_3; E\ ]$$

The assignment $wd := xxw$ of P_8 can not be eliminated because no other unhidden assignment to wd is found in its sequential statements, $[\ E_3; E\ ]$.

Elimination of wx:

As above, variable wx is eliminated from R_1' and P_4'. The matchings for R_1' are:

$$\begin{array}{rcl}
(v.v_1, v.v_2, v.v_3) := (e_1, e_2, e_3) & :: & (wx.w, wx.res, wx.rd) := (false, 0, 0) \\
S_1(\bar{v}) & :: & \left[\begin{array}{l} \textbf{if } (false) \textbf{ then } [reg(0) := 0] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil \end{array}\right] \\
S_2 & :: & \text{all the statements in sequence after } S_1(\bar{v}) \\ & & \text{until a new assignment to } wx
\end{array}$$

and for P_4':

$$\begin{array}{rcl}
v := e & :: & wx := xxw \\
S_1(v) & :: & \left[\begin{array}{l} \textbf{if } (xxw.w) \textbf{ then } [reg(xxw.rd) := xxw.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil \end{array}\right] \\
S_2 & :: & \text{all the statements in sequence after } S_1(v) \\ & & \text{until a new assignment to } wx
\end{array}$$

One obtains R_1'' and P_4'':

$$R_1'' :: \left[\begin{array}{l}
\textbf{if } (false) \textbf{ then } [reg(0) := 0] \textbf{ else } nil; \\
\textbf{if } (dx.rs1 = 0) \textbf{ then } [dx.a := 0] \textbf{ else } nil; \\
\textbf{if } (dx.rs2 = 0) \textbf{ then } [dx.b := 0] \textbf{ else } nil; \\
(xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\
(dx.w, dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\
\quad := \\
(true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd)
\end{array}\right]$$

$$P_4'' :: \left[\begin{array}{l} \textbf{if } (xxw.w) \textbf{ then } [reg(xxw.rd) := xxw.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = xxw.rd) \textbf{ then } [dx.a := xxw.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = xxw.rd) \textbf{ then } [dx.b := xxw.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (dx.w, dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right]$$

The assignment $wx := xxw$ can not be eliminated from P_8 due to no new uncovered assignment to wx is found in $[E_3; E]$.

The interface equivalence is:

$$Pipeline_1 \ =_{\mathcal{O}} \ [\ I_0''; R_1''; P_4''; P_4''; P_4''; P_8; E_3; E\ ]$$

Elimination of $dx.w$:

The elimination of variable $dx.w$ is carried out applying the *multiple assignment partial elimination* law (see law 18 of page 41). This is the following:

$$\begin{array}{c} [\ (v.v_1, ..., v.v_i, ..., v.v_j, ..., v.v_n) := (e_1, ..., e_i, ..., e_j, ..., e_n);\ S_1(v.v_i, ..., v.v_j);\ S_2\ ] \\ =_{\mathcal{O}} \\ [\ (v.v_1, ..., v.v_n) := (e_1, ..., e_n);\ S_1(e_i, ..., e_j);\ S_2\ ] \end{array}$$

The lemma is applied from left to right.

First $dx.w$ is eliminated from $[\ I_0'; R_1''\ ]$. The matchings for $i = j$ are:

$$\begin{array}{rcl} v.v_i & :: & dx.w \\ (v.v_1, ..., v.v_i, ..., v.v_n) := (e_1, ..., e_i, ..., e_n) & :: & \left[\begin{array}{l}(dx.w, dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (false, 0, 0, 0, 0, 0, 0)\end{array}\right] \\ S_1(v.v_i) & :: & \left[\begin{array}{l}\textbf{if } (false) \textbf{ then } [reg(0) := 0] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = 0) \textbf{ then } [dx.a := 0] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = 0) \textbf{ then } [dx.b := 0] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) \\ \quad := \\ (dx.w, alures(dx.func, dx.a, dx.b), dx.rd)\end{array}\right] \\ S_2 & :: & \text{all the statements in sequence after } S_1(v.v_i) \\ & & \text{until a new assignment to } dx.w \end{array}$$

Obtaining $[\ I_1; R_1'''\ ]$, where:

$$I_1 :: \left[\begin{array}{l} pc := 1; \\ (ir.rs1, ir.rs2, ir.rd, ir.func) := (0,0,0,0); \\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) := (0,0,0,0,0,0); \\ (wx.w, wx.res, wx.rd) := (false, 0, 0) \end{array}\right]$$

$$R_1''' :: \left[\begin{array}{l} \textbf{if } (false) \textbf{ then } [reg(0) := 0] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = 0) \textbf{ then } [dx.a := 0] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = 0) \textbf{ then } [dx.b := 0] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (false, alures(dx.func, dx.a, dx.b), dx.rd); \\ (dx.w, dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right]$$

Next step eliminates $dx.w$ from $[\ R_1'''; P_4''\ ]$ with the following matchings:

$$\begin{array}{rcl} v.v_i & :: & dx.w \\ (v.v_1, ..., v.v_i, ..., v.v_n) := (e_1, ..., e_i, ..., e_n) & :: & \left[\begin{array}{l} (dx.w, dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (false, 0, 0, 0, 0, 0, 0) \end{array}\right] \\ S_1(v.v_i) & :: & \left[\begin{array}{l} \textbf{if } (xxw.w) \textbf{ then } [reg(xxw.rd) := xxw.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = xxw.rd) \textbf{ then } [dx.a := xxw.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = xxw.rd) \textbf{ then } [dx.b := xxw.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) \quad := \\ (dx.w, alures(dx.func, dx.a, dx.b), dx.rd) \end{array}\right] \\ S_2 & :: & \text{all the statements in sequence after } S_1(v.v_i) \\ & & \text{until a new assignment to } dx.w \end{array}$$

One obtains $[\ R_2; P_5\ ]$:

$$R_2 :: \left[\begin{array}{l} \textbf{if } (false) \textbf{ then } [reg(0) := 0] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = 0) \textbf{ then } [dx.a := 0] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = 0) \textbf{ then } [dx.b := 0] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (false, alures(dx.func, dx.a, dx.b), dx.rd); \\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right]$$

$$P_5 :: \left[\begin{array}{l}
\textbf{if } (xxw.w) \textbf{ then } [reg(xxw.rd) := xxw.res] \textbf{ else } nil; \\
\textbf{if } (dx.rs1 = xxw.rd) \textbf{ then } [dx.a := xxw.res] \textbf{ else } nil; \\
\textbf{if } (dx.rs2 = xxw.rd) \textbf{ then } [dx.b := xxw.res] \textbf{ else } nil; \\
(xxw.w, xxw.res, xxw.rd) := (true, alures(dx.func, dx.a, dx.b), dx.rd); \\
ir := mem(pc); \\
pc := pc + 1; \\
(dx.w, dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\
\quad := \\
(true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd)
\end{array}\right]$$

where the interface equivalence is:

$$Pipeline_1 \ =_{\mathcal{O}} \ [\ I_1; R_2; P_5; P_4''; P_4''; P_8; E_3; E\ ]$$

The remaining assignments to variable $dx.w$ are removed applying the same lemma to all concatenations $[\ P_5; P_4''\ ]$. The matching are the same as above. Obtaining $[\ P_5'; P_5\ ]$, where P_5' is:

$$P_5' :: \left[\begin{array}{l}
\textbf{if } (xxw.w) \textbf{ then } [reg(xxw.rd) := xxw.res] \textbf{ else } nil; \\
\textbf{if } (dx.rs1 = xxw.rd) \textbf{ then } [dx.a := xxw.res] \textbf{ else } nil; \\
\textbf{if } (dx.rs2 = xxw.rd) \textbf{ then } [dx.b := xxw.res] \textbf{ else } nil; \\
(xxw.w, xxw.res, xxw.rd) := (true, alures(dx.func, dx.a, dx.b), dx.rd); \\
ir := mem(pc); \\
pc := pc + 1; \\
(dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\
\quad := \\
(reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd)
\end{array}\right]$$

where,

$$Pipeline_1 \ =_{\mathcal{O}} \ [\ I_1; R_2; P_5'; P_5'; P_5; P_8; E_3; E\ ]$$

Similarly for the concatenation, $[\ P_5; P_8\ ]$, obtaining $[\ P_5'; P_8'\ ]$.

$$P_8' :: \left[\begin{array}{l}
wd := xxw; \\
wx := xxw; \\
\textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\
\textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\
\textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\
(xxw.w, xxw.res, xxw.rd) := (true, alures(dx.func, dx.a, dx.b), dx.rd); \\
ir := mem(pc); \\
pc := pc + 1; \\
(xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\
\quad := \\
(true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\
dx := xdx
\end{array}\right]$$

The overall interface equivalence is now:

$$Pipeline_1 \ =_{\mathcal{O}} \ [\ I_1; R_2; P_5'; P_5'; P_5'; P_8'; E_3; E\ ]$$

Elimination of $xxw.w$:

Variable $xxw.w$ is eliminated from $[\ R_2; P_5';\ ]$ applying the above lemma with the following matchings:

$$\begin{array}{rcl}
v.v_i & :: & xxw.w \\
(v.v_1, ..., v.v_i, ..., v.v_n) & & \\
:= & :: & (xxw.w, xxw.res, xxw.rd) := (false, alures(dx.func, dx.a, dx.b), dx.rd) \\
(e_1, ..., e_i, ..., e_n) & & \\
S_1(v.v_i) & :: & \left[\begin{array}{l}
(dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\
\quad := \\
(reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\
\textbf{if } (xxw.w) \textbf{ then } [reg(xxw.rd) := xxw.res] \textbf{ else } nil; \\
\textbf{if } (dx.rs1 = xxw.rd) \textbf{ then } [dx.a := xxw.res] \textbf{ else } nil; \\
\textbf{if } (dx.rs2 = xxw.rd) \textbf{ then } [dx.b := xxw.res] \textbf{ else } nil
\end{array}\right] \\
S_2 & :: & \text{all the statements in sequence after } S_1(v.v_i) \\
 & & \text{until a new assignment to } xxw.w
\end{array}$$

One obtains $[\ R_2'; P_5''\ ]$ where:

$$R_2' :: \left[\begin{array}{l}
\textbf{if } (false) \textbf{ then } [reg(0) := 0] \textbf{ else } nil; \\
\textbf{if } (dx.rs1 = 0) \textbf{ then } [dx.a := 0] \textbf{ else } nil; \\
\textbf{if } (dx.rs2 = 0) \textbf{ then } [dx.b := 0] \textbf{ else } nil; \\
(xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd); \\
(dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\
\quad := \\
(reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd)
\end{array}\right]$$

$$P_5'' :: \left[\begin{array}{l}
\textbf{if } (false) \textbf{ then } [reg(xxw.rd) := xxw.res] \textbf{ else } nil; \\
\textbf{if } (dx.rs1 = xxw.rd) \textbf{ then } [dx.a := xxw.res] \textbf{ else } nil; \\
\textbf{if } (dx.rs2 = xxw.rd) \textbf{ then } [dx.b := xxw.res] \textbf{ else } nil; \\
(xxw.w, xxw.res, xxw.rd) := (true, alures(dx.func, dx.a, dx.b), dx.rd); \\
ir := mem(pc); \\
pc := pc + 1; \\
(dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\
\quad := \\
(reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd)
\end{array}\right]$$

where,

$$Pipeline_1 \ =_{\mathcal{O}} \ [\ I_1; R_2'; P_5''; P_5'; P_5'; P_8'; E_3; E\ ]$$

The remaining instances of variable $xxw.w$ are removed applying again the lemma to $[\ P_5''; P_5'\ ]$ with the same matchings as above, obtaining $[\ P_6; P_7\ ]$, where

$$P_6 :: \left[\begin{array}{l} \textbf{if } (false) \textbf{ then } [reg(xxw.rd) := xxw.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = xxw.rd) \textbf{ then } [dx.a := xxw.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = xxw.rd) \textbf{ then } [dx.b := xxw.res] \textbf{ else } nil; \\ (xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right]$$

$$P_7 :: \left[\begin{array}{l} \textbf{if } (true) \textbf{ then } [reg(xxw.rd) := xxw.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = xxw.rd) \textbf{ then } [dx.a := xxw.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = xxw.rd) \textbf{ then } [dx.b := xxw.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (true, alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right]$$

and

$$Pipeline_1 =_{\mathcal{O}} [\ I_1; R_2'; P_6; P_7; P_5'; P_8'; E_3; E\ ]$$

Similarly for $[\ P_7; P_5'\ ]$ obtaining $[\ P_7'; P_7\ ]$, where

$$P_7' :: \left[\begin{array}{l} \textbf{if } (true) \textbf{ then } [reg(xxw.rd) := xxw.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = xxw.rd) \textbf{ then } [dx.a := xxw.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = xxw.rd) \textbf{ then } [dx.b := xxw.res] \textbf{ else } nil; \\ (xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right]$$

and

$$Pipeline_1 =_{\mathcal{O}} [\ I_1; R_2'; P_6; P_7'; P_7; P_8'; E_3; E\ ]$$

Finally, removing assignment to $xxw.w$ from $[\ P_7; P_8'\ ]$, obtaining $[\ P_7'; P_8''\ ]$.

$$P_8'' :: \left[\begin{array}{l} (wd.w, wd.res, wd.rd) := (true, xxw.res, xxw.rd); \\ (wx.w, wx.res, wx.rd) := (true, xxw.res, xxw.rd); \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (true, alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ dx := xdx \end{array}\right]$$

The overall interface equivalence has now been transformed into:

$$Pipeline_1 \ =_{\mathcal{O}} \ [\ I_1; R_2'; P_6; P_7'; P_7'; P_8''; E_3; E\ ]$$

Observe that the values of the variable components $wd.w$ and $wx.w$ of P_8'' have been replaced.

Simple 'if ' statements Elimination:

After the variable elimination reduction step some simple boolean '**if** ' statements can be simplified applying a trivial simplification transformation:

if '$true$' congruence: $\quad$ **if** $true$ **then** S_1 **else** $S_2 \ \approx \ S_1$

if '$false$' congruence: $\quad$ **if** $false$ **then** S_1 **else** $S_2 \ \approx \ S_2$

Applying the second congruence to P_6 with the matchings:

$$\begin{array}{lcl} S_1 & :: & reg(xxw.rd) := xxw.res \\ S_2 & :: & nil \end{array}$$

obtaining P_6'. Next step applies the first **if** congruence to P_7', with the same matchings as above, obtaining P_7'':

$$P'_6 :: \left[\begin{array}{l} reg(xxw.rd) := xxw.res; \\ \textbf{if } (dx.rs1 = xxw.rd) \textbf{ then } [dx.a := xxw.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = xxw.rd) \textbf{ then } [dx.b := xxw.res] \textbf{ else } nil; \\ (xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right]$$

$$P''_7 :: \left[\begin{array}{l} reg(xxw.rd) := xxw.res; \\ \textbf{if } (dx.rs1 = xxw.rd) \textbf{ then } [dx.a := xxw.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = xxw.rd) \textbf{ then } [dx.b := xxw.res] \textbf{ else } nil; \\ (xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right]$$

where the interface equivalence is:

$$Pipeline_1 \ =_{\mathcal{O}} \ [\ I_1; R'_2; P'_6; P''_7; P''_7; P''_8; E_3; E\ ]$$

Propagation of variable $dx.rs1$ ***and*** $dx.rs2$:

The propagation of variable $dx.rs1$ and $dx.rs2$ is carried out applying the *multiple assignment partial elimination* law (see law 18 of page 41). This is the following:

$$\begin{array}{c} [\ (v.v_1, ..., v.v_i, ..., v.v_j, ..., v.v_n) := (e_1, ..., e_i, ..., e_j, ..., e_n);\ S_1(v.v_i, ..., v.v_j);\ S_2\ ] \\ =_{\mathcal{O}} \\ [\ (v.v_1, ..., v.v_i, ..., v.v_j, ..., v.v_n) := (e_1, ..., e_i, ..., e_j, ..., e_n);\ S_1(e_i, ..., e_j);\ S_2\ ] \end{array}$$

Some variables of the multiple assignment are replaced with their value in S_1. The multiple assignment remains unchanged. This lemma is applied from left to right to the concatenation $[\ R'_2; P'_6; P''_7; P''_7\ ]$ with the matchings:

$$
\begin{array}{rcl}
v.v_i & :: & dx.w \\
(v.v_1, ..., v.v_i, ..., v.v_j, ..., v.v_n) & & \left[\begin{array}{l}(dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd)\end{array}\right] \\
:= & :: & \\
(e_1, ..., e_i, ..., e_j, ..., e_n) & & \\
S_1(v.v_i, v.v_j) & :: & \left[\begin{array}{l}reg(xxw.rd) := xxw.res; \\ \textbf{if } (dx.rs1 = xxw.rd) \textbf{ then } [dx.a := xxw.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = xxw.rd) \textbf{ then } [dx.b := xxw.res] \textbf{ else } nil\end{array}\right] \\
S_2 & :: & \text{all the statements in sequence after } S_1(v.v_i, v.v_j) \\
 & & \text{until a new assignment to } dx.rs1 \text{ and } dx.rs2
\end{array}
$$

obtaining $[\ R'_2; P''_6; P'''_7; P'''_7\ ]$, where:

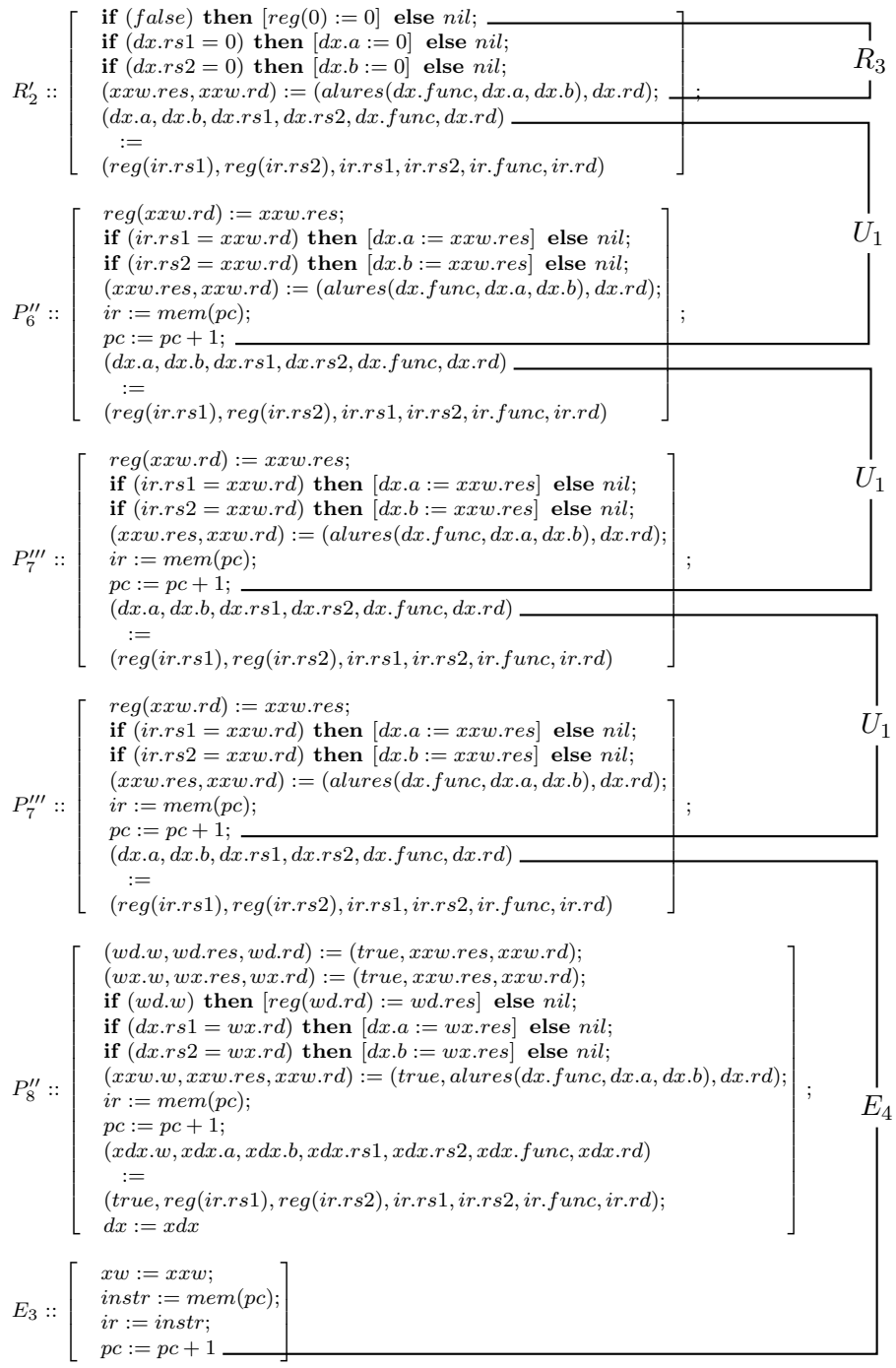
$$
P''_6 :: \left[\begin{array}{l}
reg(xxw.rd) := xxw.res; \\
\textbf{if } (ir.rs1 = xxw.rd) \textbf{ then } [dx.a := xxw.res] \textbf{ else } nil; \\
\textbf{if } (ir.rs2 = xxw.rd) \textbf{ then } [dx.b := xxw.res] \textbf{ else } nil; \\
(xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd); \\
ir := mem(pc); \\
pc := pc + 1; \\
(dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\
\quad := \\
(reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd)
\end{array}\right]
$$

$$
P'''_7 :: \left[\begin{array}{l}
reg(xxw.rd) := xxw.res; \\
\textbf{if } (ir.rs1 = xxw.rd) \textbf{ then } [dx.a := xxw.res] \textbf{ else } nil; \\
\textbf{if } (ir.rs2 = xxw.rd) \textbf{ then } [dx.b := xxw.res] \textbf{ else } nil; \\
(xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd); \\
ir := mem(pc); \\
pc := pc + 1; \\
(dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\
\quad := \\
(reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd)
\end{array}\right]
$$

The overall interface equivalence is:

$$Pipeline_1 =_{\mathcal{O}} [\ I_1; R'_2; P''_6; P'''_7; P'''_7; P''_8; E_3; E\ ]$$

The above interface equivalence can be relabeled as follows:



obtaining the new equivalence:

$$Pipeline_1 \ =_{\mathcal{O}} \ [\ I_1; R_3; U_1; U_1; U_1; E_4; E\ ]$$

where,

$$R_3 :: \left[\begin{array}{l} \textbf{if } (false) \textbf{ then } [reg(0) := 0] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = 0) \textbf{ then } [dx.a := 0] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = 0) \textbf{ then } [dx.b := 0] \textbf{ else } nil; \\ (xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd) \end{array}\right]$$

$$U_1 :: \left[\begin{array}{l} (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ reg(xxw.rd) := xxw.res; \\ \textbf{if } (ir.rs1 = xxw.rd) \textbf{ then } [dx.a := xxw.res] \textbf{ else } nil; \\ \textbf{if } (ir.rs2 = xxw.rd) \textbf{ then } [dx.b := xxw.res] \textbf{ else } nil; \\ (xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1 \end{array}\right]$$

$$E_4 :: \left[\begin{array}{l} (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ (wd.w, wd.res, wd.rd) := (true, xxw.res, xxw.rd); \\ (wx.w, wx.res, wx.rd) := (true, xxw.res, xxw.rd); \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (true, alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ dx := xdx; \\ xw := xxw; \\ instr := mem(pc); \\ ir := instr; \\ pc := pc + 1 \end{array}\right]$$

B.4 Obtaining first Von Neumann Body

Starting from equivalence 6.16 of page 177:

$$Pipeline_1 \ =_{\mathcal{O}} \ I_1; R_3; U_1'; U_1'; U_1'; E_4; E$$

To reach the desired form the concatenation commutativity law is applied to the first $[\, U_1'; U_1' \,]$ concatenation:

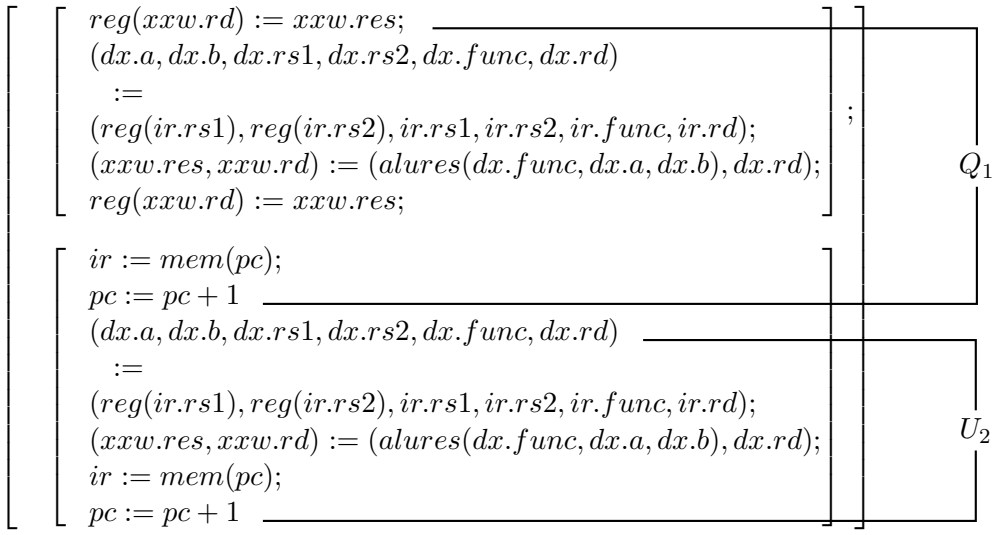
$$\left[\begin{array}{ll} & \left[\begin{array}{l} reg(xxw.rd) := xxw.res; \\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ (xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1 \end{array}\right] \\ U_1' :: & \\ & \left[\begin{array}{l} reg(xxw.rd) := xxw.res; \\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ (xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1 \end{array}\right] \end{array}\right] ;$$

with the following matchings:

$$A :: \quad \left[\begin{array}{l} ir := mem(pc); \\ pc := pc + 1 \end{array}\right]$$

$$B :: reg(xxw.rd) := xxw.res$$

one obtains the expression:



Relabeling the above concatenation, $[\, Q_1; U_2 \,]$ is obtained, where:

$$Q_1 :: \left[\begin{array}{l} reg(xxw.rd) := xxw.res; \\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ (xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd); \\ reg(xxw.rd) := xxw.res; \\ ir := mem(pc); \\ pc := pc + 1 \end{array}\right]$$

$$U_2 :: \left[\begin{array}{l} (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ (xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1 \end{array}\right]$$

and

$$Pipeline_1 \;=_{\mathcal{O}}\; I_1; R_3; Q_1; U_2; U_1'; E_4; E$$

Applying again the concatenation commutativity law to $[\, U_2; U_1' \,]$ with the same matchings, one obtains the expression:

$$\left[\begin{array}{l}\left[\begin{array}{l}(dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd)\\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd);\\ (xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd);\\ reg(xxw.rd) := xxw.res;\end{array}\right];\\ \left[\begin{array}{l}ir := mem(pc);\\ pc := pc + 1\\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd)\\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd);\\ (xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd);\\ ir := mem(pc);\\ pc := pc + 1\end{array}\right]\end{array}\right] \begin{array}{l} U_3 \\ \\ U_2 \end{array}$$

after relabeling, one obtains $[\ U_3; U_2\ ]$, where:

$$U_3 :: \left[\begin{array}{l}(dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd)\\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd);\\ (xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd);\\ reg(xxw.rd) := xxw.res;\\ ir := mem(pc);\\ pc := pc + 1\end{array}\right]$$

The overall interface equivalence becomes the following:

$$Pipeline_1 \ =_{\mathcal{O}}\ [\ I_1; R_3; Q_1; U_3; U_2; E_4; E\ ]$$

Redundant Variable Elimination:

The VN iteration body will be obtained after the elimination of all redundant variables.

Elimination of wx:

Variable wx is removed from $[\ I_1; R_3; Q_1;\ \ldots\ ]$ applying the multiple assignment elimination law (see law 17 of page 41).

$$I_1 :: \left[\begin{array}{l}pc := 1;\\ (ir.rs1, ir.rs2, ir.rd, ir.func) := (0, 0, 0, 0);\\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) := (0, 0, 0, 0, 0, 0);\\ (wx.w, wx.res, wx.rd) := (false, 0, 0)\end{array}\right]$$

and the matchings are:

$$\begin{array}{rcl}(v.v_1, v.v_2, v.v_3) := (e_1, e_2, e_3) & :: & (wx.w, wx.res, wx.rd) := (false, 0, 0) \\ S_1(v) & :: & R_3 \\ S_2 & :: & \text{is empty}\end{array}$$

Since wx is only present in I_1, obtaining:

$$I_1' :: \left[\begin{array}{l} pc := 1; \\ (ir.rs1, ir.rs2, ir.rd, ir.func) := (0,0,0,0); \\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) := (0,0,0,0,0,0)\end{array}\right]$$

and

$$Pipeline_1 \ =_{\mathcal{O}} \ [\ I_1'; R_3; Q_1; U_3; U_2; E_4; E\ ]$$

Elimination of $dx.a, dx.b, dx.rs1, dx.rs2, dx.func$ ***and*** $dx.rd$:

Applying the variable elimination, $dx.a, dx.b, dx.rs1, dx.rs2, dx.func$ and $dx.rd$, will be removed from $[\ I_1'; R_3\ ]$. R_3 is defined in page 259.

The matchings are:

$$\begin{array}{rcl} v.v_i, ..., v.v_j, ..., v.v_n & :: & dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd \\ (v.v_1, ..., v.v_i, ..., v.v_j, ..., v.v_n) := (e_1, ..., e_i, ..., e_j, ..., e_n) & :: & (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) := (0,0,0,0,0,0) \\ S_1(v.v_i, ..., v.v_j, ..., v.v_n) & :: & \left[\begin{array}{l}\textbf{if } (false) \textbf{ then } [reg(0) := 0] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = 0) \textbf{ then } [dx.a := 0] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = 0) \textbf{ then } [dx.b := 0] \textbf{ else } nil; \\ (xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd)\end{array}\right] \\ S_2 & :: & \text{all the statements in sequence after } S_1(v.v_i, ..., v.v_j, ..., v.v_n) \\ & & \text{until a new assignment to } dx.a, dx.b, dx.rs1, dx.rs2, dx.func \text{ and } dx.rd \end{array}$$

one obtains $[\ I_1''; R_3'\ ]$, where:

$$I_1'' :: \left[\begin{array}{l} pc := 1; \\ (ir.rs1, ir.rs2, ir.rd, ir.func) := (0,0,0,0);\end{array}\right]$$

$$R_3' :: \left[\begin{array}{l}\textbf{if } (false) \textbf{ then } [reg(0) := 0] \textbf{ else } nil; \\ \textbf{if } (0 = 0) \textbf{ then } [0 := 0] \textbf{ else } nil; \\ \textbf{if } (0 = 0) \textbf{ then } [0 := 0] \textbf{ else } nil; \\ (xxw.res, xxw.rd) := (alures(0,0,0), 0)\end{array}\right]$$

and

$$Pipeline_1 \ =_{\mathcal{O}} \ [\ I_1''; R_3'; Q_1; U_3; U_2; E_4; E\ ]$$

Simple 'if ' statements Elimination:

The boolean '**if** ' statements of R_3' can be simplified applying the following congruences:

if '$false$' congruence:	**if** $false$ **then** S_1 **else** $S_2 \ \approx \ S_2$
if '*true expression*' congruence:	**if** $Exp = Exp$ **then** S_1 **else** $S_2 \ \approx \ S_1$ where Exp is an expression.

Applying the **if** '$false$' congruence with the matchings:

$$\begin{array}{lcl} S_1 & :: & reg(0) := 0 \\ S_2 & :: & nil \end{array}$$

obtaining R_3''.

$$R_3'' :: \left[\begin{array}{l} nil; \\ \textbf{if } (0=0) \textbf{ then } [0:=0] \textbf{ else } nil; \\ \textbf{if } (0=0) \textbf{ then } [0:=0] \textbf{ else } nil; \\ (xxw.res, xxw.rd) := (alures(0,0,0),0) \end{array}\right]$$

The '*nil*' statement is removed applying the *concatenation with nil* congruence from R_3'':

$$\textbf{nil}; S \approx S$$

where the matching is:

$$S \ :: \ \left[\begin{array}{l} \textbf{if } (0=0) \textbf{ then } [0:=0] \textbf{ else } nil; \\ \textbf{if } (0=0) \textbf{ then } [0:=0] \textbf{ else } nil; \\ (xxw.res, xxw.rd) := (alures(0,0,0),0) \end{array}\right]$$

obtaining R_3''':

$$R_3''' :: \left[\begin{array}{l} \textbf{if } (0=0) \textbf{ then } [0:=0] \textbf{ else } nil; \\ \textbf{if } (0=0) \textbf{ then } [0:=0] \textbf{ else } nil; \\ (xxw.res, xxw.rd) := (alures(0,0,0),0) \end{array}\right]$$

The second congruence is applied twice to eliminate the '**if** $(0 = 0) \ldots$' statements of R_3'''. The matching for both are:

$$\begin{array}{rcl} Exp = Exp & :: & 0 = 0 \\ S_1 & :: & 0 := 0 \\ S_2 & :: & nil \end{array} \qquad \text{and} \qquad \begin{array}{rcl} Exp = Exp & :: & 0 = 0 \\ S_1 & :: & 0 := 0 \\ S_2 & :: & nil \end{array}$$

one obtains R_4.

$$R_4 :: \left[\;\; (xxw.res, xxw.rd) := (alures(0, dx.a, dx.b), 0) \right]$$

Now the following equivalence has been reached:

$$Pipeline_1 \;=_{\mathcal{O}}\; [\, I_1''; R_4; Q_1; U_3; U_2; E_4; E \,]$$

Elimination of dx:

The multiple assignment to dx is eliminated from Q_1 with the matchings:

$$\begin{array}{rcl} (v.v_1, v.v_2, ..., v.v_6) & & \left[\begin{array}{l} (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array} \right] \\ := & :: & \\ (e_1, e_2, ..., e_6) & & \end{array}$$

$$S_1(v) \;\; :: \;\; \left[\;\; (xxw.res, xxw.rd) := (alures(dx.func, dx.a, dx.b), dx.rd) \;\; \right]$$

$$S_2 \;\; :: \;\; \text{all the statements in sequence after } S_1(v) \text{ until a new assignment to } dx$$

obtaining Q_1':

$$Q_1' :: \left[\begin{array}{l} reg(xxw.rd) := xxw.res; \\ (xxw.res, xxw.rd) := (alures(ir.func, reg(ir.rs1), reg(ir.rs2)), ir.rd); \\ reg(xxw.rd) := xxw.res; \\ ir := mem(pc); \\ pc := pc + 1 \end{array} \right]$$

Variable dx is removed from U_3 with the same matchings as above, obtaining U_3':

$$U_3' :: \left[\begin{array}{l} (xxw.res, xxw.rd) := (alures(ir.func, reg(ir.rs1), reg(ir.rs2)), ir.rd); \\ reg(xxw.rd) := xxw.res; \\ ir := mem(pc); \\ pc := pc + 1 \end{array} \right]$$

and similarly for U_2, obtaining U_2':

$$U_2' :: \left[\begin{array}{l} (xxw.res, xxw.rd) := (alures(ir.func, reg(ir.rs1), reg(ir.rs2)), ir.rd); \\ ir := mem(pc); \\ pc := pc + 1 \end{array}\right]$$

At this stage one has:

$$Pipeline_1 \ =_{\mathcal{O}} \ [\ I_1''; R_4; Q_1'; U_3'; U_2'; E_4; E\ ]$$

Elimination of xxw:

First variables $xxw.res$ and $xxw.rd$ are eliminated from $R_4; Q_1'$ with the matchings:

$$\begin{array}{rcl} (v.v_1, v.v_2) := (e_1, e_2) & :: & [\ (xxw.res, xxw.rd) := (alures(0,0,0), 0)\ ] \\ S_1(v) & :: & [\ reg(xxw.rd) := xxw.res\ ] \\ S_2 & :: & \text{all the statements in sequence after } S_1(v) \\ & & \text{until a new assignment to } xxw.res, xxw.rd \end{array}$$

obtaining Q_1'', since R_4 has disappeared:

$$Q_1'' :: \left[\begin{array}{l} reg(0) := (alures(0,0,0); \\ (xxw.res, xxw.rd) := (alures(ir.func, reg(ir.rs1), reg(ir.rs2)), ir.rd); \\ reg(xxw.rd) := xxw.res; \\ ir := mem(pc); \\ pc := pc + 1 \end{array}\right]$$

Removing again variables $xxw.res$ and $xxw.rd$ from Q_1'' and U_3' with the matchings:

$$\begin{array}{rcl} (v.v_1, v.v_2, ..., v.v_6) := (e_1, e_2, ..., e_6) & :: & [\ (xxw.res, xxw.rd) := (alures(ir.func, reg(ir.rs1), reg(ir.rs2)), ir.rd)\ ] \\ S_1(v) & :: & [\ reg(xxw.rd) := xxw.res\ ] \\ S_2 & :: & \text{all the statements in sequence after } S_1(v) \\ & & \text{until a new assignment to } xxw \end{array}$$

one obtains Q_1''' and U_3'':

$$Q_1''' :: \left[\begin{array}{l} reg(0) := (alures(0,0,0); \\ reg(ir.rd) := alures(ir.func, reg(ir.rs1), reg(ir.rs2)); \\ ir := mem(pc); \\ pc := pc + 1 \end{array}\right]$$

$$U_3'' :: \left[\begin{array}{l} reg(ir.rd) := alures(ir.func, reg(ir.rs1), reg(ir.rs2)); \\ ir := mem(pc); \\ pc := pc + 1 \end{array}\right]$$

At this stage one has:

$$Pipeline_1 =_{\mathcal{O}} [\, I_1''; Q_1'''; U_3''; U_2'; E_4; E \,]$$

Elimination of ir:

The variable ir is removed from $I_1''; Q_1'''$ with the matchings:

$$\begin{array}{rcl} (v.v_1, v.v_2, v.v_3, v.v_4) := (e_1, e_2, e_3, e_4) & :: & \left[\; (ir.rs1, ir.rs2, ir.rd, ir.func) := (0,0,0,0) \;\right] \\ S_1(v) & :: & \left[\begin{array}{l} reg(0) := (alures(0,0,0); \\ reg(xxw.rd) := xxw.res \end{array}\right] \\ S_2 & :: & \text{all the statements in sequence after } S_1(v) \\ & & \text{until a new assignment to } ir \end{array}$$

obtaining $I_1'''; Q_1''''$:

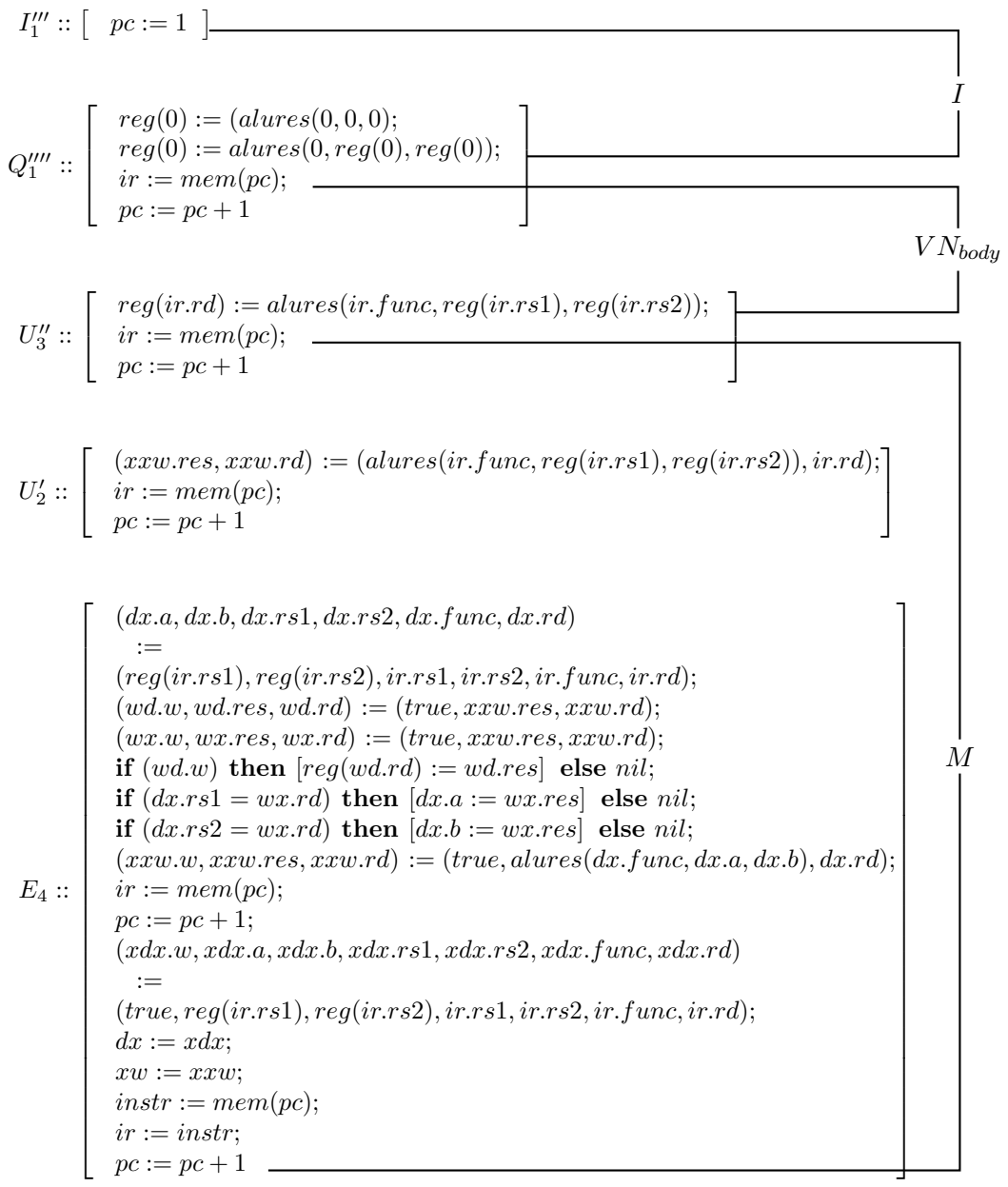
$$I_1''' :: \left[\;\; pc := 1\right]$$

$$Q_1'''' :: \left[\begin{array}{l} reg(0) := (alures(0,0,0); \\ reg(0) := alures(0, reg(0), reg(0)); \\ ir := mem(pc); \\ pc := pc + 1 \end{array}\right]$$

The obtained equivalence is:

$$Pipeline_1 =_{\mathcal{O}} [\, I_1'''; Q_1''''; U_3''; U_2'; E_4; E \,]$$

The above expression can be relabeled to match equivalence 6.17 of page 177:



reaching:

$$Pipeline_1 =_{\mathcal{O}} I; VN_{body}; M; E$$

B.5 Tail Statements

This section starts from equivalence 6.19 of page 179:

$$M;\ E\ =_{\mathcal{O}}\ M;\ P_0;\ E$$

Statement P_0 of page 168, is reduced as in page 237, after the parallelism to concatenation transformation one obtains $P1$, page 239.

Next step is detailed in section B.2 of this appendix. P_1 is reduced to P_1', page 240, after applying the concatenation commutativity transformation.

Now the interface equivalence is:

$$M;\ E\ =_{\mathcal{O}}\ M;\ P_1';\ E$$

The proof continues by applying once again the *concatenation commutativity* lemma ($[\ A;\ B\ ]\ =_{\mathcal{O}}\ [\ B; A\ ]$) to P_1' with the matchings:

$$A :: \left[\begin{array}{l} instr := mem(pc); \\ ir := instr; \\ pc := pc + 1 \end{array}\right]$$

$$B :: \left[\begin{array}{l} \textbf{if}\ (dx.rs1 = wx.rd)\ \textbf{then}\ [dx.a := wx.res]\ \textbf{else}\ nil; \\ \textbf{if}\ (dx.rs2 = wx.rd)\ \textbf{then}\ [dx.b := wx.res]\ \textbf{else}\ nil; \\ xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ dx := xdx; \\ xw := xxw \end{array}\right]$$

obtaining G_1:

$$G_1 :: \left[\begin{array}{l} wd := xw; \\ wx := xw; \\ \textbf{if}\ (wd.w)\ \textbf{then}\ [reg(wd.rd) := wd.res]\ \textbf{else}\ nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ \textbf{if}\ (dx.rs1 = wx.rd)\ \textbf{then}\ [dx.a := wx.res]\ \textbf{else}\ nil; \\ \textbf{if}\ (dx.rs2 = wx.rd)\ \textbf{then}\ [dx.b := wx.res]\ \textbf{else}\ nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ dx := xdx; \\ xw := xxw; \\ instr := mem(pc); \\ ir := instr; \\ pc := pc + 1 \end{array}\right]$$

The new interface equivalence is the following:

$$M;\ E\ =_{\mathcal{O}}\ M;\ G_1;\ E$$

After eliminating variables wd and wx from M, shown in page 178, and propagating the value of variables $dx.rs1$ and $dx.rs2$ of M, obtaining M':

$$M' :: \left[\begin{array}{l}
ir := mem(pc);\\
pc := pc + 1;\\
(xxw.res, xxw.rd) := (alures(ir.func, reg(ir.rs1), reg(ir.rs2)), ir.rd);\\
ir := mem(pc);\\
pc := pc + 1;\\
(dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd)\\
\quad :=\\
(reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd);\\
\textbf{if}\ (true)\ \textbf{then}\ [reg(xxw.rd) := xxw.res]\ \textbf{else}\ nil;\\
\textbf{if}\ (ir.rs1 = xxw.rd)\ \textbf{then}\ [dx.a := xxw.res]\ \textbf{else}\ nil;\\
\textbf{if}\ (ir.rs2 = xxw.rd)\ \textbf{then}\ [dx.b := xxw.res]\ \textbf{else}\ nil;\\
(xxw.w, xxw.res, xxw.rd) := (true, alures(dx.func, dx.a, dx.b), dx.rd);\\
ir := mem(pc);\\
pc := pc + 1;\\
(xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd)\\
\quad :=\\
(true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd);\\
dx := xdx;\\
xw := xxw;\\
instr := mem(pc);\\
ir := instr;\\
pc := pc + 1
\end{array}\right]$$

Applying the '**if** ' true congruence to the following statement of M',

$$\textbf{if}\ (true)\ \textbf{then}\ [reg(xxw.rd) := xxw.res]\ \textbf{else}\ nil$$

and the *Data Forwarding Elimination*, page 175, one obtains M'':

$$M'' :: \left[\begin{array}{l} ir := mem(pc); \\ pc := pc + 1; \\ (xxw.res, xxw.rd) := (alures(ir.func, reg(ir.rs1), reg(ir.rs2)), ir.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ reg(xxw.rd) := xxw.res; \\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ (xxw.w, xxw.res, xxw.rd) := (true, alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ dx := xdx; \\ xw := xxw; \\ instr := mem(pc); \\ ir := instr; \\ pc := pc + 1 \end{array}\right]$$

To reach the desired form the concatenation commutativity is applied to M'' with the following matchings:

$$A :: \left[\begin{array}{l} ir := mem(pc); \\ pc := pc + 1 \end{array}\right]$$

$$B :: reg(xxw.rd) := xxw.res$$

obtaining M''', where:

$$M''' :: \left[\begin{array}{l} ir := mem(pc); \\ pc := pc + 1; \\ (xxw.res, xxw.rd) := (alures(ir.func, reg(ir.rs1), reg(ir.rs2)), ir.rd); \\ reg(xxw.rd) := xxw.res; \\ ir := mem(pc); \\ pc := pc + 1; \\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ (xxw.w, xxw.res, xxw.rd) := (true, alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ dx := xdx; \\ xw := xxw; \\ instr := mem(pc); \\ ir := instr; \\ pc := pc + 1 \end{array}\right]$$

Variables $xxw.res$ and $xxw.rd$ is removed from M''' applying the *multiple assignment elimination* law (see law 17 of page 41). The matchings are:

$$(v.v_1, v.v_2) := (e_1, e_2) \quad :: \quad \left[\begin{array}{l} (xxw.res, xxw.rd) \\ \quad := \\ (alures(ir.func, reg(ir.rs1), reg(ir.rs2)), ir.rd) \end{array}\right]$$

$$S_1(v) \quad :: \quad [\ reg(xxw.rd) := xxw.res\ ]$$

$$S_2 \quad :: \quad \begin{array}{l}\text{all the statements in sequence after } S_1(v) \\ \text{until a new assignment to } xxw.res, xxw.rd\end{array}$$

obtaining the concatenation $[\ VN_{body}; M_1\ ]$, where

$$M_1 :: \left[\begin{array}{l} ir := mem(pc); \\ pc := pc + 1; \\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ (xxw.w, xxw.res, xxw.rd) := (true, alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ dx := xdx; \\ xw := xxw; \\ instr := mem(pc); \\ ir := instr; \\ pc := pc + 1 \end{array}\right]$$

At this stage the resulting interface equivalence is:

$$M;\ E\ =_{\mathcal{O}}\ VN_{body}; M_1;\ G_1;\ E$$

Now the concatenation $[\ M_1; G_1\ ]$ is reduced. First, redundant variables $dx.a, dx.b, \ldots$ and $xdx.w, xdx.a, \ldots$ are eliminated from M_1, obtaining M_1':

$$M_1' :: \left[\begin{array}{l} ir := mem(pc); \\ pc := pc + 1; \\ (xxw.w, xxw.res, xxw.rd) := (true, alures(ir.func, reg(ir.rs1), reg(ir.rs2)), ir.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (dx.w, dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ xw := xxw; \\ instr := mem(pc); \\ ir := instr; \\ pc := pc + 1 \end{array}\right]$$

Next step applies the *concatenation commutativity* to G_1 with the following matchings:

$$A :: \left[\begin{array}{l} (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd) \end{array}\right]$$

$$B :: \left[\begin{array}{l} \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd) \end{array}\right]$$

one obtains G_1':

$$G_1' :: \left[\begin{array}{l} wd := xw; \\ wx := xw; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ dx := xdx; \\ xw := xxw; \\ instr := mem(pc); \\ ir := instr; \\ pc := pc + 1 \end{array}\right]$$

Applying the *concatenation commutativity* to [$M_1'; G_1'$] with the following matchings:

$$A :: \left[\begin{array}{l} instr := mem(pc); \\ ir := instr; \\ pc := pc + 1 \end{array}\right]$$

$$B :: \left[\begin{array}{l} wd := xw; \\ wx := xw; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd) \end{array}\right]$$

obtaining $[\ M_1'';G_1''\ ]$:

$$
\left[\begin{array}{ll}
 & \left[\begin{array}{l}
ir := mem(pc);\\
pc := pc+1;\\
(xxw.w, xxw.res, xxw.rd)\\
\quad :=\\
(true, alures(ir.func, reg(ir.rs1), reg(ir.rs2)), ir.rd);\\
ir := mem(pc);\\
pc := pc+1;\\
(dx.w, dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd)\\
\quad :=\\
(true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd);\\
xw := xxw
\end{array}\right]\\
M_1'' :: & \\
 & \left[\begin{array}{l}
wd := xw;\\
wx := xw;\\
\textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil;\\
\textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil;\\
\textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil;\\
(xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd);\\
instr := mem(pc);\\
ir := instr;\\
pc := pc+1\\
(xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd)\\
\quad :=\\
(true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd);\\
dx := xdx;\\
xw := xxw;\\
instr := mem(pc);\\
ir := instr;\\
pc := pc+1
\end{array}\right]\\
G_1'' :: &
\end{array}\right]
$$

Next step eliminates the component variable $dx.w$ of M_1'' obtaining M_1'''. Finally, the first assignment to variable $instr$ of G_1'' is eliminated and obtaining G_1''':

$$M_1''' :: \left[\begin{array}{l} ir := mem(pc); \\ pc := pc + 1; \\ (xxw.w, xxw.res, xxw.rd) \\ \quad := \\ (true, alures(ir.func, reg(ir.rs1), reg(ir.rs2)), ir.rd); \\ ir := mem(pc); \\ pc := pc + 1; \\ (dx.a, dx.b, dx.rs1, dx.rs2, dx.func, dx.rd) \\ \quad := \\ reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ xw := xxw \end{array}\right]$$

$$G_1''' :: \left[\begin{array}{l} wd := xw; \\ wx := xw; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (true, alures(dx.func, dx.a, dx.b), dx.rd); \\ ir := mem(pc); \\ pc := pc + 1 \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ dx := xdx; \\ xw := xxw; \\ instr := mem(pc); \\ ir := instr; \\ pc := pc + 1 \end{array}\right]$$

and

$$[\ M_1'''; \ G_1'''] \ =_{\mathcal{O}} \ M$$

Therefore, the equivalence 6.18 of page 178 have been proved:

$$M; \ E \ =_{\mathcal{O}} \ VN_{body}; \ M; \ E$$

Appendix C

PROOF OF ID_{PAR} AND EX_{PAR}

C.1 Proof of ID_{par}

Starting from equivalence 6.22 of page 181:

$$ID_{par} =_{\mathcal{O}} P_0; E_0$$

Obtaining first ID_{seq_unh} body:

The inner cooperation statements which appeared in P_0, page 182, as a result of the communication elimination, are transformed to sequential statements by applying *Cooperation and Concatenation* transformation procedure of law 11 of page 39. The parallelism to concatenation transformations within P_0 are the following ones:

$$\left\{ \left[\left[c := true\right] \quad || \quad \begin{bmatrix} cwdW \Rightarrow wd.w; \\ cwdRES \Rightarrow wd.res; \\ cwdRD \Rightarrow wd.rd \end{bmatrix} \right] \quad => \quad \begin{bmatrix} cwdW \Rightarrow wd.w; \\ cwdRES \Rightarrow wd.res; \\ cwdRD \Rightarrow wd.rd; \\ c := true \end{bmatrix} \right\}$$

$$\left\{ \left[\left[cfd \Rightarrow fd\right] \;||\; \left[\begin{array}{l}\textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil;\\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd)\\ \quad :=\\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd)\end{array}\right] \right] \right.$$

$$=>$$

$$\left. \left[\begin{array}{l}\textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil;\\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd)\\ \quad :=\\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd);\\ cfd \Rightarrow fd\end{array}\right] \right\}$$

and,

$$\left\{ \left[\left[c := true\right] \;||\; \left[\begin{array}{l}cdxW \Leftarrow xdx.w;\\ cdxA \Leftarrow xdx.a;\\ cdxB \Leftarrow xdx.b;\\ cdxRS1 \Leftarrow xdx.rs1;\\ cdxRS2 \Leftarrow xdx.rs2;\\ cdxFUNC \Leftarrow xdx.func;\\ cdxRD \Leftarrow xdx.rd;\\ cwdW \Rightarrow wd.w;\\ cwdRES \Rightarrow wd.res;\\ cwdRD \Rightarrow wd.rd\end{array}\right] \right] \right.$$

$$=>$$

$$\left. \left[\begin{array}{l}cdxW \Leftarrow xdx.w;\\ cdxA \Leftarrow xdx.a;\\ cdxB \Leftarrow xdx.b;\\ cdxRS1 \Leftarrow xdx.rs1;\\ cdxRS2 \Leftarrow xdx.rs2;\\ cdxFUNC \Leftarrow xdx.func;\\ cdxRD \Leftarrow xdx.rd;\\ cwdW \Rightarrow wd.w;\\ cwdRES \Rightarrow wd.res;\\ cwdRD \Rightarrow wd.rd;\\ c := true\end{array}\right] \right\}$$

An intermediate P_0' statement is obtained after the transformations.

Lemma 20 (Variable Grouping): Let v and w be variables of type T, where type T is the cartesian product of the fields f_1, f_2, ..., f_n. The next sequence of assignments:

$$\left[\begin{array}{l}v.f_1 := w.f_1;\\ v.f_2 := w.f_2;\\ \ldots\\ v.f_n := w.f_n\end{array}\right]$$

can be expressed as $v := w$.

The last step removes redundant variable assignments from the above P_0'. These are c, w, and fd. Before eliminating variable fd lemma 20 is applied, it is expressed as:

$$\left\{ \begin{bmatrix} ir.rs1 := fd.rs1; \\ ir.rs2 := fd.rs2; \\ ir.rd := fd.rd; \\ ir.func := fd.func \end{bmatrix} \Rightarrow \; [ir := fd] \right\}$$

After the above reductions, the final equivalence is obtained:

$$ID_{par} \; =_{\mathcal{O}} \; ID_{seq\_unh\_body}; P_1; E_0$$

where $ID_{seq\_unh\_body}$ and P_1 have the forms:

$$ID_{seq\_unh\_body} :: \begin{bmatrix} cwdW \Rightarrow wd.w; \\ cwdRES \Rightarrow wd.res; \\ cwdRD \Rightarrow wd.rd; \\ \textbf{if } (wd.w) \textbf{ then } [reg(wd.rd) := wd.res] \textbf{ else } nil; \\ (xdx.w, xdx.a, xdx.b, xdx.rs1, xdx.rs2, xdx.func, xdx.rd) \\ \quad := \\ (true, reg(ir.rs1), reg(ir.rs2), ir.rs1, ir.rs2, ir.func, ir.rd); \\ cfd \Rightarrow ir; \\ cdxW \Leftarrow xdx.w; \\ cdxA \Leftarrow xdx.a; \\ cdxB \Leftarrow xdx.b; \\ cdxRS1 \Leftarrow xdx.rs1; \\ cdxRS2 \Leftarrow xdx.rs2; \\ cdxFUNC \Leftarrow xdx.func; \\ cdxRD \Leftarrow xdx.rd \end{bmatrix}$$

$$P_1 :: \begin{bmatrix} cwdW \Rightarrow wd.w; \\ cwdRES \Rightarrow wd.res; \\ cwdRD \Rightarrow wd.rd; \\ c := true; \\ w := c \end{bmatrix}$$

and E_0 is shown in page 183.

Induction step:

The proof continues by unfolding once the indefinite loops. Applying again all the above transformations (the communication elimination, the cooperation to concatenation, and the variable elimination), a new $ID_{seq\_unh\_body}$ and the same tail statements are obtained:

$$ID_{par} \; =_{\mathcal{O}} \; ID_{seq\_unh\_body}; ID_{seq\_unh\_body}; P_1; E_0$$

After ∞ unfoldings and transformations, the equivalence may be reduced to:

$$ID_{par} =_{\mathcal{O}} \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[ID_{seq\_unh\_body}\right]\end{array}\right]$$

then,

$$ID_{par} =_{\mathcal{O}} ID_{seq\_unh}$$

Note that P_1 and E_0 do not appear since they will never be reached.

C.2 Proof of EX_{par}

Starting from equivalence 6.24 of page 187:

$$EX_{par} =_{\mathcal{O}} P_0; Q_0; E_0$$

Obtaining first EX_{seq_unh} body:

The inner cooperation statements which appeared in P_0, page 187, as a result of the communication elimination, are transformed to sequential statements. The parallelism to concatenation transformations applied to P_0 are the following:

$$\left\{ \left[\begin{bmatrix} cwx \Rightarrow wx; \\ resA := wx.res \end{bmatrix} \;\|\; [b := dx.b] \right] => \begin{bmatrix} b := dx.b; \\ cwx \Rightarrow wx; \\ resA := wx.res \end{bmatrix} \right\}$$

$$\left\{ \left[[resB := wx.res] \;\|\; \begin{bmatrix} rs1 := dx.rs1; \\ rs2 := dx.rs2 \end{bmatrix} \right] => \begin{bmatrix} resB := wx.res; \\ rs1 := dx.rs1; \\ rs2 := dx.rs2 \end{bmatrix} \right\}$$

and,

$$\left\{ \left[[selB := selectB] \;\|\; \begin{bmatrix} \textbf{if } selA \textbf{ then } [outA := resA] \textbf{ else } [outA := a]; \\ aluA := outA \end{bmatrix} \right] \right.$$

$$\left. => \begin{bmatrix} \textbf{if } selA \textbf{ then } [outA := resA] \textbf{ else } [outA := a]; \\ aluA := outA; \\ selB := selectB \end{bmatrix} \right\}$$

An intermediate P'_0 statement is obtained after the transformations.

The following cooperation and concatenation transformation is applied twice to Q_0 statement, from page 188, obtaining Q_1:

$$\left\{ \begin{bmatrix} [selB := selectB] \\ \| \\ \begin{bmatrix} \begin{bmatrix} \textbf{if } selA \textbf{ then } [outA := resA] \textbf{ else } [outA := a] \end{bmatrix} \\ \| \quad \begin{bmatrix} xxw.res := alures(func, aluA, aluB); \\ cxwRES \Leftarrow xxw.res \end{bmatrix}; \\ aluA := outA \end{bmatrix} \end{bmatrix} \right.$$

$$\Rightarrow \quad \left. \begin{bmatrix} xxw.res := alures(func, aluA, aluB); \\ cxwRES \Leftarrow xxw.res; \\ \textbf{if } selA \textbf{ then } [outA := resA] \textbf{ else } [outA := a]; \\ aluA := outA; \\ selB := selectB \end{bmatrix} \right\}$$

The last step removes the following redundant variable assignments from P'_0, obtaining $EX_{seq\_unh\_body}$: a, b, $resA$, $resB$, $rs1$, $rs2$, rd, $selectA$, $selectB$, $selA$, and $selB$.

The final equivalence is:

$$EX_{par} =_{\mathcal{O}} EX_{seq\_unh\_body}; Q_1; E_0$$

where $EX_{seq\_unh\_body}$, and $Q1$ have the next forms:

$$EX_{seq\_unh\_body} ::$$

$$\begin{bmatrix} cwx \Rightarrow wx; \\ \textbf{if } (dx.rs1 = wx.rd) \textbf{ then } [dx.a := wx.res] \textbf{ else } nil; \\ \textbf{if } (dx.rs2 = wx.rd) \textbf{ then } [dx.b := wx.res] \textbf{ else } nil; \\ (xxw.w, xxw.res, xxw.rd) := (dx.w, alures(dx.func, dx.a, dx.b), dx.rd); \\ cdxW \Rightarrow dx.w; \\ cdxA \Rightarrow dx.a; \\ cdxB \Rightarrow dx.b; \\ cdxRS1 \Rightarrow dx.rs1; \\ cdxRS2 \Rightarrow dx.rs2; \\ cdxFUNC \Rightarrow dx.func; \\ cdxRD \Rightarrow dx.rd; \\ cxwW \Leftarrow xxw.w; \\ cxwRES \Leftarrow xxw.res; \\ cxwRD \Leftarrow xxw.rd \end{bmatrix}$$

$$
Q_1 :: \left[\begin{array}{l}
a := dx.a; \\
b := dx.b; \\
cwx \Rightarrow wx; \\
resA := wx.res; \\
resB := wx.res; \\
rs1 := dx.rs1; \\
rs2 := dx.rs2; \\
rd := wx.rd; \\
selectA := (rs1 = rd); \\
selectB := (rs2 = rd); \\
selA := selectA; \\
\textbf{if } selA \textbf{ then } [outA := resA] \textbf{ else } [outA := a]; \\
aluA := outA; \\
selB := selectB; \\
\textbf{if } selB \textbf{ then } [outB := resB] \textbf{ else } [outB := b]; \\
aluB := outB; \\
func := dx.func; \\
xxw.w := dx.w; \\
xxw.rd := dx.rd; \\
cdxW \Rightarrow dx.w; \\
cdxA \Rightarrow dx.a; \\
cdxB \Rightarrow dx.b; \\
cdxRS1 \Rightarrow dx.rs1; \\
cdxRS2 \Rightarrow dx.rs2; \\
cdxFUNC \Rightarrow dx.func; \\
cdxRD \Rightarrow dx.rd; \\
cxwW \Leftarrow xxw.w; \\
cxwRD \Leftarrow xxw.rd
\end{array}\right]
$$

and E_0 is shown in page 189.

Induction Step:

The proof continues by applying again all the above transformations to $Q_1; E_0$. A new $EX_{seq\_unh\_body}$ and the same tail statements are obtained:

$$EX_{par} =_{\mathcal{O}} EX_{seq\_unh\_body}; EX_{seq\_unh\_body}; Q_1; E_0$$

After ∞ unfoldings and communication elimination transformations, the equivalence may be reduced to:

$$EX_{par} =_{\mathcal{O}} \left[\begin{array}{l}\textbf{loop forever do} \\ \quad \left[EX_{seq\_unh\_body}\right]\end{array}\right]$$

then,

$$EX_{par} =_{\mathcal{O}} EX_{seq\_unh}$$

Q_1 and E_0 do not appear in the expression since they will never be reached.

BIBLIOGRAPHY

[AC75] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[BBC+95] N.S. Bjørner, A. Browne, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562, Stanford University, Computer Science Department, November 1995.

[BBC05a] Francesc Babot, Miquel Bertran, and August Climent. A Static Communication Elimination Algorithm for Distributed System Verification. In Kung-Kiu Lau and Richard Banach, editors, *Formal Methods and Software Engineering. 7th International Conference on Formal Engineering Methods, ICFEM 2005*, volume 3785 of *LNCS*, pages 375–389, Manchester, England, November 2005. Springer.

[BBC05b] Miquel Bertran, Francesc-Xavier Babot, and August Climent. An Input/output Semantics for Distributed Program Equivalence Reasoning. *Electronic Notes in Theoretical Computer Science*, 137(1):25–46, July 2005.

[BBCN01] Miquel Bertran, Francesc Babot, August Climent, and Miquel Nicolau. Communication and Parallelism Introduction and Elimination in Imperative Concurrent Programs. In Patrick Cousot, editor, *Static Analysis. 8th International Symposium, SAS 2001*, volume 2126 of *LNCS*, pages 20–39, Paris, France, July 2001. Springer.

[BBMC+00] N.S. Bjørner, A. Browne, B. Finkbeiner M. Colón, Z. Manna, H.B. Sipma, and T.E. Uribe. Verifying Temporal Properties of Reactive Systems. A Step Tutorial. In *Formal Methods in System Design*, pages 227–270, June 2000.

[BDP+09] Miquel Bertran, Albert Duran, Miquel Porta, Joan-Andreu Margalef, Roman Duch, and Francesc Babot. PADD Reference Manuals, `http://www.gsystemsd.com/paddrale.htm`, April 2009.

[Ber88] Miquel Bertran. On a Formal Definition and Application of Dimensional Design. *Software-Practice and Experience*, 18(11):1029–1045, November 1988.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[BK84] Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[BK85] Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstraction. *Theor. Comput. Sci.*, 37:77–121, 1985.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* MIT Press, 2008.

[Bro97] Manfred Broy. Refinement of time. In M. Bertran and T. Rus, editors, *4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software*, volume 1231 of *LNCS*, pages 44–63. Springer, January 1997.

[Bro99] Manfred Broy. A logical basis for component-based systems engineering. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design.* IOS Press, 1999.

[Bro01] Manfred Broy. Refinement of Time. *Theoretical Computer Science*, 253(1):3–26, February 2001.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Yorktown Heights, 1981. Springer.

[CGL94] Edmund M. Clarke, Orna Grumberg, and D.E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* The MIT Press, 1999.

[CRTC99] Rafael Corchuelo, David Ruiz, Miguel Toro, and Antonio Ruiz Cortés. Implementing multiparty interactions on a network computer. In *25th Euromicro Conference (EUROMICRO '99)*, pages 2458–2465, Milan, Italy, 1999. IEEE Computer Society.

[CS99] David E. Culler and Jaswinder P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., USA, 1999.

[dA03] L. de Alfaro. Game Models for Open Systems. In *International Symposium on Verification (Theory and Practice)*, volume 2772 of *LNCS*, pages 269–289. Springer, 2003.

[Dep83] Department of Defense. *Reference Manual for the Ada Programming Language*, 1983. ANSI/MIL-STD-1815A.

[dFEGR05] David de Frutos-Escrig and Carlos Gregorio-Rodríguez. Bisimulations up-to for the linear time-branching time spectrum. In *CONCUR 2005 - Concurrency Theory*, volume 3653 of *LNCS*, pages 278–292, London, UK, 2005. Springer-Verlag.

[dFEGR09] David de Frutos-Escrig and Carlos Gregorio-Rodríguez. (Bi)simulations up-to characterise process semantics. *Information and Computation*, 207(2):146–170, 2009.

[dFEVGR07] David de Frutos-Escrig, Fernando Rosa Velardo, and Carlos Gregorio-Rodríguez. New bisimulation semantics for distributed systems. In John Derrick and Jüri Vain, editors, *FORTE*, volume 4574 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2007.

[dFS98] Nicoletta de Francesco and Antonella Santone. A Transformation System for Concurrent Processes. *Acta Informatica*, 35(12):1037–1073, December 1998.

[dMGMJS07] María del Mar Gallardo, Pedro Merino, Christophe Joubert, and David Sanán. On-the-fly model checking for c programs with extended cadp in fmics-jeti. In *12th International Conference on Engineering of Complex Computer Systems (ICECCS 2007)*, pages 321–329, Auckland, New Zealand, July 2007. IEEE Computer Society.

[dMGMS08] María del Mar Gallardo, Pedro Merino, and David Sanán. Model checking c programs with dynamic memory allocation. In *Jornadas sobre Programación y Lenguajes, PROLE 2008*, pages 195–209, Gijón, Spain, October 2008.

[dRdBH+01] Willem-Paul de Roever, Franck de Boer, Ulrich Hanneman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositonal and Noncompositional Methods*. Cambridge University Press, 2001.

[EF82] Tzilla Elrad and Nissim Francez. Decomposition of Distributed Programs into Communication Closed Layers. *Science of Computer Programming*, 2:155–173, 1982.

[FF96] Nissim Francez and Ira R Forman. *Interacting Processes. A Multiparty Approach to Coordinated Distributed Programming.* Addison-Wesley, 1996.

[FMS98] B. Finkbeiner, Z. Manna, and H. Sipma. Deductive Verification of Modular Systems. In *In Compositionality: The Significant Difference, COMPOS'97*, volume 1536 of *LNCS*, pages 239–275. Springer, July 1998.

[GSD] GSD. General Systems Development company, `http://www.gsystemsd.com`.

[GTL02] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. Polychrony for System Design. *Journal of Circuits, Systems and Computers. Application Specific Hardware Design*, August 2002.

[HBG+06] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Modular System Programming in MINIX 3. *;login: The USENIX Magazine*, 31(2):19–28, April 2006.

[HO82] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, 1982.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[Hol91] Gerald Holtzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Mateo, California, 1990.

[IL85] INMOS-Limited. *Occam Programming Manual.* Prentice Hall, 1985.

[IL88] INMOS-Limited. *Occam 2 Reference Manual.* Prentice Hall, 1988.

[Jon87] G. Jones. *Programming in Occam.* Prentice Hall, 1987.

[KLM+98] R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Static Partial Order Reduction. In B. Steffen, editor, *Proceedings of TACAS'98*, volume 1384 of *LNCS*, pages 335–357, Noordwijkerhout, The Netherlands, June 1-4 1998. Springer.

[KM97] Matt Kaufmann and J. Strother Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, 1997.

[KM09] Matt Kaufmann and J. Strother Moore. *The ACL2 Home Page*. `http://www.cs.utexas.edu/users/moore/acl2/`. Dept. of Computer Sciences, University of Texas at Austin, 2009.

[Lee06] Edward A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33–42, May 2006.

[MBSU98] Zohar Manna, Anca Browne, Henny Sipma, and Tomas Uribe. Visual Abstraction for Temporal Verification. In *Algebraic Methods and Software Technology, AMAST'98*, volume 1548 of *LNCS*, pages 28–41. Springer, 1998.

[MD93] K.L. McMillan and D.L. Dill. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.

[Mer01] S. Merz. Model checking: A tutorial overview. In F.Cassez, editor, *Modeling and Verification of Parallel Processes*, volume 2067 of *LNCS*, pages 3–38. Springer, 2001.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[MOSS99] M.Muller-Olm, D.A. Schmit, and B. Steffen. Model Checking: A Tutorial Introduction. In G.File A.Cortesi, editor, *Static Analysis, Proc. 6th Intl. Symp. SAS'99*, volume 1694 of *LNCS*, pages 330–354, Venice, Italy, September 1999. Springer.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Specification*. Springer, 1991.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems. Safety*. Springer, 1995.

[OSRSC01a] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001.

[OSRSC01b] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001.

[PCT04] José A. Pérez, Rafael Corchuelo, and Miguel Toro. An order-based algorithm for implementing multiparty synchronization: Research articles. *Concurrency and Computation: Practice and Experience*, 16(12):1173–1206, 2004.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Symposium on Programming*, pages 337–351, 1982.

[Ram00] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages Systems*, 22(2):416–430, 2000.

[RH88] A.W. Roscoe and C.A.R. Hoare. The laws of OCCAM programming. *Theoretical Computer Science*, 60:177–229, 1988.

[RR92] R. Ramesh and I. V. Ramakrishnan. Nonlinear pattern matching in trees. *J. ACM*, 39(2):295–316, 1992.

[Sch99] Michael Schenke. Transformation Design for Real-Time Systems. part ii: From Program Specifications to Programs. *Acta Informatica*, 36(1):67–96, January 1999.

[SJ00] Qin Shengchao and He Jifeng. An algebraic approach to hardware/software partitioning. In *The 7th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2000)*, pages 273–276, Jounieh, Lebanon, Decembre 17-20 2000. IEEE Computer Society Press.

[SO99] Michael Schenke and Ernst-Rüdiger Olderog. Transformation Design for Real-Time Systems. part i: From Requirements to Program Specifications. *Acta Informatica*, 36(1):1–65, January 1999.

[SORSC01] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001.

[SSB04] Leila Silva, Augusto Sampayo, and Edna Barros. A Constructive Approach to Hardware/Software Partitioning. *Formal Methods in System Design*, 24(1):45–90, 2004.

[SUM99] Henny B. Sipma, Tomas E. Uribe, and Zohar Manna. Deductive Model Checking. *Formal Methods in System Design*, 15(1):49–74, July 1999.

[SZ97] Dennis Shasha and Kaizhong Zhang. Approximate tree pattern matching. In *Pattern Matching Algorithms*, pages 341–371. Oxford University Press, 1997.

[Tay83] Richard N. Taylor. Complexity of Analyzing the Synchronization Sof Concurrent Programs. *Acta Informatica*, 19:57–84, 1983.

[TDB+06] S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, Erhard Ploedereder, and Pascal Leroy. *Ada 2005 Reference Manual. Language and Standard Libraries*. LNCS. Springer Berlin / Heidelberg, 2006. International Standard ISO/IEC 8652/1995 (E) with Technical Corrigendum 1 and Amendment 1.

[THB06] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can we Make Operating Systems Reliable and Secure. *IEEE Computer*, 39(5):44–51, May 2006.

[vG01] Rob J. van Glabbeek. *Handbook of Process Algebra, chapter The Linear Time - Branching Time Spectrum I: The semantics of Concrete, Sequential Processes*. Elsevier, 2001.

[Wit77] Robert W. Witty. Dimensional flowcharting. *Software-Practice and Experience*, 7:553–584, 1977.

[Wit81] Robert W. Witty. Small scale software engineering. Ph.d. dissertation, Department of Computer Science, Brunel University, Uxbridge, UK, September 1981.

[YG04] Karen Yorav and Orna Grumberg. Static Analysis for State-space Reductions. *Formal Methods in System Design*, 25:67–96, 2004.

Universitat Ramon Llull

Aquesta Tesi Doctoral ha estat defensada el dia ____ d _____ de 2009

al Centre _____

de la Universitat Ramon Llull

davant el Tribunal format pels Doctors sotasignants, havent obtingut la qualificació:

President/a

_____

Vocal

_____

Vocal

_____

Vocal

_____

Secretari/ària

_____

Doctorand/a

C. Claravall, 1-3
08022 Barcelona
Tel. 936 022 200
Fax 936 022 249
E-mail: urlsc@sec.url.es
www.url.es